Polymorphism

• “Polymorphism is the genie in OOP who takes instruction from clients and properly interprets their wishes.”

Definition: many shape/forms
– Thus, a polymorphic function has many forms
– Polymorphism can be thought of as the proper resolution of operation based on type

Types of Polymorphism

• ad hoc polymorphism
  – Also called overloading in which the same function (name) has many implementations
  – Examples:
    ```cpp
    string s = test;
    int b = 10;
    cout << s;
    cout << b;
    CountdownTimer ct1;
    CountdownTimer ct2 (60, NOT_RUNNING);
    int i1, i2;
    float f1, f2;
    ... i1/i2;
    f1/f2;
    ```

• parameterized polymorphism
  – name of an operation in a parameterized class stands for different operations based on different parameterized types
  – based on the template function of C++
  – Example:
    ```cpp
    vector<Shape*> shape_stack;
    vector<Employee> emp_stack;
    ... shape_stack[1]->pop();
    emp_stack[2]->pop();
    ```

Pointer Notation for Function Calls

• Objects have a syntax for accessing members
  = object.member // we don’t do this, use accessor
  = object.member_function()
• Object pointers
  = object_ptr -> member
  = object_ptr -> member_function()
• Avoids the awkward dereferencing syntax
  = (*object_ptr).member
  = (*object_ptr).member_function()

Pure Polymorphism

• pure polymorphism
  – selection of the appropriate function in a class hierarchy based on the “true” type of the object
  – this is generally what people mean when they talk about polymorphism

```cpp
Shape* fig = new Shape();
fig[0] = new Point(100, 100);
fig[1] = new Rectangle(new Point(40, 30), new Point(60, 70));
fig[2] = new FilledRect(new Point(80, 40), new Point(130, 160), Color.red);
fig[3] = new ScalableText(new Point(500, 100), "Hi", 36);
fig[4] = new Text(new Point(160, 150), "Hello" Sticky);...
fig[4].scale(new Point(8, 8), 0.9);
```
Pure Polymorphism in Java

- Java automatically provides pure polymorphism
  - type resolution in class hierarchies
  - used (but not mentioned) in our previous example
- What are the resolution rules?
  - In a class hierarchy, what happens if a class does not implement a function, but one of its parents does?
  - Let’s find out

Heterogeneous Collections

- Typically applications assemble “like” objects
  - shapes, employees, cars have all been examples
  - specialization makes the heterogeneous
    - added function or extra state
    - overloaded functions
    - restrictions
- In Java, heterogeneous collections are implemented as arrays of object (references) of a base type
  - all Java objects are references
- In C++, heterogeneous collections are typically implemented
  - arrays of pointers
  - with parameterized (template) classes such as vector<Shape*> shapes;

The Problem of Type Resolution

- So we have a roster of baseball players
- The roster is constructed as a vector of pointers to players
- We want to print out last years stats for each player
  - but stats vary by position
- Let’s look at the class hierarchy

What the Program Wants to Do

- Iterate over the roster of players and print statistics
  - statistics should be customized to the the specialized object
- Nice idea, but how do we accomplish this
- The code below would seem to print player::stats() when we want to print the specialized function for each type of player

```c++
vector<Player*> parray;
...
for ( int i = 0; i < parray.size(); i++)
{
  parray[i]->print_stats();
}
```

Pure Polymorphism in C++

- No automatic support for pure polymorphism in C++
  - by default, pointers resolve to the pointer type and invoke the function of that type
- Example
  - All objects are “truncated” to the base class only
    - this is called slicing away
    - both function and state are subject to slicing
Automatic Support or Not?

- The dangers of not supporting pure polymorphism in C++ are obvious
  - one doesn’t get the expected results
- What are the dangers of Java supporting pure polymorphism by default?
  - Recall, Java can always call the super function explicitly

The Bad Idea – Type Selection

- Add a “type” field to the base class which encodes the actual type of the object in a heterogeneous collection

```cpp
class Person
{
public:
    enum SubObj { FACULTY, STUDENT, BADSTUDENT };
    SubObj _type;
    string _name;
};
```

The Bad Idea – Type Selection

- Add a “type” field to the base class which encodes the actual type of the object in a heterogeneous collection
- Why is this such a terrible idea?
  - tedious, has to be performed for every type
  - cannot extend base classes without modifying existing code
  - types cannot be checked by the compiler
- Thus, error prone and a maintenance nightmare

Static and Dynamic Binding

- Static binding
  - the compiler resolves the type of a variable at compile time
  - based on type (including pointer type)
- Dynamic binding
  - runtime evaluation of the “true” type of an object
  - this is what Java does by default
- Pure polymorphism is dynamic binding
  - overloading is (generally) static binding
  - parametric polymorphism is either static or dynamic

Virtual Functions in C++

- The keyword virtual when added to a function instructs the compiler that the function is polymorphic
  - the compiler builds dynamic binding of the function
- Virtual functions allow C++ functions to exhibit pure polymorphism in class hierarchies
  - just like Java functions
- Derived classes override the virtual function to provide specialized behavior
  - overloaded function must have the exact same function prototype

```cpp
void Person::print()
{
    if (_type == FACULTY)
        Faculty::print();
    else if (_type == STUDENT)
        Student::print();
    else if (_type == BADSTUDENT)
        BadStudent::print();
}
```
Virtual Functions

- Derived classes need not override virtual functions
  - they can use the base class function
- Base class must provide a function definition of a virtual function
- Compiler retains state about the type of object
  - resolves their real identity at runtime
- Performance issues
  - this is a level of indirection and a little bit of state
  - don’t worry about it
- Examples

Virtual Functions in the Derived Class

- Do virtual functions in a derived class need to be declared as virtual as well?
  - no, i.e. no virtual specifier is necessary to overload the virtual function
- But, what about classes derived from a derived class that does not declare its overloaded function virtual?
  - let’s find out

Virtual Functions in the Derived Class

- Do virtual functions in a derived class need to be declared as virtual as well?
  - no, i.e. no virtual specifier is necessary to overload the virtual function
- But, what about classes derived from a derived class that does not declare its overloaded function virtual?
  - let’s find out
  - declared or not, they are virtual, i.e. the virtual (pure polymorphic) feature is passed on to all subsequent derived classes

Casting

- In both Java and C++, we convert pointers/references to objects of a derived class to pointers/references to objects of the base class
- This is an example of casting
- Casting is the name for type conversion
  - changing one type into another type
  - also called a type coercion
- Casting is often dangerous many types cannot be converted into other types
  - e.g. pointers into integers, etc.
- Never cast unless you have to
  - prefer to use polymorphism

Casting Rules

- It is always safe and customary practice to cast a derived class into its base/super class
  - this is an implicit cast, no specific indication of the type conversion
- C++ provides a type-safe, dynamically checked cast
  - this is an explicit cast

```
Person * pp;
...
Employee * ep = dynamic_cast<Employee *> pp;
// returns a pointer if pp is an Employee
// returns 0 (null pointer) otherwise
if ( ep != 0 )
{
...
```

Deprecated C-Style Casting

- C (and thus C++) provided a syntax for casting
- This is now frowned upon because:
  - not checked (statically evaluated)
  - hard to identify in code, i.e. less explicit
- Stroustrup says “should have been deprecated”
- This is also the casting style for java

```
Person * pp;
...
Employee * ep = (Employee *) pp;
// always returns a pointer
// dangerous if pp is not an Employee
```

4
More on Casting

- There’s a lot more to it
  - classes can provide specific casting methods
  - casting can be done with constructors and anonymous objects
- Just know that it exists and...
- don’t do it, use polymorphism

Abstract Base Classes

- Many classes are useful both as base classes for a class hierarchy and as concrete classes from which objects can be constructed
  - the Person class in our example is meaningful
- Many classes are good abstractions for a base class, but not meaningful on their own
  - consider the shape from last week’s example
  - Declaring an instance of this class does not make sense
  - Shape *s; // silly shapeless shape

Abstract Base Classes (Java)

- Java allows us to declare classes (and functions) abstract
  - cannot create an instance of an abstract class
  - abstract functions have no definition
  - an abstract class can have data and methods
- Shape was an abstract class in last week’s example
  - had no member data or member functions
  - but, it could have

Abstract Base Classes (C++)

- C++ does not allow a class to be declared abstract
- Rather, a class is abstract if it contains a pure virtual function
  - see syntax example (`virtual ... = 0`)
  - no objects of this class can be instantiated
  - abstract classes can have member data and defined member functions
- Example

```cpp
class Shape
{
    virtual void scale ( Point center, double scaleFactor ) = 0;
    virtual void plot ( Graphics& G ) = 0;
};
```

Errata NULL

- For pointers, Horstmann recommends the use of NULL
  - “because it carries more information for the human reader.”
  - `Employee * pe = NULL;`
- As of now, NULL has been deprecated and pointers should be assigned to 0
  - `Employee * pe = 0;`
  - 0 is preferred because it is a literal, not a symbol/name
  - not all C++ compilers like NULL anymore