Polymorphism - why?

- Eliminate the need to test for class type i.e., typeid() && the switch statement in some coding problems
- Derived classes can implement operations differently while retaining common interface
- Treats all derived objects as objects of the base class
- Extensibility without redesign
- Add new object with minimal recoding.

The ability of inheritance-related objects to respond differently to the same message.

Polymorphism - why?(cont’d)

Example: “Problem without polymorphism”
```cpp
Employee  e;  Employee*  ePtr = &e;
HourlyEmp eh;  HourlyEmp* ehPtr = &eh;
// calls in code - assume base & derived class have overloaded
// Print() methods
ePtr->Print();  // calls base class method
ehPtr->Print();  // calls derived class method
if (A == B) ePtr = ehPtr;  // allowable implicit conversion
ePtr->Print();  // still calls base class method, need cast or scope
// compiler doesn’t know about conversion, runtime does
```

Key - dynamic binding - resolving calls at run time, not compile time.
Polymorphism - why?(cont’d)

Example: “Solution without Polymorphism”

```cpp
OutputInfo(Employee& emp) { // can pass Employee or HourlyEmp
    if (typeid(emp) == typeid(Employee&))
        emp.Print();
    else if (typeid(emp) == typeid(HourlyEmp&))
        static_cast<HourlyEmp&>(emp).Print();
    else cout << "Employee Error";} // end of function
```

Polymorphism - how? - syntax

```cpp
class Employee {  
public:
    virtual void print(); // reserved word virtual
    // rest of class members
};  // end of class

class HourlyEmp : public Employee {  
public:
    void print(); // just like other methods
    // rest of class members
};  // end of class

Employee* e = new HourlyEmp; // creates HourlyEmp
e->Print(); // calls HourlyEmp print method
```

Polymorphism - how? - operationally

```cpp
Employee e;  HourlyEmp eh;
Employee& eRef = &e; HourlyEmp& ehRef = &eh;

OutputInfo(Employee& anEmp) {  
    anEmp.Print();
}  // will call Employee print() - runtime dec.
OutputInfo(eRef);   // will call Employee print() - runtime dec.
```

Variables created at runtime - pointers & reference vars.
Compiler doesn’t decide - decide at runtime instead
Gain - code flexibility; Loss - slower execution
Pure virtual functions

A base pure virtual functions does not exist
⇒ Place holders “prototypes” for derived class
⇒ Derived class must implement pure virtual base functions or declare them virtual
⇒ Syntax uses “= 0”
  e.g., virtual Print(void) = 0;

Example:
class Employee { public:
  virtual Print(void) = 0; // pure virtual
  // rest of class members
};

Abstract Class

Class is called abstract if it has one or more pure virtual functions
⇒ Must be a base class - i.e., must be used in some derived class
⇒ Abstract classes are ideal for enabling users to adapt the class to their needs - extensibility
For example, one might create an abstract base class of Employee which would enable users to create their own derived classes i.e., Hourly, Salaried, Student, Temporary, Consultant, etc. Each user could create only those classes and design them to suit his business.

Exception Handling

Inadequate error handling has always been a problem in languages.
⇒ Usually implemented after the fact.
⇒ Doesn’t give user much choice.
⇒ Must handle problem at point in code where it occurs.
⇒ Errors are individual and type can vary widely
Syntax - try, throw and catch

Example:
```
try
  { // if (divisor != 0)
    if (divisor != 0) percent = num100/divisor;
  } // else

catch
  (char* msg)
  { cerr << msg; // additional actions as needed }
// end catch
// continue on with additional code.
```

Exception Handling - features

- Multiple catch calls
- Overload catch by different signatures
- Selects first one which fits - implicit conversion
- Uncaught exceptions call terminate() which ends program - you can provide a terminate function if needed
- Exception can be thrown in one function and caught in a calling function - nested - execution starts at next catch statement(s).

Exception Handling - an exception class

Example:
```
class Divide_Error {
private:
  long div, num;
public:
  Divide_Error (long a, long b) : div(a), num(b) { };
  long GetDivisor() { return div;};
  long GetNum() { return num;}; // end class
}

double DivideIt (long a, long b) { 
  if (a == 0) throw Divide_Error(a, b);
  else return (double)b/a; // end function
}

try
  { double getNum = DivideIt(0, 25); // try block error
  catch (Divide_Error& error) { // handle error as needed }
```