Polymorphism

10: Polymorphism

Copyright © 2001-2002 by J. S. Sumey

Polymorphism Concept
- refers to selection between multiple meanings of a given function (or operator) at run-time
  - applies to an inheritance hierarchy
- polymorphism: run-time binding of a function's name to its code (entry pt.)
  - as opposed to compile-time
  - concept from pure OO languages, Smalltalk
- entry point – starting address of a function

Polymorphism Goal
- to have user-defined ADTs and inheritance hierarchies behave as if they were native to the language
  - is also a principle of OOP
- examples:
  - complex numbers
  - dates

Review: Overloaded Functions
- recall def: providing multiple meanings (versions) for a given function name
- recall: function signature
  - combination of: function name, number of parameters, types of parameters
- compiler uses a signature-matching algorithm to resolve function calls
  - resolved at compile-time

Polymorphic Functions
- a fn. is polymorphic if its invocation is bound to its entry point at run-time
- 3 requirements:
  - must be part of an inheritance hierarchy
  - hierarchy must include at least one virtual method having identical signature
  - must involve a pointer (or reference) to a base class

Virtual Functions – 1
- a member function may be declared virtual in a base class (BC) which may then be redefined (overridden) in the derived class (DC)
  - allows derived class to more appropriately define the meaning of the function
  - thus, invoking a virtual fn. on a particular object selects the correct version
**Polymorphism**

### Virtual Functions – 2
- **rules for using `virtual`:**
  - only applies to *object methods*
  - no top-level functions or class (static) methods
  - precedes function header in the base class
  - only used in the declaration
  - overridden inherited function in a derived class is automatically `virtual`

### Base Class Pointers
- a BC pointer can be considered a “generic” pointer in that it may also access any subclass object derived from the BC
- does not require typecasts
- ex: *Shapes* hierarchy

### Virtual Functions – 3
- run-time binding via the **Vtable**
  - neat concept, but certainly increases overhead in both space and time
- **constructors** must not be virtual
  - because subclass ctor first invokes base class ctor during instantiation
- **destructors** may (and should) be virtual
  - especially with dynamic allocation

### Clarifications
- **name overloading**
  - always resolved at compile-time
- **name overriding**
  - resolved at run-time if the fn. is `virtual`
  - resolved at compile-time if not `virtual`
- **name hiding**
  - occurs when a DC overrides a BC method
  - can always use scope resolution operator

### Abstract Base Classes
- is used only to derive subclasses, not to declare objects directly
  - i.e. no objects can instantiate from an ABC
- defines at least one **pure virtual** member function whose implementation is deferred to subclasses
  - typically all are `public`
  - syntax: `virtual fn_prototype = 0;`

### Uses of Abstract Base Classes
- to specify a shared interface
  - all subclasses must implement all pure virtual functions
- to specify design requirements
  - for internal code maintenance or standard behavior throughout inheritance hierarchy
Polymorphism

Abstract Base Class Examples
- “windows” in a GUI environment
- Microsoft COM (Component Object Model) application building infrastructure

OO Design with Inheritance
- a given problem domain should be modeled after the real world by incorporating inheritance
  - **OOP Design Methodology:**
    1. decide on appropriate ADTs
    2. design software based on their relatedness using inheritance to share code
    3. use virtual functions for polymorphism

Run-Time Type Identification
- **RTTI** - a C++ mechanism to allow the programmer to perform run-time operations based on type information
  - uses:
    - determine type of an object
    - verify safeness of type conversions (casts)

The typeid Operator
- typeid can be used to determine the type of an expression or typename
  - typically used in equality comparison tests
  - return value of typeid is “arbitrary”
  - must #include <typeinfo>

typeid Example
- given: int k;
  Date d;
  typeid(k) == typeid(int) T
  typeid(k) == typeid(float) F
  typeid(d) == typeid(long) F
  typeid(d) == typeid(Date) T
  typeid(135) == typeid(int) T

Static Type Casts
- recall: static_cast is used to make type casts at compile-time
  - ex: float avg = sum / static_cast<float>(n);
- what about type casts in an inheritance hierarchy?
  - ex: a pointer to a BC object that has no DC augmented members...

CSC333 Object Oriented Programming
Polymorphism

Dynamic Type Casts
- A **dynamic_cast** is performed from a source type to a target type with safety at run-time.
- Source type must be polymorphic (at least 1 virtual function).
- Target type must be a pointer or reference.
- Returns source (pointer) if cast is type safe, or NULL/False if not.

Dynamic Cast Example
- Given: DC derived from BC

  ```cpp
  DC *p = dynamic_cast<DC*>(new BC);
  if (!p)
     cerr << "unsafe type cast";
  ...
  ```
- This would be unsafe if DC adds a member function not in BC.