

# Advanced C++ Programming

**Keyword Safari** 

Preliminaries

## Overview & Goals

- Goal: We want to be fluent in advanced C++
  - This also includes reading code from any source
- There are a number of concepts which are important to understand in order to achieve this goal, but that we cannot dedicate an entire lecture to
  - Gathered in this lecture

Storage Class Specifiers

# Background

Objects (e.g. variables) in C++ feature storage and have a type of linkage

Storage Duration	
Automatic storage duration Allocated at the beginning of a block, deallocated at the end; e.g. local variables	<b>Thread</b> storage duration Allocated when a <i>thread</i> begins, deallocated when it ends
Static storage duration Allocated at the start of the <i>program</i> , deallocated at the end; e.g. global variables	<b>Dynamic</b> storage duration Explicitly allocated and deallocated by the programmer using dynamic memory allocation functions

# Background

Objects (e.g. variables) in C++ feature storage and have a type of linkage

#### Linkage

#### No linkage

Names with no linkage always generate a unique object instance. E.g. local variables

#### **Internal linkage**

If a name has internal linkage, it can be referred to within the current translation unit

#### **External linkage**

Names with external linkage can be referred to from other translation units. E.g. normal (unqualified) functions

# Storage Class Specifiers

- **static**Forces *static* or *thread* storage duration and *internal* linkage
- extern
   Forces static or thread storage duration and external linkage
- thread\_localForces thread storage duration

05\_01\_storage\_classes.cpp

## Local Static Initialization

- Local "static" variables are initialized the first time control flow passes through their declaration
  - On subsequent passes, the initialization is skipped
  - Destructors are called at program exit
- This is demonstrated in 05\_02\_static\_initialization.cpp
- *Note*: local statics introduce unexpected state
  - → Carefully consider their usage!

CV Type Qualifiers

## CV Qualifier Basics

- For any type T which is not a function or reference type, there are three more distinct types in the C++ type system:
  - const T
     A T which cannot be modified.
  - volatile T
     A T which might be modified externally (accesses are side-effects for the purpose of optimization).
  - const volatile T
     A T which cannot be modified in this context, but might be changed externally.

# CV Ordering and Implicit Conversion

Ordering

("<" = "less qualified")

*unqualified* < const

unqualified < volatile</pre>

unqualified < const volatile</pre>

const < const volatile

volatile < const volatile

Implicit conversion occurs to more qualified CV types.

E.g. unqualified → const const → const volatile

05\_03\_cv\_qualification.cpp

Additional Class Member Options

### Overview

- We already discussed virtual, override, and final, as well as const
- The keyword "static" has a different meaning for class members
- Constructors can be designated "explicit"
- In addition to const qualifiers, member functions can be Ivalue and rvalue reference qualified
- Class data members may be designated "mutable"

## Static Class Members

- In class definitions, "static" declares
   members not bound to any class instance
- It can be applied to both data members and member functions
- Simple examples are shown in 05\_04\_static\_members.cpp
- For most practical purposes, static data members act like global variables, and static member functions act like freestanding functions

# **Explicit Constructors**

- Single-argument constructors are, by default, implicitly used for conversion operations
- This concept is demonstrated in 05\_05\_explicit\_constructors.cpp
- The keyword "explicit" prevents this from occurring
- Guidelines
  - 1. By default, declare all single-argument constructors explicit, unless you really want implicit conversion to occur (e.g. building a "Complex" number from a double)
  - 2. Prefer named conversion functions over constructor conversion

## Member Function Reference Qualifiers

- As we discussed previously, member functions may be "const" qualified
  - This affects the const-ness of the this object
- In the same fashion, they can be I- or r-value reference qualified
  - This affects the value categories for this that they bind to
- The example in 05\_06\_member\_ref\_qualifiers.cpp illustrates this
  - Also shows an important use case which can significantly increase the safety of a library

## Mutable Data Members

- Data members may be mutable-qualified to allow modifying them in constqualified member functions
- This should only be used for members which do not change the externally visible state of the class
- Example use cases: caching, lazy evaluation
   Basic idea in 05\_07\_mutable\_data\_members.cpp
- Beware:
   These types of caching can easily lead to race conditions in parallel execution

Constexpr

## Constexpr Basics

- constexpr is a specifier that can be applied to functions and variables and indicates that it is possible to evaluate their value at compile time
- Let's look at an example in 05\_08\_constexpr.cpp
- The idea is to make a declaration of intent,
   and codify this requirement as part of an interface
- There are a number of constraints on what can be declared constexpr

## Constexpr Constraints

#### Constexpr functions

- Must not be virtual
- Return and parameter types must be literal types
- May not contain some kinds of statements

http://en.cppreference.com/w/cpp/concept/LiteralType

#### **Prohibited Statements**

asm declarations

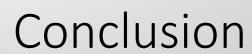
goto

labels (other than case and default)

try blocks

#### variable definitions

- of Non-literal type; or
- with static or thread storage; or
- with no initialization



# Summary

- Understand variable and object storage, linkage, and CV qualification
- For classes
  - Distinguish static and non-static members
  - Correctly use const, mutable, and reference qualification
  - Understand the reasoning for explicit single-argument constructors
- Use constexpr for compile-time computation