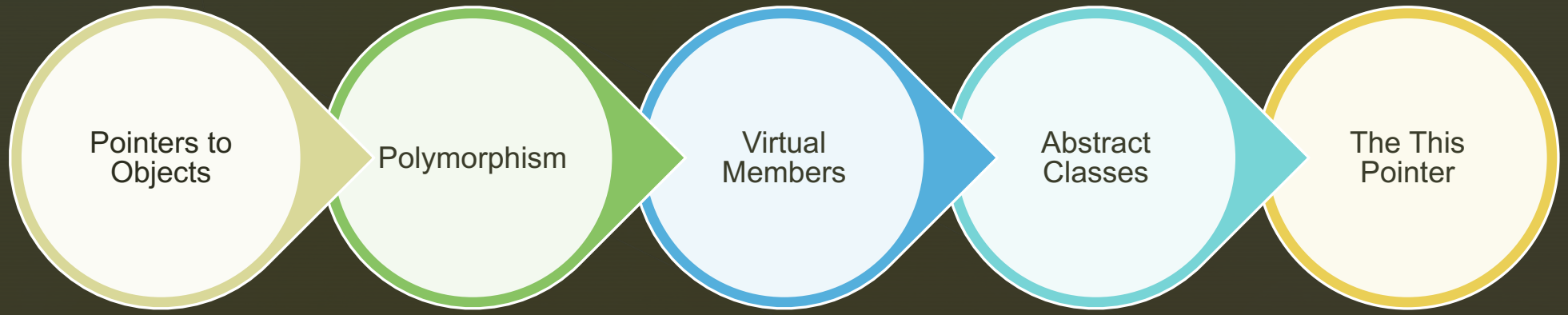
DR. (MRS) T-S.M.A. ADJAIDOO

09

Polymorphism

Today's Lesson



Learning Outcomes

1

- Learners should appreciate the need for polymorphism

2

- Learners should be able to crate polymorphic classes

Polymorphism

Pointers to Objects

```
1   #include <iostream>
2   using namespace std;
3
4   class person      //class of persons
5   {
6       protected:
7           char name[40];  //person's name
8       public:
9           void setName()  //set the name
10          {
11              cout << "Enter name: ";
12              cin >> name;
13          }
14          void printName()      //get the name
15          {
16              cout << "\n Name is: " << name;
17          }
18  };
```

Pointers to Objects

```
int main()
{
    person* persPtr[100];       //array of pointers to persons
    int n = 0;      //number of persons in array
    char choice;
    do //put persons in array
    {
        persPtr[n] = new person;     //make new object
        persPtr[n]->setName();  //set person's name
        n++;    //count new person
        cout << "Enter another (y/n)? ";     //enter another
        cin >> choice;   //person?
    }
    while( choice=='y' );    //quit on 'n'

    for(int j=0; j<n; j++)   //print names of
    {           //all persons
        cout << "\nPerson number " << j+1;
        persPtr[j]->printName();
    }
    cout << endl;
    return 0;
} //end main()
```

Pointers to Objects

- The main() function defines an array, persPtr, of 100 pointers to type person.
- In a do loop it then asks the user to enter a name.
- With this name it creates a person object using new, and stores a pointer to this object in the array persPtr.
- To demonstrate how easy it is to access the objects using the pointers, it then prints out the name data for each person object

Pointers to Objects

- Each of the elements of the array persPtr is specified in array notation to be persPtr[j] (or equivalently by pointer notation to be *(persPtr+j)).
- The elements are pointers to objects of type person.
- To access a member of an object using a pointer, we use the -> operator.
- Putting this all together, we have the following syntax for getname(): persPtr[j]->getName()
- This executes the getname() function in the person object pointed to by element j of the

  persPtr array.

Pointers to Objects

- Suppose you have a number of objects of different classes but you want to put them all in an array and perform a particular operation on them using the same function call.
- For example, suppose a graphics program includes several different shapes: a triangle, a ball, a square, and so on. Each of these classes has a member function draw() that causes the object to be drawn on the screen.
- Now suppose you plan to make a picture by grouping a number of these elements together, and you want to draw the picture in a convenient way.
- What do you do?

Pointers to Objects

- One approach is to create an array that holds pointers to all the different objects in the picture. The array might be defined like this:

```
shape* ptrarr[100]; // array of 100 pointers to shapes
```

- If you insert pointers to all the shapes into this array, you can then draw an entire picture using a simple loop:

```
for ( int j=0; j<N; j++ )

    ptrarr[ j ] -> draw( );
```

Pointers to Objects

- This is an amazing capability: Completely different functions are executed by the same function call.
- If the pointer in ptrarr points to a ball, the function that draws a ball is called; if it points to a triangle, the triangle-drawing function is called.
- This is called *polymorphism,* which means different forms.
- The functions have the same appearance, the draw() expression, but different actual functions are called, depending on the contents of ptrarr[j].

Polymorphism

- Polymorphism is one of the key features of object-oriented programming, after classes and inheritance.
- For the polymorphic approach to work, several conditions must be met.
  - First, all the different classes of shapes, such as balls and triangles, must be descended from a single base class.
  - Second, the draw() function must be declared to be virtual in the base class.

Polymorphism

The example that follows shows what happens when a base class and derived classes all have functions with the same name, and you access these functions using pointers but without using virtual functions.

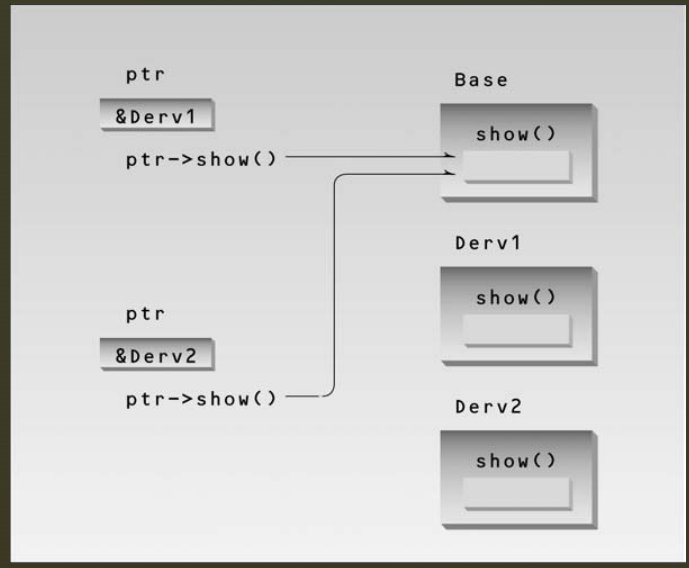
```
#include <iostream>
using namespace std;
class Base //base class
{
    public:
        void show() //normal function
        { cout << "Base\n"; }
};

class Derv1 : public Base //derived class 1
{
    public:
        void show()
        { cout << "Derv1\n"; }
};

class Derv2 : public Base //derived class 2
{
    public:
        void show()
        { cout << "Derv2\n"; }
};
```

Polymorphism

```
int main()
{
    Derv1 dv1; //object of derived class 1
    Derv2 dv2; //object of derived class 2
    Base* ptr; //pointer to base class
    ptr = &dv1; //put address of dv1 in pointer
    ptr->show(); //execute show()
    ptr = &dv2; //put address of dv2 in pointer
    ptr->show(); //execute show()
    return 0;
}
```



Polymorphism

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

class CPolygon {
    protected:
        int width, height;
    public:
        void set_values (int a, int b) {
            width=a; height=b; }
};

class CRectangle: public CPolygon {
    public: int area () {
        return (width * height);
        }
};

class CTriangle: public CPolygon {
    public: int area () {
        return (width * height / 2);
        }
};
```

Polymorphism

```
int main () {
    CRectangle rect;
    CTriangle trgl;
    CPolygon * ppoly1 = &rect;
    CPolygon * ppoly2 = &trgl;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    cout << rect.area() << endl;
    cout << trgl.area() << endl;
return 0;
}
```

Polymorphism

- In function main, we create two pointers that point to objects of class CPolygon (ppoly1 and ppoly2).
- Then we assign references to rect and trgl to these pointers, and because both are objects of classes derived from CPolygon, both are valid assignment operations.
- The only limitation in using *ppoly1 and *ppoly2 instead of rect and trgl is that both *ppoly1 and *ppoly2 are of type CPolygon* and therefore we can only use these pointers to refer to the members that CRectangle and CTriangle inherit from CPolygon.
- For that reason when we call the area() members at the end of the program we have had to use directly the objects rect and trgl instead of the pointers *ppoly1 and *ppoly2.

Polymorphism

- In order to use area() with the pointers to class CPolygon, this member should also have been declared in the class CPolygon, and not only in its derived classes, but the problem is that CRectangle and CTriangle implement different versions of area, therefore we cannot implement it in the base class.
- This is when virtual members become handy

Virtual members

A member of a class that can be redefined in its derived classes is known as a virtual member.

In order to declare a member of a class as virtual, we must precede its declaration with the keyword virtual.

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

class CPolygon {
    protected:
        int width, height;
    public:
        void set_values (int a, int b) {
            width=a; height=b; }
        virtual int area () {
            return (0); }
};
```

Virtual members

```
class CRectangle: public CPolygon {
    public:
        int area () {
            return (width * height);
        }
};

class CTriangle: public CPolygon {
        public:
            int area () {
                return (width * height / 2);
            }
 };
```

Virtual members

```
int main () {
    CRectangle rect;
    CTriangle trgl;
    CPolygon poly;
    CPolygon * ppoly1 = &rect;
    CPolygon * ppoly2 = &trgl;
    CPolygon * ppoly3 = &poly;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    ppoly3->set_values (4,5);
    cout << ppoly1->area() << endl;
     cout << ppoly2->area() << endl;
    cout << ppoly3->area() << endl;
return 0; }
```

Virtual members

- Now the three classes (CPolygon, CRectangle and CTriangle) have all the same members:
  - width, height, set_values() and area().
- The member function area() has been declared as virtual in the base class because it is later redefined in each derived class.
- A class that declares or inherits a virtual function is called a *polymorphic class*.

Abstract Base Classes

- When we will never want to instantiate objects of a base class, we call it an *abstract* class.
- Such a class exists only to act as a parent of derived classes that will be used to instantiate objects.
- It may also provide an interface for the class hierarchy.
- A pure virtual function is one with the expression = 0 added to the declaration

Abstract Base Classes

- The main difference between an abstract base class and a regular polymorphic class is that because in abstract base classes at least one of its members lacks implementation we cannot create instances (objects) of it.

Abstract Base Classes

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

class CPolygon {
    protected:
        int width, height;
    public:
        void set_values (int a, int b) {
            width=a; height=b;
        }
        virtual int area (void) = 0;
};

class CRectangle: public CPolygon {
    public:
        int area (void) {
            return (width * height);
        }
};
```

Abstract Base Classes

```
class CTriangle: public CPolygon {
    public:
        int area (void) {
        return (width * height / 2);
        }
};

int main () {
    CRectangle rect;
    CTriangle trgl;
    CPolygon * ppoly1 = &rect;
    CPolygon * ppoly2 = &trgl;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    cout << ppoly1->area() << endl;
    cout << ppoly2->area() << endl;
return 0;
}
```

Abstract Base Classes

- If you review the program you will notice that we refer to objects of different but related classes using a unique type of pointer (CPolygon*). This can be tremendously useful.
- For example, now we can create a function member of the abstract base class CPolygon that is able to print on screen the result of the area() function even though CPolygon itself has no implementation for this function.

Abstract Base Classes

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

class CPolygon {
    protected:
        int width, height;
    public:
        void set_values (int a, int b) {
            width=a; height=b; }
        virtual int area (void) =0;
        void printarea (void) {
            cout << this->area() << endl;
        }
};

class CRectangle: public CPolygon {
    public:
        int area (void) {
            return (width * height);
        }
};
```

Abstract Base Classes

```
class CTriangle: public CPolygon {
        public:
            int area (void) {
                return (width * height / 2);
            }
};

int main () {
    CRectangle rect;
     CTriangle trgl;
    CPolygon * ppoly1 = &rect;
    CPolygon * ppoly2 = &trgl;
    ppoly1->set_values (4,5);
    ppoly2->set_values (4,5);
    ppoly1->printarea();
    ppoly2->printarea();
return 0; }
```

The this Pointer

The member functions of every object have access to a sort of magic pointer named this, which points to the object itself.

Thus any member function can find out the address of the object of which it is a member.

```
#include <iostream>
using namespace std;
class where
{
    private:
        char charray[10]; //occupies 10 bytes
    public:
        void reveal()
            { cout << "\nMy object's address is " << this; }
};

int main()
{
    where w1, w2, w3; //make three objects
    w1.reveal(); //see where they are
    w2.reveal();
    w3.reveal();
    cout << endl;
    return 0;
}
```

The this Pointer

```
#include <iostream>
using namespace std;
class what
{
    private:
        int alpha;
    public:
        void tester()
        {
            this->alpha = 11; //same as alpha = 11;
            cout << this->alpha; //same as cout << alpha;
        }
};

int main()
{
    what w;
    w.tester();
    cout << endl;
return 0;
}
```

Any Questions?

The End

Contact: tsadjaidoo@knust.edu.gh

Office: Caesar Building, Room 413