Polymorphism

Polymorphism literally means "many shapes". Inheritance allows you to define a base class and derive classes from the base class. Polymorphism allows you to make changes in the method definition for the derived classes and have those changes apply to the methods written in the base class. In C++ you must specify a method as "virtual" if you want this to happen.

Consider the `Transaction` and `CreditCardTransaction` classes that we wrote in the last lecture. In this case, CreditCardTransaction is derived from Transaction.

```
void Transaction::print()
{
  cout << "Name: " << name << " Amount: " << amount << endl;
}

void CreditCardTransaction::print()
{
        Transaction::print(); // Call parent print function
        cout << "Number: " << number << endl;  // Add number
}
```

If we create a Transaction object then we get what you might expect:

```
Transaction t("Bilbo", 199.59);
t.print();        //  Outputs "Name: Bilbo Amount: 199.59
```

If we create a CreditCardTransaction object then we also get what you might expect:

```
CreditCardTransaction cct("Frodo", "1111 0000", 10.25);
cct.print();      //  Outputs Name: Frodo Amount: 10.25
                  //          Number: 1111 0000
```

But what if we create a CreditCardTransaction, copy (or typecast) it to a Transaction, and call print? You might wonder why you would want to do this, but such a circumstance is actually quite common. For example:

```
CreditCardTransaction cct("Frodo", "1111 0000", 10.25);
Transaction *t = &cct;
t->print();
```

First, why is this legal? It is because a CreditCardTransaction "Is A" Transaction. This makes it legal to assign the more specific object to the more general one. But we can't go the other way – we can't assigned a Transaction object to a CreditCardTransaction, because the Transaction might be something like a CashTransaction and it's not valid to make it a CreditCardTransaction.

So what is the output? We have two possible choices since there are two "print" functions. Since we originally created cct as a CreditCardTransaction it makes some

sense to use CreditCardTransaction's print function. On the other hand, since we are invoking it as a Transaction object, it also makes sense to use Transaction's print function.

The answer is:

```
Frodo Amount: 10.25
```

In other words, it uses the print function defined for Transaction. This might be surprising to you if you learned Java first, because in Java you'd get the credit card number. So the bottom line, is **when we redefine a function in C++, the function that is invoked is determined by the type of the object that does the invoking**. We start at the class based on the type of the calling object, if the function is defined there then we use it, otherwise C++ searches up the inheritance hierarchy and invokes the first function hat is found that matches the name (i.e. the inheritance part).

Unfortunately, the behavior we just saw is not what we normally want. **Usually we want to call the function that was defined for the class of the object when it was originally created**!

Here is an example. Let's say that we have an array of transactions that represent all of the transactions that happened in one day. There could be many different kinds of transactions, some CreditCardTransactions, some CashTransactions, some CheckTransactions, etc.

```
Transaction* trans[4];            // Array to pointer of Transaction

trans[0] = new CreditCardTransaction("Frodo", "0000 0001", 10.50);
trans[1] = new CreditCardTransaction("Bilbo", "0000 0000", 99.23);
trans[2] = new CashTransaction("Sauron", 10000.50);
trans[3] = new CashTransaction("Gandalf", 9510.21);

// Output each transaction
for (int i = 0; i < 4; i++)
{
      trans[i]->print();
      cout << endl;
}

// Free memory
for (int i = 0; i < 4; i++)
      delete trans[i];
```

The problem is this program will only use Transaction::print() and we miss out on the credit card details that we could be printing for Frodo and Bilbo. How to get around this? The solution is to use what are called **virtual functions**. (This is the default behavior in Java, so you might not be aware that it has a name, since it needs-not-be-named when it always happens in Java).

When we declare a function to be virtual then C++ remembers the type of the object when it was created. **If we call a virtual function that exists at multiple levels in the inheritance hierarchy, then C++ uses the function associated with the class of the object when the object was created**.

To make a function virtual, we just put the keyword "virtual" in front of the function in the class definition:

```
class Transaction
{
public:
	Transaction(void);
	~Transaction(void);
	Transaction(string newName, double newAmount);
	string getName();
	double getAmount();
	virtual void print();
private:
	double amount;
	private:
	string name;
};
```

If we run the same program then now it will output:

```
Name: Frodo Amount: 10.5
Number: 0000 0001

Name: Bilbo Amount: 99.23
Number: 0000 0000

Name: Sauron Amount: 10000.5

Name: Gandalf Amount: 9510.21
```

Other names for this behavior are: **polymorphism**, **dynamic binding**, and **late binding**. We also say that the function / method "print" is **overridden** in the child class. Lots of terminology!

At first glance, this might seem like an esoteric feature of C++. However, it's really extremely powerful. One of the amazing things about polymorphism is it lets us invoke functions that might not even exist yet! For example, we could write our code to handle CreditCardTransactions and CashTransactions. One day we decide to handle PayPalTransactions or BitcoinTransactions. We don't have to change any of our existing code! We just write the new classes so they override the "print" function, and if we add one of these classes to our array then the details will automagically get printed out. We wouldn't even need to recompile the Transaction class. This makes our program really extensible for future functionality.

Pure virtual functions and abstract classes

Sometimes it doesn't make sense to define a function at the base class. The prototypical example is to have a "Shape" class and derived classes like "Circle" or "Square". Consider the following:

```
class Shape {
  public:
    virtual int area();
};

class Square : public Shape {
  public:
    int area();
    int side;
};
int Square::area() { return side*side; }
```

In this case we might want to define area() as a virtual function so we can invoke the function for anything derived from Shape. But the function is going to do totally different things for different kinds of shapes, so it's impossible to define at the Shape class. It must be defined at a derived class. When this situation arises, you can make a **pure virtual function**. A pure virtual function has no implementation. When you do this in a class then the class becomes an **abstract class**. Since no implementation exists for a function, you can't make an instance of it.

Here is the Shape class turned into an abstract class with a pure virtual function. Just add =0 to the end of the function definition.

```
class Shape {
  public:
    virtual int area() = 0;
};
```

One of the side-effects is that any class derived from Shape must implement the area function. The compiler will give an error message if the function is missing. For this reason, abstract classes are sometimes referred to as **interfaces**. A class has to implement all of the functions defined in the interface.

Polymorphism Example – Guessing Game

Consider a guessing game where someone is thinking of a number from 0-99 and two other players try to guess the number. The players are told if the guess is too high or too low if they are not correct.

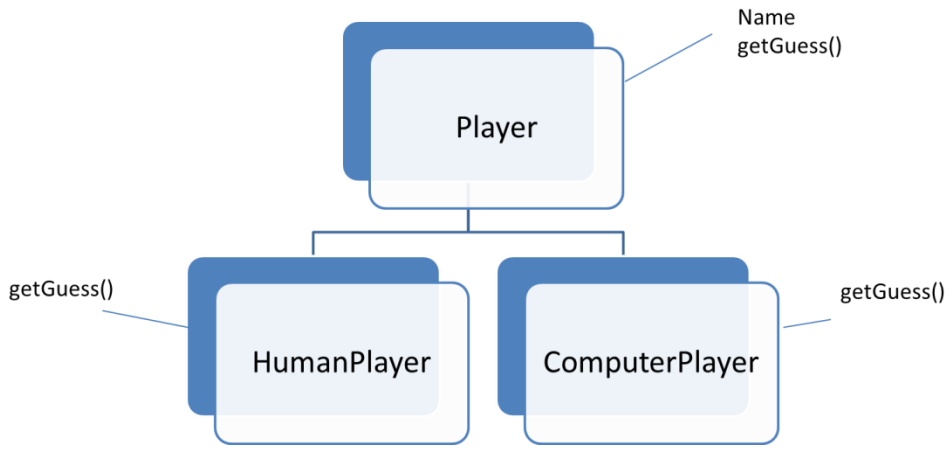
Here is the main game code with most of the logic in the `playRound` method:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "Player.h"
#include "ComputerPlayer.h"
#include "HumanPlayer.h"
using namespace std;

// Have to pass p1 and p2 by reference to use virtual function
static bool playRound(Player &p1, Player &p2, int number)
{
        // Get P1's guess
        cout << "Player 1: " << p1.getName() << ", it is your turn. ";
        int p1Guess = p1.getGuess();
        cout << p1.getName() << " guessed " << p1Guess << endl;
        if (p1Guess == number)
        {
                cout << "That is the correct number!" << endl;
                return false;
        }
        else if (p1Guess < number)
                cout << "The guess is too low." << endl;
        else
                cout << "The guess is too high." << endl;

        // Get P2's guess
        cout << "Player 2: " <<  p2.getName()
                << ", it is your turn. " << endl;
        int p2Guess = p2.getGuess();
        cout << p2.getName() << " guessed " << p2Guess << endl;
        cout << "Player 2, " << p2.getName() <<
                        ", guessed " << p2Guess << endl;
        if (p2Guess == number)
        {
                cout << "That is the correct number!" << endl;
                return false;
        }
        else if (p2Guess < number)
                cout << "The guess is too low." << endl;
        else
                 cout << "The guess is too high." << endl;
        return true;
}

int main()
{
        srand(time(NULL));
```

```
        HumanPlayer p1("Kenrick");
        ComputerPlayer p2("Tobor");
        int numToGuess = rand() % 100;

        while (playRound(p1,p2,numToGuess))
        {
                cout << endl;
                cout << "Starting next round.";
        }
}
```

The key part about this method is it takes two arbitrary `Player` objects and then invokes the `getGuess()` method to get the player's guess.

We can implement the `getGuess()` method different ways depending upon the player. The nice thing about polymorphism is that C++ automatically uses the method defined for the Player object passed in. In our case we'll define a `Player` class and a `HumanPlayer` and a `ComputerPlayer` derived from it.



FILE: Player.h

```
#ifndef _PLAYER_
#define _PLAYER_
#include <string>
using namespace std;

class Player
{
        private:
                string name;
        public:
                Player() : name("Unknown") {};
                Player(string n) : name(n) {};
                string getName() { return name; };
                virtual int getGuess() = 0;
};
#endif
```

FILE: ComputerPlayer.h

```
#include "Player.h"
#include <iostream>
using namespace std;

class ComputerPlayer : public Player
{
 public:
        ComputerPlayer() : Player() {};
        ComputerPlayer(string n) : Player(n) {};
        int getGuess();
};
```

FILE: ComputerPlayer.cpp

```
#include "ComputerPlayer.h"
#include <cstdlib>
using namespace std;

int ComputerPlayer::getGuess()
{
        int guess = rand() % 100;
        return guess;
}
```

FILE: HumanPlayer.h

```
#include "Player.h"
#include <iostream>
using namespace std;

class HumanPlayer : public Player
{
 public:
        HumanPlayer() : Player() {};
        HumanPlayer(string n) : Player(n) {};
        int getGuess();
```

```
};
```

FILE: HumanPlayer.cpp

```
#include "HumanPlayer.h"
#include <iostream>
using namespace std;

int HumanPlayer::getGuess()
{
        cout << "Enter your guess. " << endl;
        int guess;
        cin >> guess;  // could do input validation here
        return guess;
}
```

We've made a really dumb implementation for `getGuess()` for the computer. It just guesses a random number. The human player just inputs a guess from the keyboard.

When this program is run it will alternate between inputting a guess from the human from the keyboard (for player 1) and randomly guessing a number for player 2. If we wanted to make two computer players we simply change p1 to an instance of `ComputerPlayer` and re-run the program. We could make a smarter AI program as well and plug it right in (although it would need additional methods to tell the AI if the guess was too high or too low).