



# ***Algorithm***

*(noun.)*

Word used by  
programmers when  
they do not want  
to explain what  
they did.

# Advanced C++ Programming

Containers, Lambdas  
and Algorithms



# Preliminaries

# Overview & Goals

- We want to be able to express algorithms **as naturally as possible**
  - Important: *while also keeping the code easy to read*
- Lambda expressions are an important tool to achieve that
- Algorithms from the `std::algorithm` library are another
- To meaningfully use both we should first understand STL containers
  - We will start with an overview of these (no implementation details → templates)

# STL Containers

- Generic collection of common data structures
- Three categories of containers:

Sequence Containers	Associative Containers	Unordered Associative Containers
array	set	unordered_set
vector	map	unordered_map
deque	multiset	unordered_multiset
list	multimap	unordered_multimap
forward_list		

03\_01\_containers\_sequence.cpp

03\_02\_containers\_associative.cpp

03\_03\_containers\_unordered.cpp

# Container Reference & Complexity

- Reference: <http://en.cppreference.com/w/cpp/container>
- The C++ STL provides **complexity guarantees** on container operations where appropriate – examples:
  - `std::map::operator[]` has logarithmic complexity in the size of the container
  - `std::unordered_map::operator[]`: average case constant; worst case: linear in size
  - `std::vector::insert()`: linear in distance between pos and end of container

# Iterators

	Iterator Category				Operations
Contiguous Iterator	RandomAccess Iterator	Bidirectional Iterator	ForwardIterator	InputIterator	<ul style="list-style-type: none"> <li>• Read</li> <li>• Increment (without multiple passes)</li> </ul>
					<ul style="list-style-type: none"> <li>• Increment (with multiple passes)</li> </ul>
					<ul style="list-style-type: none"> <li>• Decrement</li> </ul>
					<ul style="list-style-type: none"> <li>• Random Access</li> </ul>
					<ul style="list-style-type: none"> <li>• Contiguous Storage</li> </ul>
OutputIterator					<ul style="list-style-type: none"> <li>• Write</li> <li>• Increment (without multiple passes)</li> </ul>

Iterators that satisfy the requirements of one of the first 5 categories *and* OutputIterator are called **mutable iterators**. E.g. “mutable RandomAccessIterator”

# Iterator Examples & Adaptors

- 03\_04\_iterators.cpp shows a few examples of iterator use
- *Iterator operations* allow uniform operations on many or all types of iterators
  - E.g. advance, distance
- *Iterator adaptors* create derived iterators for specific purposes
  - E.g. reverse\_iterator, back\_inserter



# Lambda Expressions



# Lambda Expression Basics

- We'll start with 03\_05\_lambda\_basics.cpp
- Lambda expressions allow **defining anonymous functions**
  - Important: *at the place where they are used*
- Return type is usually implicit
- Can be stored/used as parameters with `std::function`
  - Also using templates, and in some cases as plain function pointers

# Lambda Expression Capturing

- Lambdas are not just anonymous functions:  
**they can capture variables in their declaration scope**
- Such a construct is called a ***closure***
- Very useful in many scenarios, let's look at 03\_06\_lambda\_captures.cpp for some examples

# Lambda Expression Syntax

*[optional]* Parameter list - same as for functions, "auto" builds a generic lambda

*[optional]* Provide the exception and attribute specifications for the call

*[required]*  
Function body

**[captures] (params) specifiers exception attr -> ret { body }**

*[required]*

Comma separated list of captures

Examples:

- [a, &b] capture a by copy and b by reference
- [this] capture the current object by reference
- [&] capture all used automatic vars and *this* by reference
- [=] capture all used automatic vars by copy and *this* by reference
- [] no captures

*[optional]*

May contain one of these specifiers:

- mutable  
Specifies that the captured data may be modified
- constexpr  
explicitly specify that the call to this lambda is a constexpr

*[optional]*

Explicitly specify the return type. If this is not provided, the return type is implied by the return statements in the body.

# Implementation & Background

*The lambda expression is a prvalue expression of unique unnamed non-union non-aggregate class type, known as **closure type**, which is declared (for the purposes of ADL) in the smallest block scope, class scope, or namespace scope that contains the lambda expression.*

<http://en.cppreference.com/w/cpp/language/lambda>

- We study what this closure type looks like in 03\_07\_lambda\_implementation.cpp
- Note that *lambdas which have no captures can be converted to function pointers*
  - E.g. for use with C-style interfaces



# Standard Algorithms

# Standard Algorithms

- Most defined in `<algorithm>` :
  - Non-modifying and modifying sequence operations
  - Sorting, search and partitioning operations
  - Set and heap operations
  - Minimum, maximum and permutation operations
- Numeric operations defined in `<numeric>`
- Full reference:  
<http://en.cppreference.com/w/cpp/algorithm>

# Algorithm Examples

- The source file 03\_08\_alg\_examples.cpp shows some very simple uses of standard algorithms
- Note that many algorithms have defaults but can also be customized by **predicates**
  - Often a good use case for **lambdas**
- *Check the algorithm library before re-implementing functionality!*

# Parallel Algorithms

- Since C++17, most algorithms have an overload which allows an optional ***execution policy*** parameter
  - Options are *seq*, *par* and *par\_unseq*
  - “unseq” allows work-stealing scheduling and vectorization
- Compiler/library support currently often incomplete
  - But this should change





Conclusion

# Summary

- The C++ STL includes a large and well-specified set of **containers**
  - Sequential, associative and unordered
  - With distinct requirements on types, and performance characteristics
- Standard **algorithms** are provided to operate on these data structures
  - Or any other data structures which provide functionally equivalent iterators!
- **Lambda Expressions** are a great way to write terse predicates
  - And they also allow for **closures**, which are useful in many scenarios
  - Some care is required with captures – lifetime concerns