C++ Advanced

Casting Review

Polymorphism Review

Multiple Inheritance

RValue reference

Move Semantics

Generalized Constant expression

Lesson Objectives

- Review C++ style casts
- Review Polymorphism
- Discuss Multiple Inheritance
- Understand RValue reference and Move semantics
- Generalized Constant Expressions

Run Time Type Identification

- Because C Style casts are inherently unsafe, a lot of home grown type-checking was invented
- C++11 standardized this
  - typeid() operator - Used where the dynamic type of a polymorphic object must be known
  - Returns std::type_info
  - Sits at the core of C++ style casts

C++ Style Casts

- Replace the “c style” casts
  - Don’t use: they’re ambiguous
- const_cast<type> - adds or removes const-ness at compile time
- static_cast<type> - changes between types, at run time
  - Unsafe, no runtime checking for polymorphism
  - Cannot add or remove references

C++ Casts Continued

- dynamic_cast<type> - safely converts pointers or references throughout inheritance tree, at run time
  - Is “safe” due to runtime type checking
    - If cast to type& fails, throws bad_cast
    - If cast to type* fails, returns a nullptr
- reinterpret_cast<type> - used for converting pointers at compile time
  - Converts T* to int
  - Converts any T1* to any other T2*

Polymorphism

- Derived D : Virtual_Base B
  - A class is virtual if it has at least 1 virtual member
  - Calls B's ctor, then Ds
  - In memory {[Bs Members]Ds Members}
  - If D overrides B's member
    - vTable so that the overridden member both "point" to the same fn addr
    - If B allocates memory, must have a virtual destructor, ~D() dtor must deallocate B's memory, or memory will leak

Multiple Inheritance

- C++ allows multiple base classes
  - Avoid this - it inherently violates Liskov's principle, probably represents a "has a"
  - Java and .Net don't (can implement multiple interfaces)
  - Industry has moved to shallower trees
- D1: B and D2: B and DD: D1, D2?
  - In memory: ({[B]D1}{[B]D2}DD), if we call some member of B, what happens?
  - Compiler can't know "which B?": error

Multiple Inheritance, Cont'

- Like functions, entire classes can be virtually inherited
  - Creates V-Table entry for the entire class
- D1: virtual B; D2: virtual B; DD : D1, D2
  - When we instantiate DD, both D1 and D2 get a V-Table entry that points to the same B
  - Called "Diamond Shaped Inheritance"
  - DD() must call B(), calls from D1() D2() omitted unless constructed alone
- Can 'share' a f'n across siblings via pure virtual base

R Value Reference: Class&&

- lvalues - multiple statement duration, can take address
- rvalue - single statement duration, typically can't take address
  - Most common: a functions return value
- foo(Widget&& w) - an R value reference
  - Inside foo, the parm w is lvalue
  - at the calling location the arguement is rvalue
- lvalues get copied, rvalues get moved

Universal Reference: T&&

- Can be either lvalue or rvalue, const or not, volitile or not
  - Myers dubs them Universal Reverence
- 2 times '&&' is universal
  - auto&& var2 = var1;
  - template<typename T> foo(T&& param);
- Both use type deduction
  - If the argument of foo is an lvalue, param is of type T&, if rvalue, then T&&

Class&& vs T&&

- Consider std::vector
  - template<class T, class Allocator = allocator<T>
  - push_back(T&& temp)
    - Looks like a universal reference, but T is known at compile time, so is always rvalue reference
  - emplace_back(Args&& ...args)
    - Types of args not known at compile time, universal reference
    - Takes the args and passes them to the ctor of T(...)

Move Semantics

- Allows the compiler to “steal” the value held in an rvalue reference
  - Prevents the need to generate many temporary objects
  - No difference for POD classes (like Point)
  - Useful for dynamic memory classes: swap the pointer, not copy the memory
- The value of something that has been moved from is undefined!

Move ctor and move assignment

- Widget(Widget&& other)
- Widget& operator = (Widget&& other)
  - Both must be sure to move any members that are composite objects
- “Rule of 3” now “Rule of 5”
  - Self Documenting code - show intention
- Often: myPoints.push_back(Point{x,y})
- Bad practice to move from an lvalue
  - The object is still “good”, hasn’t been destroyed, etc., but value is undefined

std::move and std::forward

- std::move is used to cast to an rvalue ref
- std::forward used to handle universal references

Generalized Constant Expressions: constexpr

- New keyword constexpr
  - Differs from const which applies at runtime and indicates a value cannot change
  - constexpr is a compile time directive
- Tells the compiler to attempt to evaluate an expression at compile time
- Variables, Functions, Ctors can all be constexpr

C++11 constexpr

- Variable
  - Immediately initialized full statement must be constexpr
- Function
  - Not virtual, all params literal types
  - Should be no-op with the exception of a single return of literal type
  - Can use recursion
- Constructor : same as fn plus
  - All base must be constexpr and fully initialized

C++14 constexpr: improvements

- Functions same rules for params and return value
  - Body can be a normal function except
    - No try/catch, goto, asm (don't do this anyway), cannot instantiate a nonLiteral, cannot have uninitialized variables, cannot define variables of static or thread lifetime
- Ctors : same as functions, with same rules for base classes as C++11.