# Chapter 8: Polymorphism

CSE 2010 - Week 12

# What is Polymorphism?

**Polymorphism:** A mechanism in object-oriented programming that gives us the ability to handle objects of different types at the same time. In C++ we do this by implementing several versions of a function, each in separate classes.

- Literal definition of polymorphism: having many forms.
- This is different from function overloading or overriding, which depends on the parameters.

Polymorphism is sometimes easier to learn by first looking at an example....

## Clocks Example Program: A program that uses local clocks (base class) and travel clocks (derived class).

Base Class: Clock

```h
/*
 * Filename: clock.h
 * Definition of the Clock class.
 * Clock: base class that can tell the current local time
 * In the constructor, you can set the format to either
 * "military format"  or "am/pm"
 */
#ifndef CLOCK_H
#define CLOCK_H

#include <string>
#include <iostream>
using namespace std;

class Clock{
        private:
                bool military;//determines format of time
        public:
                Clock(bool use_military);
                ~Clock();
                string get_location() const;
                int get_hours() const;
                int get_minutes() const;
                bool is_military() const;
};
#endif
```

```cpp
/*
 * Filename: clock.cpp
 * Clock class member function definitions
 */
#include "ccc_time.h"
#include "clock.h"
//constructor
Clock::Clock(bool use_military): military(use_military){
}
//return location
string Clock::get_location() const{return "Local: San Bernardino, CA";}
//destructor
Clock::~Clock(){cout << "Destroying Clock.\n";}
//return hours
int Clock::get_hours() const{
   Time now;//creates Time object from ccc_time.h
   int hours = now.get_hours();
   //hours by default is returned in military format
   //if military = true, we just return the current hours
   if (military == true)
           return hours;
   //if not military and hours = 0, then it is midnight (12am)
   if (hours == 0)
      return 12;
   //if hours > 12, then it is pm and we must subtract 12
   else if (hours > 12)
      return hours - 12;
   else
      return hours;//<= 12 can be returned as is
}
//return minutes
int Clock::get_minutes() const{
   Time now;//creates Time object
   return now.get_minutes();//returns minutes
}
bool Clock::is_military() const{
   return military;//returns true or false
}
```

# Derived Class: TravelClock

```h
/*
 *Filename: travelclock.h
 * TravelClock Class Definition
 *
 * Constructs a travel clock that can tell the time at a specified location.
 *     mil=true if the clock uses military format
 *     loc= the location
 *     diff=the time difference from the local time
 */
#include <string>

using namespace std;

#include "clock.h"

#ifndef TRAVELCLOCK_H
#define TRAVELCLOCK_H
class TravelClock : public Clock//inherits military data member
{
        private:
                string location;
                int time_difference;
        public:
                TravelClock(bool mil, string loc, int diff);
                ~TravelClock();
                string get_location() const;
                int get_hours() const;
};
#endif
```

```cpp
/*
 * Filename: travelclock.cpp
 * TravelClock class member function definitions
 */
#include "travelclock.h"
//constructor
TravelClock::TravelClock(bool mil, string loc, int diff)
   : Clock(mil), location(loc), time_difference(diff){
   while (time_difference < 0)//need positive value
      time_difference = time_difference + 24;
}
//destrcutor
TravelClock::~TravelClock(){
        cout << "Destroying Travel Clock.\n";
}
//return location
string TravelClock::get_location() const{
   return location;
}

//return hours with time difference
int TravelClock::get_hours() const{
   int hour = Clock::get_hours();
   if (is_military())
      return (hour + time_difference) % 24;
   else//am/pm
   {
      hour = (hour + time_difference) % 12;
      if (hour == 0) return 12;
      else return hour;
   }
}
```

# Test Program for Clock and TravelClock Classes

```cpp
/* Filename: main.cpp
* Program that creates some local and travel clocks
*/
#include <iostream>
#include <iomanip>
#include "travelclock.h"
using namespace std;

int main()
{

   //declare Clock object for local time
   Clock clock1(false);
   //declare two Travelclock objects
   TravelClock clock2(true, "Zacatecas", 2);
   TravelClock clock3(false, "Tokyo", 17);

   //display information about each clock
   //notice we have overridden functions, so the compiler
   //will use the function associated with each object
   cout << clock1.get_location() << " time is "
      << clock1.get_hours() << ":"
      << setw(2) << setfill('0') //fills empty spaces with 0
      << clock1.get_minutes()
      << setfill(' ') << "\n";//fills empty spaces with ' '

   cout << clock2.get_location() << " time is "
      << clock2.get_hours() << ":"
      << setw(2) << setfill('0')
      << clock2.get_minutes()
      << setfill(' ') << "\n";

   cout << clock3.get_location() << " time is "
      << clock3.get_hours() << ":"
      << setw(2) << setfill('0')
      << clock3.get_minutes()
      << setfill(' ') << "\n";
   return 0;
}
```

Output:

```
Local: San Bernardino, CA time is 3:22
Zacatecas time is 17:22
Tokyo time is 8:22

Destroying Travel Clock.
Destroying Clock.
Destroying Travel Clock.
Destroying Clock.
Destroying Clock.
```

Notice that there is a lot of repetitive code when it comes to displaying clock information.

Let's make our code look nicer by sticking all of our objects into a vector and using a loop to call the functions repeatedly.

# Storing derived objects with base objects

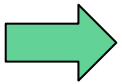
```
/*
 * Remove repetitive code
 * Create a vector of Clock objects to store all clocks
 */
vector <Clock> clocks;
//create clock objects and add to vector
clocks.push_back(Clock(false));
clocks.push_back(TravelClock(true, "Zacatecas", 2));
clocks.push_back(TravelClock(false, "Tokyo", 17));

//use each object in the vector to invoke functions
for(int i = 0; i < clocks.size(); i++){
        cout << clocks[i].get_location() << " time is "
             << clocks[i].get_hours() << ":"
             << setw(2) << setfill('0')
             << clocks[i].get_minutes()
             << setfill(' ') << "\n";
}
```

- The compiler will not give us any errors when we try to store a TravelClock into a Clock vector, but we encounter problems with how the vector is allocating memory.
  - For each object, the vector only allocates space for one attribute, “military”.
  - For the TravelClock objects with additional attributes, the vector will “slice away” the additional TravelClock attributes (“location”, “time_difference”)

# Storing derived objects with base objects

- Using a vector of pointers, we can simply store the memory addresses of the objects stored in heap memory.

```
/*
 * Remove repetitive code
 * Create a vector of Clock objects to store all clocks
 * No compiler errors, but doesn't work :(
 */
vector <Clock> clocks;
//create clock objects and add to vector
clocks.push_back(Clock(false));
clocks.push_back(TravelClock(true, "Zacatecas", 2));
clocks.push_back(TravelClock(false, "Tokyo", 17));

//use each object in the vector to invoke functions
//We will see that the TravelClock information would be spliced
for(int i = 0; i < clocks.size(); i++){
        cout << clocks[i].get_location() << " time is "
             << clocks[i].get_hours() << ":"
             << setw(2) << setfill('0')
             << clocks[i].get_minutes()
             << setfill(' ') << "\n";
}
```

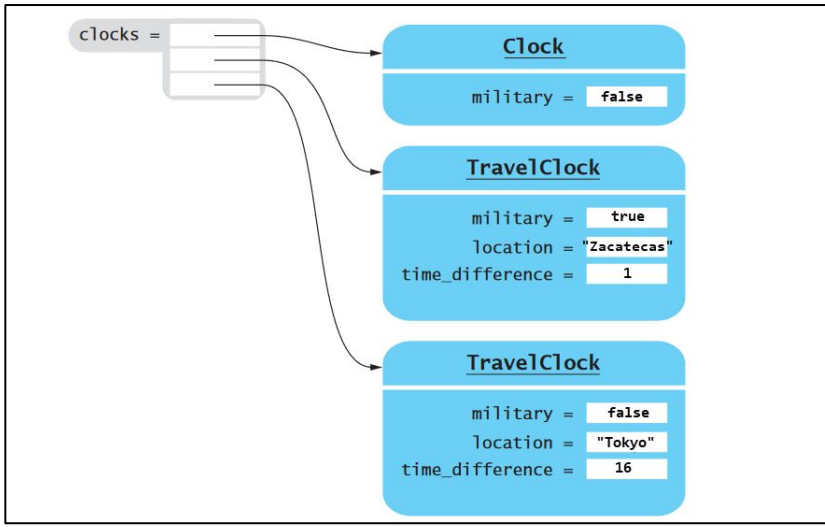


```
/*
 * Remove repetitive code
 * Create a vector of pointers to to Clock objects
 * Pointers will all have the same size, but will point
 * to either a Clock or Travelclock object
 */
vector <Clock*> clocks(3);
//create clock objects and add to vector
//objects will be in heap memory
clocks[0] = new Clock(false);
clocks[1] = new TravelClock(true, "Zacatecas", 2);
clocks[2] = new TravelClock(false, "Tokyo", 17);

//use each object in the vector to invoke functions
//to display info
for(int i = 0; i < clocks.size(); i++){
        cout << clocks[i]->get_location() << " time is "
             << clocks[i]->get_hours() << ":"
             << setw(2) << setfill('0')
             << clocks[i]->get_minutes()
             << setfill(' ') << "\n";
}
cout << "\n";
//deallocate the objects in heap memory
//cannot just do 'delete clocks', since clocks is a vector
//stored in stack memory
for(int i = 0; i < clocks.size();i++)
        delete clocks[i];
```

# First example of polymorphism: a vector that stores different “forms” of a clock

- The pointers will all have the same size (size of a memory address), even though the objects they are pointing to will vary in size.
- We can assign a pointer of type Clock* to point to TravelClock*, but we can’t have a TravelClock* point to a Clock*

```
/*
 * Remove repetitive code
 * Create a vector of pointers to to Clock objects
 * Pointers will all have the same size, but will point
 * to either a Clock or Travelclock object
 */
vector <Clock*> clocks(3);
//create clock objects and add to vector
//objects will be in heap memory
clocks[0] = new Clock(false);
clocks[1] = new TravelClock(true, "Zacatecas", 2);
clocks[2] = new TravelClock(false, "Tokyo", 17);

//use each object in the vector to invoke functions
//to display info
for(int i = 0; i < clocks.size(); i++){
        cout << clocks[i]->get_location() << " time is "
             << clocks[i]->get_hours() << ":"
             << setw(2) << setfill('0')
             << clocks[i]->get_minutes()
             << setfill(' ') << "\n";
}
cout << "\n";
//deallocate the objects in heap memory
//cannot just do 'delete clocks', since clocks is a vector
//stored in stack memory
for(int i = 0; i < clocks.size();i++)
        delete clocks[i];
```




(The time_difference above should be 2 and 17 for Zacatecas and Tokyo, respectively.)

# What happens if we run this?

```
/*
 * Remove repetitive code
 * Create a vector of pointers to to Clock objects
 * Pointers will all have the same size, but will point
 * to either a Clock or Travelclock object
 */
vector <Clock*> clocks(3);
//create clock objects and add to vector
//objects will be in heap memory
clocks[0] = new Clock(false);
clocks[1] = new TravelClock(true, "Zacatecas", 2);
clocks[2] = new TravelClock(false, "Tokyo", 17);

//use each object in the vector to invoke functions
//to display info
for(int i = 0; i < clocks.size(); i++){
        cout << clocks[i]->get_location() << " time is "
             << clocks[i]->get_hours() << ":"
             << setw(2) << setfill('0')
             << clocks[i]->get_minutes()
             << setfill(' ') << "\n";
}
cout << "\n";
//deallocate the objects in heap memory
//cannot just do 'delete clocks', since clocks is a vector
//stored in stack memory
for(int i = 0; i < clocks.size();i++)
        delete clocks[i];
```

Output:

```
Local: San Bernardino, CA time is 3:30
Local: San Bernardino, CA time is 15:30
Local: San Bernardino, CA time is 3:30

Destroying Clock.
Destroying Clock.
Destroying Clock.
```

- Unfortunately, when we run the program we will get some unexpected output.
- The get_location() and get_hours() functions for the TravelClock objects were not called.
- Since the compiler sees the objects as pointers of type Clock*, it makes a note to use Clock member functions for those function calls (so it does what it thinks it's supposed to do).
- Notice that the value of the "is_military" data member is set properly, but nothing else.
- What we want is for our program to be able to first check the object type before calling the function, and this needs to be done during run-time….how do we do this?

# `virtual` Functions

- Virtual functions are base class member functions whose behavior can be overridden in derived classes.
- Virtual functions allow for overriding behavior even if there is no compile-time information about the type of object invoking a function (such as with pointers that point to objects in heap memory).
- Using the `virtual` keyword in the base class will automatically make all functions in the derived class with the same name and parameters types virtual functions as well.
- Whenever a `virtual` function is called, the exact function that is going to be called will be determined at run-time.
    - This is referred to as *Dynamic Binding*, whereas *Static Binding* occurs for function calls determined at compilation time.

Base class with virtual **destructor** and **member functions**

```
/*
 * Filename: clock.h
 * Definition of the Clock class.
 * Clock: base class that can tell the current local time
 * In the constructor, you can set the format to either
 * "military format"  or "am/pm"
 */
#ifndef CLOCK_H
#define CLOCK_H

#include <string>
#include <iostream>
using namespace std;

class Clock{
        private:
                bool military;//determines format of time
        public:
                Clock(bool use_military);
                virtual ~Clock();
                virtual string get_location() const;
                virtual int get_hours() const;
                int get_minutes() const;
                bool is_military() const;
};
#endif
```

Derived class with virtual member functions (not noted but will inherit from base class)

```
/*
 *Filename: travelclock.h
 * TravelClock Class Definition
 *
 * Constructs a travel clock that can tell the time at a specified location.
 *      mil=true if the clock uses military format
 *      loc= the location
 *      diff=the time difference from the local time
 */
#include <string>

using namespace std;

#include "clock.h"

#ifndef TRAVELCLOCK_H
#define TRAVELCLOCK_H
class TravelClock : public Clock//inherits military data member
{
        private:
                string location;
                int time_difference;
        public:
                TravelClock(bool mil, string loc, int diff);
                ~TravelClock();
                string get_location() const;
                int get_hours() const;
};
#endif
```

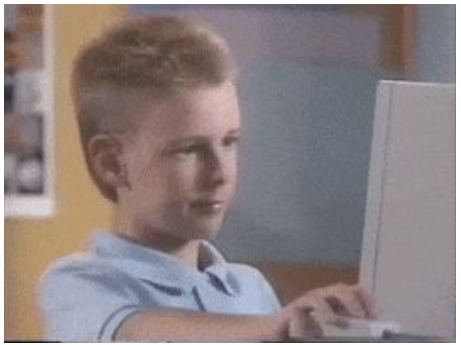
- When the compiler encounters a call to “get_location” or “get_hours”, it’s going to skip the binding and it will allow the object type to be determined during run-time.
- Similarly when an object is deleted, it will check what type of object is being deleted to call the correct destructor.

# After properly implementing polymorphism...we are all good now!

```
/*
 * Remove repetitive code
 * Create a vector of pointers to to Clock objects
 * Pointers will all have the same size, but will point
 * to either a Clock or Travelclock object
 */
vector <Clock*> clocks(3);
//create clock objects and add to vector
//objects will be in heap memory
clocks[0] = new Clock(false);
clocks[1] = new TravelClock(true, "Zacatecas", 2);
clocks[2] = new TravelClock(false, "Tokyo", 17);

//use each object in the vector to invoke functions
//to display info
for(int i = 0; i < clocks.size(); i++){
        cout << clocks[i]->get_location() << " time is "
             << clocks[i]->get_hours() << ":"
             << setw(2) << setfill('0')
             << clocks[i]->get_minutes()
             << setfill(' ') << "\n";
}
cout << "\n";
//deallocate the objects in heap memory
//cannot just do 'delete clocks', since clocks is a vector
//stored in stack memory
for(int i = 0; i < clocks.size();i++)
        delete clocks[i];
```

Output:

```
Local: San Bernardino, CA time is 3:55
Zacatecas time is 17:55
Tokyo time is 8:55

Destroying Clock.
Destroying Travel Clock.
Destroying Clock.
Destroying Travel Clock.
Destroying Clock.
```



# Inheritance and Polymorphism Review

- As we have seen in this example program, we can represent polymorphic collections of different object types.
- **Inheritance** is used to express the commonality between objects.
- **Polymorphism** , such as what is implemented with virtual functions and vectors of pointers, gives our programs a great deal of flexibility and extensibility.
- We can easily extend the number of derived classes of a base class and make sure that the appropriate functions are called each time.