Polymorphism (abstract classes, default parameters, user-defined types)

Abstract classes:

```
computer$ g++ -std=c++14 practice.cpp
computer$ ./a.out
20
10
10
10
This is in the function in the Shape class: 20
This is in the function in the Shape class: 10
```

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

//the presence of a pure virtual function makes this an abstract class.
//This class is only to be used as a base class, you cannot make objects from this class.  A class that we can make objects from would be a called concrete class
class Shape {
  protected:
    int w;
    int h;
  public:
    virtual ~Shape(){} //add this for deleting any derived pointer
    void make_shape (int a, int b)
    {
       w=a;
       h=b;
    }

    virtual int area () =0; //pure virtual function (indicated by =0).  There is no implementation to this function-derived classes provide an implementation (in some cases you can provide a body).  Also note it should be a virtual function (don't do int area()=0)

    void print_area()
    {
      cout << "This is in the function in the Shape class: "<<this->area() << endl;
    }
};

class Rectangle: public Shape {
  public:
    //in this class, we provide a definition to the function
    int area ()
    {
       return (w * h);
    }
};
```

```
class Triangle: public Shape {
  public:

    int area ()
    {
      return (w * h / 2);
    }

};

int main (int argc, char **argv) {
  //Shape sh;    we can't do this (make an object of the abstract class)
  Rectangle rect;
  Shape * sh1 = &rect; //but we can do this (make a pointer of the abstract class type)
  shared_ptr<Shape> sh2=std::make_shared<Triangle>(); //and this
  Shape *sh3=new Triangle();   //and this

  shared_ptr<Triangle> sh4 = std::make_shared<Triangle>();


  sh1->make_shape (4,5);
  sh2->make_shape (4,5);
  sh3->make_shape (4,5);
  sh4->make_shape (4,5);

  cout << sh1->area() << endl;
  cout << sh2->area() << endl;
  cout << sh3->area() << endl;
  cout << sh4->area() << endl;

  sh1->print_area();
  sh3->print_area();

  delete(sh3); //using destructor
  return 0;
}
```

Abstract class example:

```
computer$ g++ -std=c++14 people.cpp
computer$ ./a.out
Bonjour!
Ciao!
Francois!  Ciao bello!
Ciao!  Arrivederci!
```

```
#include <iostream>
#include <vector>
```

```
using namespace std;

class Person{  //this is an abstract class

	string name;
	char gender;

public:

	Person(string name, char gender)
	{
		this->name=name;
		this->gender=gender;
	}

	virtual void greet()=0;  //override this function in derived classes


	virtual void say_bye()=0;  //override this function in derived classes


	char get_gender()
	{
		return gender;
	}

	string get_name()
	{
		return name;
	}

};

class French_person:public Person{

public:
	using Person::Person; //using the Person constructor

	void greet()
	{
		cout<<"Bonjour!"<<endl;
	}

	void say_bye()
	{
		cout<<"Au revoir!"<<endl;
	}

};
```

```
class German_person:public Person{

public:
	using Person::Person; //using the Person constructor
	void greet()
	{
		cout<<"Guten tag!"<<endl;
	}

	void say_bye()
	{
		cout<<"Auf weidersehen!"<<endl;
	}

};

class Arab_person:public Person{

public:
	using Person::Person; //using the Person constructor
	void greet()
	{
		cout<<"Marhaba!"<<endl;
	}

	void say_bye()
	{
		cout<<"Ma'a salama!"<<endl;
	}

};

class Filipino_person:public Person{

public:
	using Person::Person; //using the Person constructor
	void greet()
	{
		cout<<"Mabuhay!"<<endl;
	}

	void say_bye()
	{
		cout<<"Bye!"<<endl;
	}


};
```

```
class Italian_person:public Person{

public:
	using Person::Person; //using the Person constructor
	void greet()
	{
		cout<<"Ciao!"<<endl;
	}

	//overloading the overriden function
	void say_bye()
	{
		cout<<"Ciao!  Arrivederci!"<<endl;
	}

	void say_bye(Person *p) //check person's gender-notice we are passing a base pointer parameter but we will still be able to handle derived classes.
	{
		if(p->get_gender()=='f')
		{
			cout<<p->get_name()<<"!  Ciao bella!"<<endl;
		}

		else
		{
			cout<<p->get_name()<<"!  Ciao bello!"<<endl;
		}
	}
};


class Party{

	vector<Person*> party_people; //vector of base pointers-we can store derived pointers also

public:

	~Party()  //clear vector of pointers
	{
		for (int i=0;i<party_people.size();i++)
		{
			delete(party_people.at(i));
		}
	}

	//join party
	void join_party(Person *p1) //using base pointer parameter, but can pass in derived pointers
	{
```

```
                party_people.push_back(p1);
        }

};


int main(int argc, char **argv)
{
        Party p1;
        Person* f1=new French_person("Francois", 'm');
        Italian_person* i1=new Italian_person("Pietro", 'm');

        f1->greet(); //prints the correct greeting even though we used a Person pointer
        p1.join_party(i1);

        Italian_person i2("Francesca", 'f');
        French_person f2("Pierre", 'm');

        i2.greet();
        i2.say_bye(f1); //using the overloaded say_bye function
        i2.say_bye();
}
```

Default parameters:

```
computer$ g++ -std=c++14 practice.cpp
Computers-MacBook-Air:C++ computer$ ./a.out
Strings on this guitar: 7
Strings on this guitar: 5
guitar:do re mi
guitar:fa so so
guitar:fa so so

Playing instrument!!!
~~~~
Done playing!

2 people will play...A B C minor
just a random function...
```

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

//abstract class with pure virtual function
class Instrument{

protected: //derived classes can access
  string name;
```

```
  virtual void play(int num, string notes="do re mi")=0; //using default parameter-note that you can do this other functions, not just pure virtual functions

public:
  //overload operator <<
  void operator << (string s)
  {
    cout<<"\nPlaying instrument!!!"<<endl;
    cout<<s<<endl;
    cout<<"Done playing!\n"<<endl;
  }

};

class Guitar:public Instrument{
  int strings;

public:
  Guitar() //create a constructor specific to the derived class
  {
    name="guitar";
    strings=6;
  }

  void see_strings()
  {
    cout<<"Strings on this guitar: "<<strings<<endl;
  }

  void play(int num, string notes="do re mi") //using default parameter
  {
    for (int i=0;i<num; i++)
    {
       cout<<name<<":"<<notes<<endl;
    }
  }

  //overloading <<, take a string from second, give to first one
  void operator << (Guitar &g)
  {
    this->strings++;
    g.strings--;

    this->see_strings();
    g.see_strings();
  }

  //DON'T DO THIS FUNCTION:
  // void play(int num)
  // {
```

```
  //   cout<<"example..."<<endl;
  // }
  //^^^It will be ambiguous because now we can call: play(3) and we do not know if we are talking about the
//inherited pure virutal function (since it can also be called as play(3) using the default parameter)

};

class Piano:public Instrument{

public:
  Piano() //create a constructor specific to the derived class
  {
    name="piano";
  }

  void play(int num, string notes="A B C minor") //using default parameter (still a string)
  {
    if(num==1)
    {
      cout<<"1 person will play..."<<notes<<endl;
    }

    else
    {
      cout<<num<<" people will play..."<<notes<<endl;
    }
  }

  //overloading-same name as something in base, including PVF (pure virtual function)
  void play() //overloading the play() function
  {
    cout<<"just a random function..."<<endl;
  }

};


int main (int argc, char **argv) {
  Guitar g;
  Guitar g1;

  g<<g1; //notice that this is using the the overloaded << defined in the Guitar class (if you defined an
overloaded operator in a derived class that is also in the base class, then that is what is available to you)

  g.play(1); //will use default parameter-since we didn't give a second argument, the default (defined above as:
do re mi) will be used
  g1.play(2, "fa so so"); //will use what was passed in (still same function)

  Piano p;
  Piano p1;
```

```
 p1<<"~~~~"; //use overloaded operator defined in the base class (Instrument) since I didn't define anything new in the Piano class

 p.play(2);
 p.play(); //using the overloaded function

}
```

Types:

- We have built-in types like *int* or *char*
  - When we declare a variable, we can say that the type is *int* or *char*
  - By letting the computer know the type, we let it know how much space to set aside for the variable
  - For example, when we say *int s=3;* the computer knows to set aside 4 bytes (assuming an integer type with 4 bytes)

- We can define our own types
  - People can declare variables using the types we declare
  - Just like we don't care about behind the scenes when using a type like *int* or *char*, when people use our types, they don't care about how we defined it
    - They just use the types

```
computer$ g++ -std=c++11 practice.cpp
computer$ ./a.out
3...4...1...55
```

```
#include <iostream>
#include <string>

//defining a type outside of a class
struct Tree{
 int year;
 std::string type;
};

//defining types inside of a class
class Making_types{

public: //making all the types we are defining public

 typedef int distance; //instead of something being declared as an int, we can give another name that makes more sense in whatever program we are making (it is still stored as an int)
 typedef int* width;

 using currency=float;
```

```
 struct Person{
   int age;
   std::string name;
 };

 class Cat{  //You can declare a class within a class

 public:
 bool likes_catnip;
 std::string name;

 };

   enum Cars{Car1, Car2}; //we will talk about enums in a future lecture

};


int main (int argc, char **argv)
{
 Tree t1; //we declared this struct outside of the class, so we can access it like this

 //Cat c1; Won't work because the type is defined in a class
 Making_types::Cat c1; //declaring c1 as a Cat type (note the :: indicating the type is in the Making_types class). just like when we say int s; (where int is the type of s)
  Making_types::Person p;
 p.age=33;
 c1.likes_catnip=true; //we can now just use the name c1 alone (just like if we declared int s, after that we can just say s)

  Making_types::Person* p_ptr=&p;
 p_ptr->age=55;

 Making_types::Cars ca1;

 ca1=Making_types::Car2; //Car2 is the second in the enum above, so the value is 1 (Car1 is 0)

 int s=3;
 Making_types::width w1=&s;
 Making_types::distance d1=4;

 std::cout<<*w1<<"..."<<d1<<"..."<<ca1<<"..."<<p.age<<std::endl;

}
```