Polymorphism

- Polymorphism means *many* (poly) *shapes* (morph)
- Done using inheritance
  - No concept of interfaces

```cpp
class Person{
public:
  int ssn;
  void whoAmI() {
    cout << "Person" << endl;
  }
};

class Employee: public Person {
public:
  int wages;
};
```

Employee is A Person
- Employee has all members/methods of Person

Things not inherited
- A derived class inherits every member of a base class except:
  - its constructor and its destructor
  - its operator=() members
  - its friends
    - A friend class has access to a class’s private and protected members without being a member

Default constructor and destructor
- Default constructor and its destructor are always called when a new object of a derived class is created or destroyed

- If the base class has no default constructor i.e. you have an overloaded constructor, you can specify it in each constructor definition of the derived class:
  ```cpp
derived_constructor_name(parameters) : base_constructor_name (parameters) {...}
```
- In Java this is done by the call to super(..)
Overriding

- Overriding occurs when a child class has a method with the exact same type signature as one of the parent class methods
- To do work specific to the class
- Example: Employee class does
  ```cpp
  void whoAmI() {
    cout << "Employee" << endl;
  }
  ```
- Binding is the process of deciding whether to execute the parent’s version or the child’s version

Static Binding

- Happens when the virtual keyword is not used
- Decision is made at compile time based on the type of the variable
  ```cpp
  Employee * e = new Employee();
  Person * p = e;
  e->whoAmI(); //Employee
  p->whoAmI(); //Person
  ```

Dynamic Binding

- Virtual is used to declare the method
  ```cpp
  virtual void whoAmI() {
    cout << "Person" << endl;
  }
  ```
- The binding decision is made at runtime based on the type of object
  ```cpp
  Employee * e = new Employee();
  Person * p = e;
  e->whoAmI(); //Employee
  p->whoAmI(); //Employee
  ```

Overriding Limitations

- Works only with heap-resident values
  ```cpp
  Person p = *e;
  p.whoAmI(); //Person
  ```
- Child classes cannot change the type of binding
  - Once virtual is used for method in the parent class, the method is always virtual even if the keyword is missing in the child class
  - A method not declared virtual in the parent class cannot be made virtual in the child class
Abstract Classes

- A class that contains pure virtual methods
  - A pure virtual method does not have a body
    ```cpp
    class Animal{
    public:
        virtual void speak() = 0;  // assigned a null value
    }
    ```
  - Abstract base classes can only be used through inheritance
    - Cannot create an instance of an abstract class

Downcasting

- If a pointer to parent is type casted to point to a child the behavior can be unpredictable
  - C++ does not perform run time type checking
    ```cpp
    Person * p = new Employee();
    Employee * e = (Employee *) p;
    ```
  - Note that only the data type associated with the pointer is changed
    - Object that the pointer points at is not changed

Downcasting contd.

- Run-Time Type Information System (RTTI) provides a mechanism to protect against this using dynamic_cast
  - Returns a valid pointer if the case was successful, and 0 if not successful
    ```cpp
    Person * p = new Employee();
    Employee * e = dynamic_cast<Employee *> (p);
    if(e){
        cout << “Employee Type” << endl;
    } else{
        cout << “Not Employee Type” << endl;
    }
    ```

Forest not a Tree

- No C++ class is the ancestor of all classes
- A void pointer can be used as a generic pointer
  ```cpp
  Person * p = new Employee();
  void * v = p;
  ```
- A dynamic cast is needed to safely change the void pointer to the original type
  ```cpp
  Employee * e = dynamic_cast<Employee *> (v);
  ```
How is dynamic dispatch carried out?
- Using a virtual function table
  - Table of pointers to its member functions

class A {
    struct _a_vtbl {
        int x;
        virtual void foo(void);
        virtual int baz(int arg);
    };

    struct _class_A {
        struct _a_vtbl * vtbl;
        int x;
    };
}

Recall:
Class = fancy struct

Dynamic dispatch
- C++ code:
  A* a = (something);
  a->foo();

- a->foo() call translates into
  a->vtbl->foo();

- Why a vtable?
  - Allow more methods in children

Class is fancy struct
struct _a_vtbl the_class_A_vtable;

- Program starts:
  the_class_A_vtable.foo = A's foo
  the_class_A_vtable.baz = A's baz

- new A()

struct _class_A * a = (struct _class_A *)malloc(sizeof(struct _class_A));
a->vtbl = &the_class_A_vtable;

Inheritance
class A{
    int x;
    virtual void foo() {...}
};

class B : public A {
    int y;
    virtual void foo() {...}
    virtual void bar() {...}
};
Observe

- Calling virtual member function:
  - Follow ptr to vtbl
  - Index vtbl for fn ptr
  - Call indirect
- B “looks like” A:
  - A* foo = new B();

Having the ability to choose virtual or non-virtual

- Virtual can affect performance
  - Constructor of an object containing virtual functions must initialize the vptr table
  - Virtual functions are called using pointer indirection, which results in a few extra instructions per method invocation as compared to non-virtual method invocation
  - Virtual functions whose resolution is only known at runtime cannot be inlined
    - In general, when you claim to inline a function, it does mean it always gets inlined. Compiler may or may not inline it.

Difference between C++ and Java

- Java
  - Vtables of an object are initialized to the proper vtable for the class being constructed immediately by the object constructor
  - If you call an overridden method in any of the constructors, it will dispatch to the method in the class being created
- C++
  - Each constructor sets the object’s vtable to the vtable of the type the object “is so far”
  - Implies that while in the parent object’s super constructor, any functions will be called as if the object was only the parent type

Restricting Access

- class B: public A
  - Specifies no change on access of members of A
- class B: protected A
  - Specifies public members become protected
- class B: private A
  - Specifies all inherited members appear private
- Restricts polymorphism:
  - A* foo = new B(); /* with protected or private inheritance*/
  - Error: ‘A’ is an inaccessible base of ‘B’
Virtual Destructors

- If any virtual methods are used, the destructor should be virtual to ensure that both the parent and child destructors are called.

- Example: In the header for each class
  ```
  virtual ~Person() ;
  virtual ~Employee();
  ```

UML Class diagram relationship

- Generalization
  - Kind of relationship that indicates inheritance
  - E.g. Parent is a generalization of Child1 and Child2

  ![UML Class Diagram]

  - Pure virtual functions have their names in italics
  - Abstract classes are represented by having their class name in italics

Name Resolution

```
class A{
    public:
    void foo(int i);
};

class B: public A{
    public:
    void foo(A & a);
};
```

The following code will not compile:
```
B * b = new B();
b->foo(5);
```

Name Resolution contd..

- The compiler could not find foo(int i)
  - There are three name scopes
    - One for each class
    - The global scope
  - The scopes are nested inside each other
    - B is in A's scope
    - A is in the global scope
  - The compiler first looks for the innermost scope that has the function foo, which will be B
  - It then looks for foo function that takes an single integer but B does not have one
Name Resolution contd..

- The problem can be fixed by adding `foo(int i)` to class B:
  ```java
  void foo(int i){
    A :: foo(i);
  }
  ```

- The new method will simply call the same method in the parent class.

Multiple Inheritance

- C++ allows inheriting from more than one parent class `DerivedClass : public class1, public class2, ..{}`

- However, it can lead to some problems:
  - E.g., Consider a Person class with `resolveConflict()` method.

Problems with Multiple Inheritance

- Governor might override `resolveConflict()` to mediate a solution.

- Terminator might override `resolveConflict()` a slightly differently.

- What is a Governor to do?
  - Which parent class `resolveConflict()` method does it inherit?

- This known as Diamond Inheritance.

Java's solution to Multiple Inheritance problem

- To avoid diamond inheritance Java does not allow extending more than one class in Java.

- Instead, we use an interface which allows us to specify behavior that we want to have without actually specifying the exact implementation.