AAB University

Faculty of Computer Sciences

Object Oriented Programming

Week 11:

Polimorphism

Asst. Prof. Dr. **Mentor Hamiti**
mentor.hamiti@universitetiaab.com

Last Time?!

- Inheritance
- Base & Derived Classes
- Access Control and Type of Inheritance
- Constructor and Inheritance
- Overriding Base Class Functions
- Virtual Base Class

Today

- Introduction
- Base & Derived Classes
- Polymorphism
- Pointers to base class
- Virtual members
- Abstract base classes

Base classes and derived classes

- Inheritance is a fundamental requirement of oriented programming

- It allows us to create new classes by refining existing classes

- Essentially a derived class can inherit data members of a base class
  - The behaviour of the derived class can be refined by redefining base class member functions or adding new member function
  - A key aspect of this is **polymorphism** where a classes behaviour can be adapted at run-time

Base classes and derived classes

- There are many examples in real life of how a (base) class can be refined to a set of (derived) classes

- <u>Example</u>:

  A Polygon class can be refined to be a Quadrilateral which can be further refined to be a Rectangle

  - A Quadrilateral IS-A Polygon
  - A Rectangle IS-A Quadrilateral

Base classes and derived classes

- Example:

Base class	Derived class
Shape	Triangle, Circle, Rectangle
Bank Account	Current, Deposit
Student	Undergraduate, Postgaduate
Vehicle	Car, Truck, Bus
Filter	Low-pass, Band-pass, High-pass

Base classes and derived classes

- Example: A BankAccount class

- An BankAccount base class models basic information about a bank account
  - *Account holder*
  - *Account number*
  - *Current balance*

- Basic functionality
  - *Withdraw money*
  - *Deposit money*

Class member	Can be accessed from
private	public member functions of same class
protected	public member functions of same class and derived classes
public	Anywhere

Object Oriented Programming

- Introduction
- Base & Derived Classes
- Polymorphism

Polymorphism and Object Oriented Programming

- **Polymorphism** is the **key concept** in Object Oriented Programming
- Polymorphism literally means many forms
- Essentially we are able to get <u>many different types of object behaviour</u> from a <u>single reference type</u>
  - This enables us to write easily extensible applications

Polimorphism

- The ability to declare functions/methods as **virtual** is one of the central elements of Polymorphism in **C++**

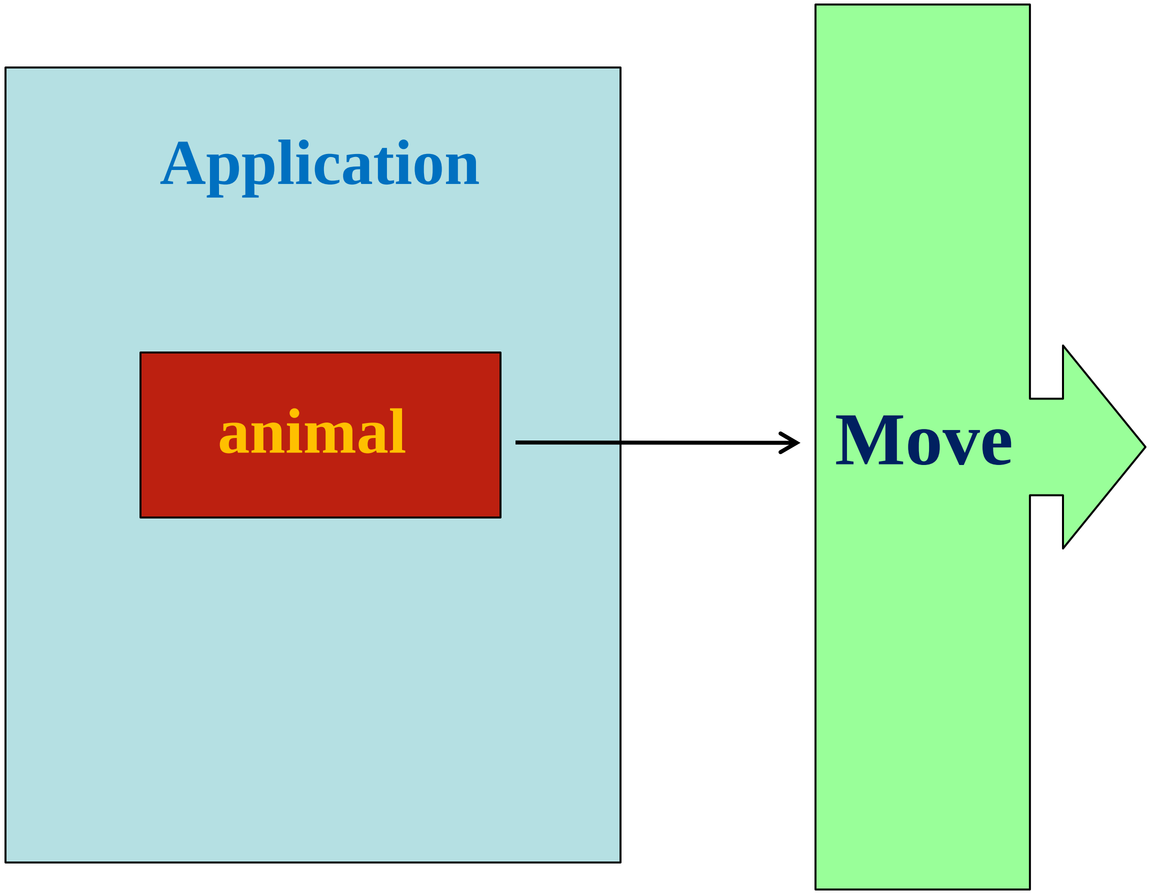
- Polymorphism: *from the Greek*
  "having multiple forms"
  - In programming languages, the ability to assign a different meaning or usage to something in different contexts
    - *The same method can be called on different objects*
    - *They may respond to it in different ways*

Polymorphism

- Example: In a computer game that simulates the movement of animals we can send ‘**move**’ commands to different types of **animal**

- We send the commands via an animal reference which is the base class for the different animal types
  - But each type behaves differently once it receives the command
  - Such an approach leads to a readily extendable application

Polymorphism



Polymorphism

- **Polymorphism** is implemented through references to objects

- We can assign base class object references to any derived class object

Object Oriented Programming

- Introduction
- Base & Derived Classes
- Polymorphism
- Pointers to base class

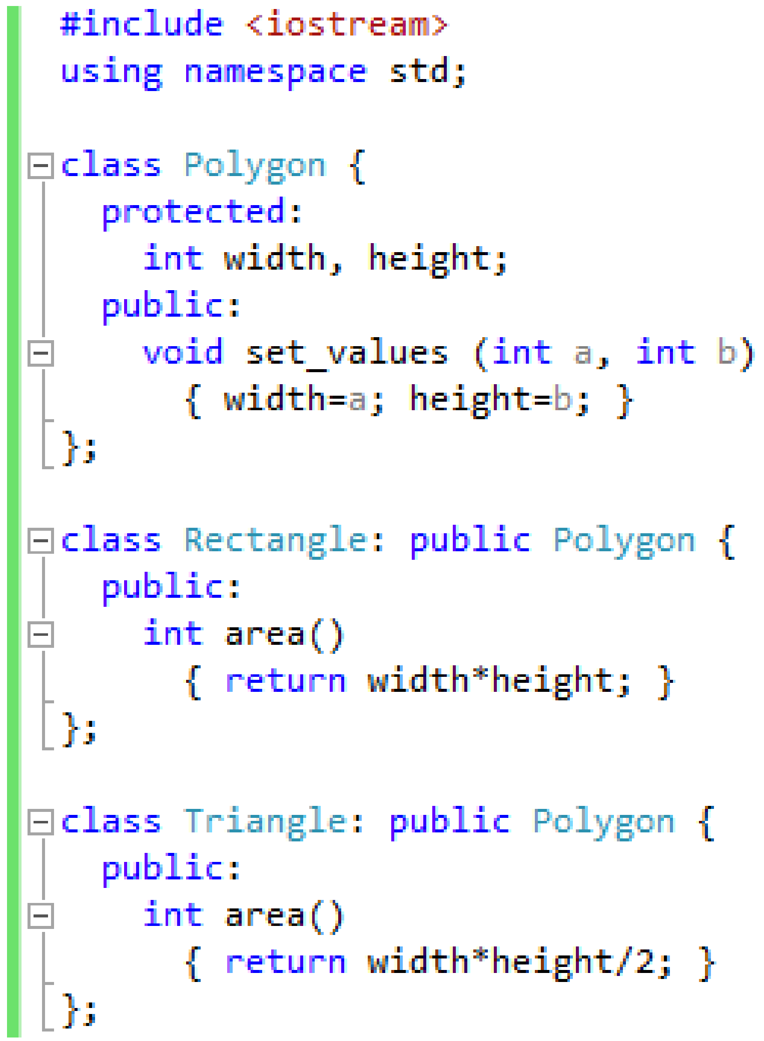
Pointers to base class

- One of the key features of class **inheritance** is that a **pointer** to a derived class is type-compatible with a pointer to its base class

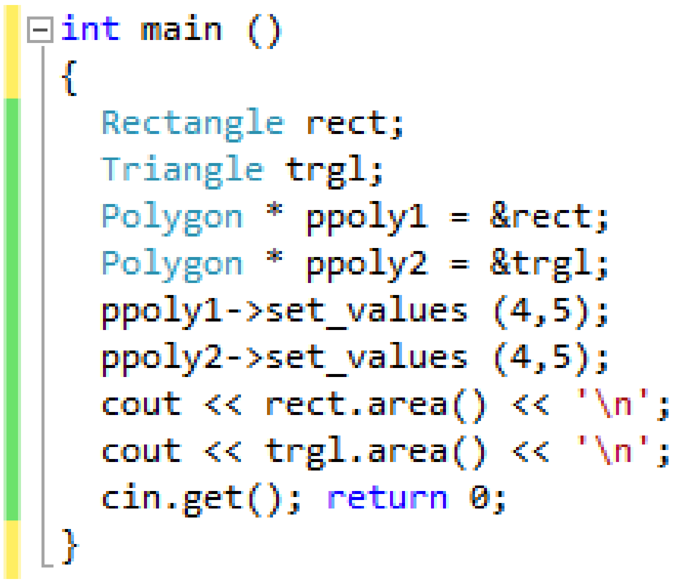
- **Polymorphism** is the **art** of taking advantage of this simple but powerful and versatile feature

Pointers to base class

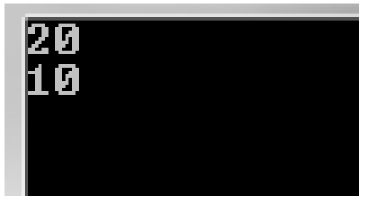
- Example 1:



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

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

class Rectangle: public Polygon {
  public:
    int area()
      { return width*height; }
};

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



```cpp
int main ()
{
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  cin.get(); return 0;
}
```



```
20
10
```

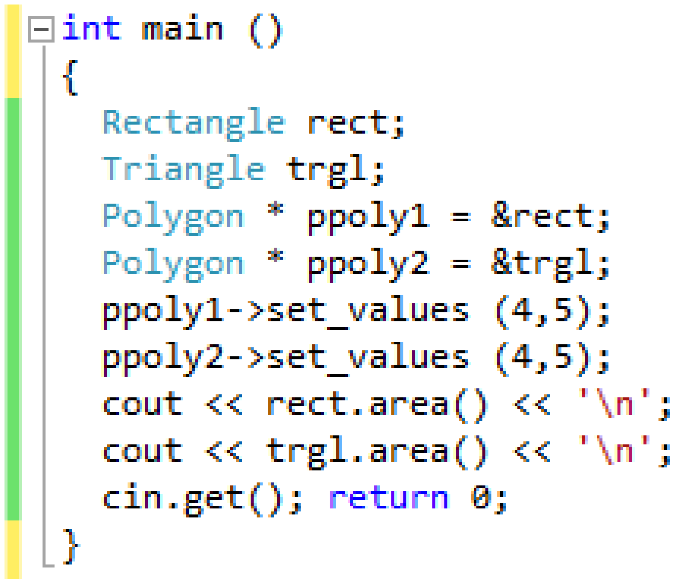
Pointers to base class

- Example 1:
  - Function main declares two pointers to Polygon (named ppoly1 and ppoly2). These are assigned the addresses of rect and trgl, respectively, which are objects of type Rectangle and Triangle. Such assignments are valid, since both Rectangle and Triangle are classes derived from Polygon
  - Dereferencing ppoly1 and ppoly2 (with *ppoly1 and *ppoly2) is valid and allows us to access the members of their pointed objects. But because the type of ppoly1 and ppoly2 is pointer to Polygon (and not pointer to Rectangle nor pointer to Triangle), only the members inherited from Polygon can be accessed, and **not** those of the derived classes Rectangle and Triangle
  - That is why the program above accesses the area members of both objects using rect and trgl directly, instead of the pointers; the pointers to the base class cannot access the area members

Pointers to base class

- Example 1:

```
ppoly1->set_values (4,5);
rect.set_values (4,5);
```



```
int main ()
{
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  cin.get(); return 0;
}
```

- Member area could have been accessed with the pointers to Polygon if area were a member of Polygon instead of a member of its derived classes, but the problem is that Rectangle and Triangle implement different versions of area, therefore there is not a single common version that could be implemented in the base class

Object Oriented Programming

- Introduction
- Base & Derived Classes
- Polymorphism
- Pointers to base class
- Virtual members

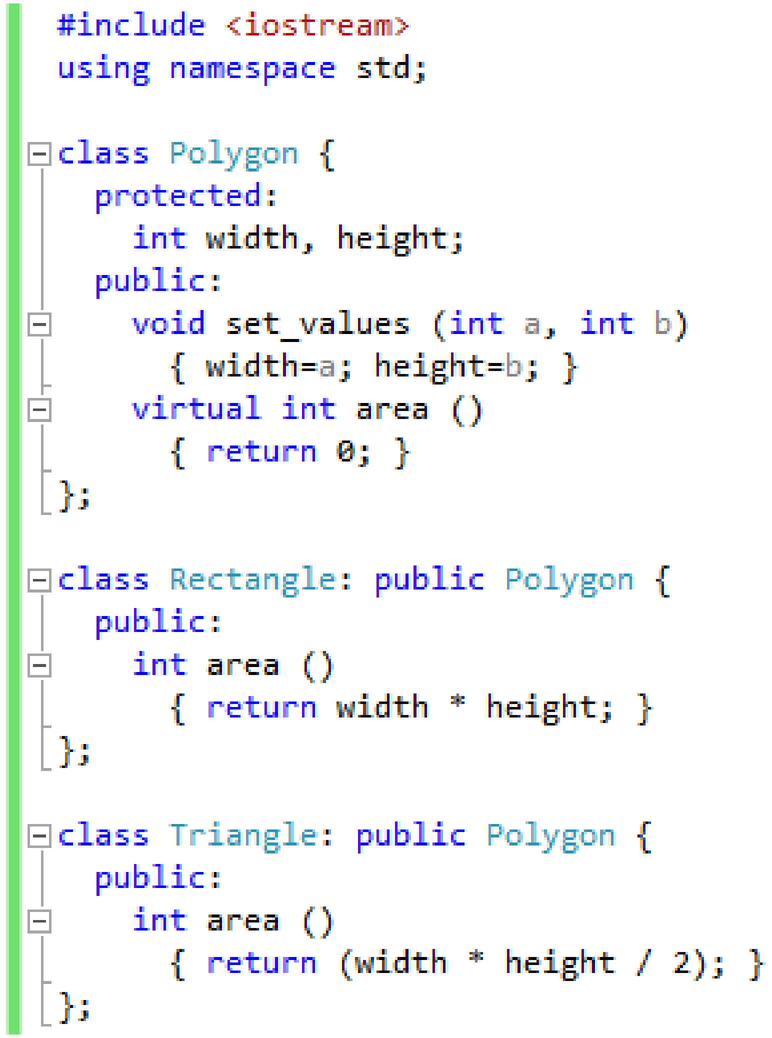
Virtual members

- A **virtual** member is a member function that can be redefined in a derived class, while preserving its calling properties through references

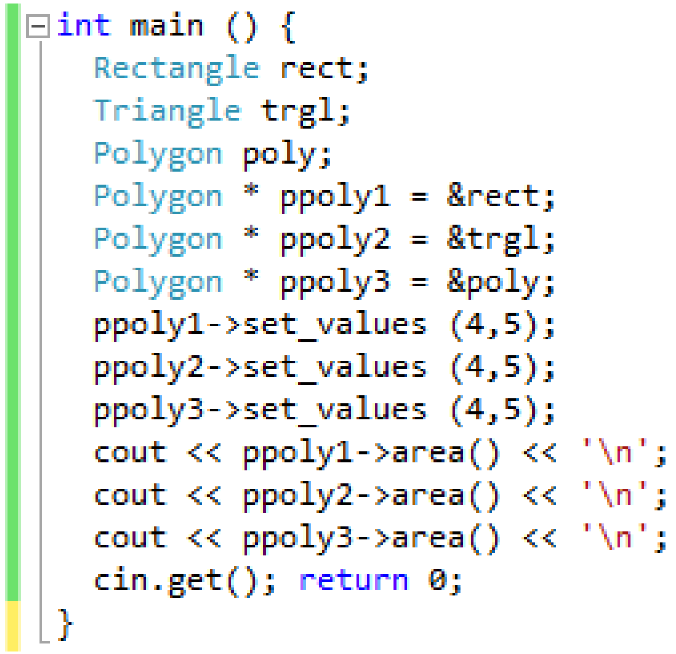
- The syntax for a function to become virtual is to precede its declaration with the **virtual** keyword

Virtual members

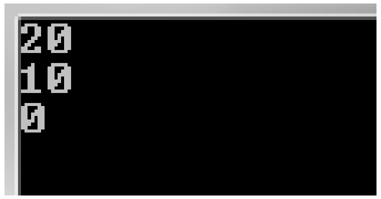
- Example 2:



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

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

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

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



```cpp
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon poly;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  Polygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  cout << ppoly3->area() << '\n';
  cin.get(); return 0;
}
```



```
20
10
0
```

Virtual members

- Example 2:
  - All three classes (Polygon, Rectangle and Triangle) have the same members: width, height, and functions set_values and area
  - The member function area has been declared as virtual in the base class because it is later redefined in each of the derived classes
  - Non-virtual members can also be redefined in derived classes, but non-virtual members of derived classes cannot be accessed through a reference of the base class: i.e., if virtual is removed from the declaration of area in the example above, **all three calls to area would return zero**, because in all cases, the version of the base class would have been called instead

Virtual members

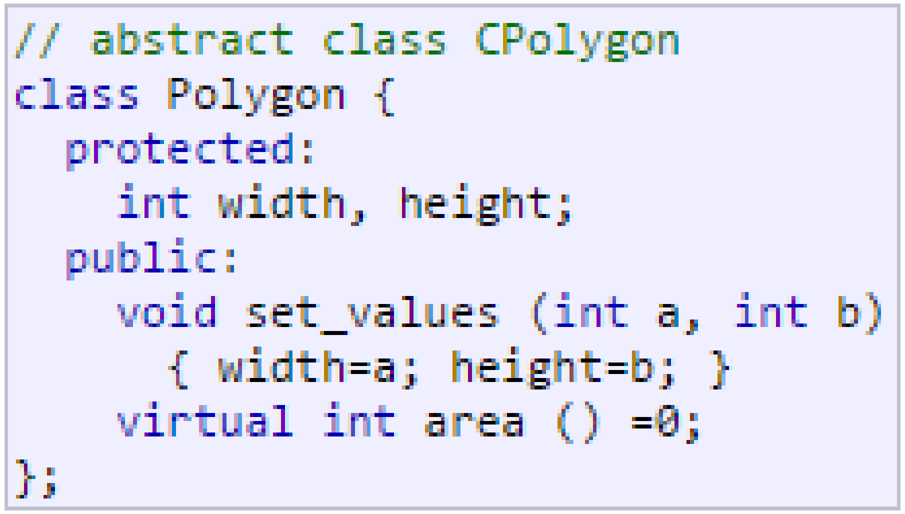
- Example 2:
  - Therefore, essentially, what the virtual keyword does is to allow a member of a derived class with the same name as one in the base class to be appropriately called from a pointer, and more precisely when the type of the pointer is a pointer to the base class that is pointing to an object of the derived class, as in the above example
  - A class that declares or inherits a virtual function is called a **polymorphic class**

Object Oriented Programming

- Introduction
- Base & Derived Classes
- Polymorphism
- Pointers to base class
- Virtual members
- Abstract base classes

Abstract base classes

- Abstract base classes are something very similar to the Polygon class in the previous example
  - They are classes that can only be used as base classes, and thus are allowed to have virtual member functions without definition (*known as pure virtual functions*)
  - The syntax is to replace their definition by **=0**
- An abstract base Polygon class could look like this:

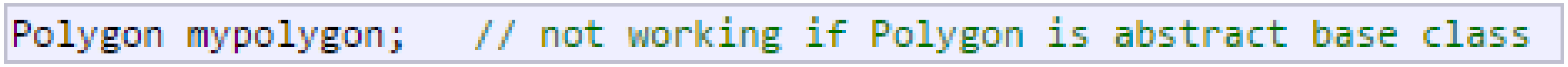


```
// abstract class CPolygon
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area () =0;
};
```

Abstract base classes

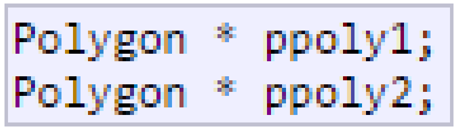
- Notice that area has no definition; this has been replaced by **=0**, which makes it a **pure virtual function**. Classes that contain at least one pure virtual function are known as **abstract base classes**

- Abstract base classes cannot be used to instantiate objects Therefore, this last abstract base class version of Polygon could not be used to declare objects like:



```
Polygon mypolygon;   // not working if Polygon is abstract base class
```

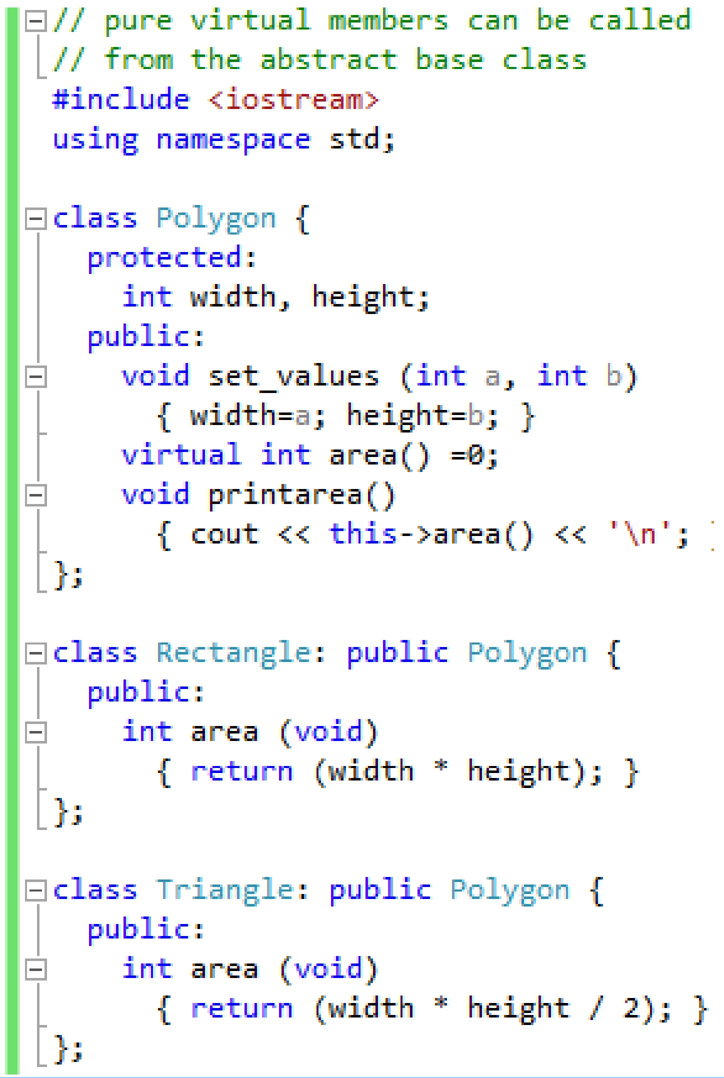
- But an abstract base class is not totally useless. It can be used to create pointers to it, and take advantage of all its polymorphic abilities. For example, the following pointer declarations would be valid:



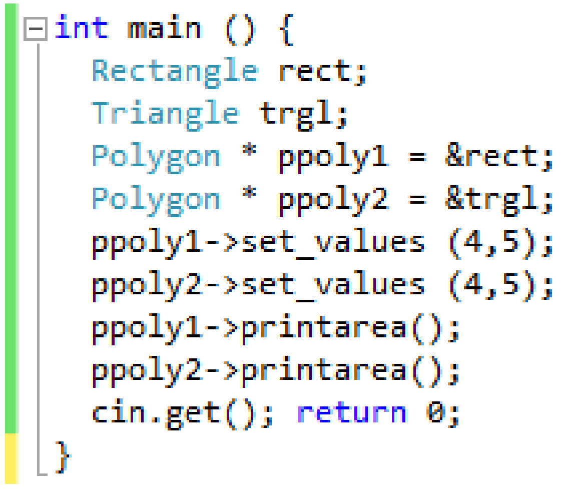
```
Polygon * ppoly1;
Polygon * ppoly2;
```

Pointers to base class

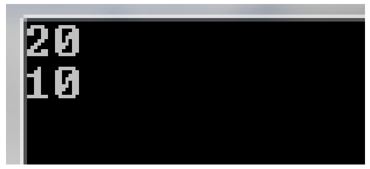
- Example 4:



```
// pure virtual members can be called
// from the abstract base class
#include <iostream>
using namespace std;

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

class Rectangle: public Polygon {
  public:
    int area (void)
      { return (width * height); }
};

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



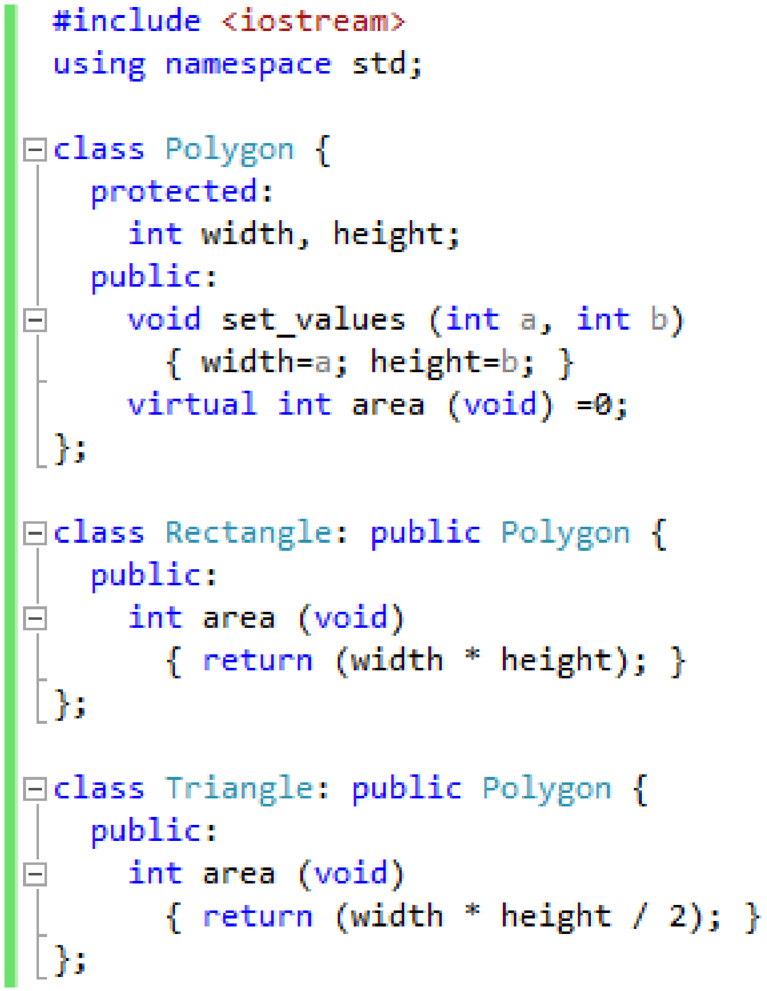
```
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  cin.get(); return 0;
}
```



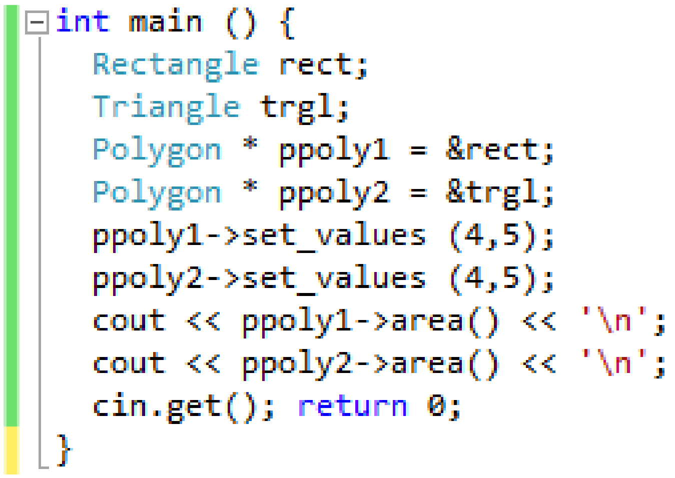
```
20
10
```

Abstract base classes

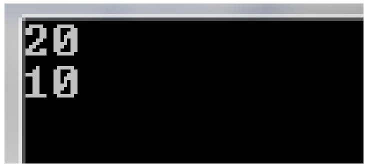
- Example 3:



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

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

class Rectangle: public Polygon {
  public:
    int area (void)
      { return (width * height); }
};

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



```
int main () {
   Rectangle rect;
   Triangle trgl;
   Polygon * ppoly1 = &rect;
   Polygon * ppoly2 = &trgl;
   ppoly1->set_values (4,5);
   ppoly2->set_values (4,5);
   cout << ppoly1->area() << '\n';
   cout << ppoly2->area() << '\n';
   cin.get(); return 0;
}
```



```
20
10
```

Abstract base classes

- Virtual members and abstract classes grant C++ polymorphic characteristics, most useful for Object-Oriented Projects

- Of course, the examples above are very simple use cases, but these features can be applied to arrays of objects or dynamically allocated objects

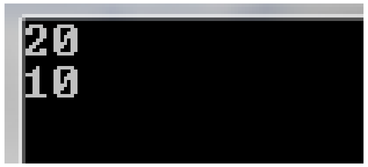
- Next example combines some of the features, such as dynamic memory, constructor initializers and polymorphism

Pointers to base class

- Example 5:

```cpp
// dynamic allocation and polymorphism
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    Polygon (int a, int b) : width(a), height(b) {}
    virtual int area (void) =0;
    void printarea()
      { cout << this->area() << '\n'; }
};

class Rectangle: public Polygon {
  public:
    Rectangle(int a,int b) : Polygon(a,b) {}
    int area()
      { return width*height; }
};

class Triangle: public Polygon {
  public:
    Triangle(int a,int b) : Polygon(a,b) {}
    int area()
      { return width*height/2; }
};
```

```cpp
int main () {
  Polygon * ppoly1 = new Rectangle (4,5);
  Polygon * ppoly2 = new Triangle (4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  delete ppoly1;
  delete ppoly2;
  cin.get(); return 0;
}
```


```
20
10
```

Pointers to base class

- Example 5:

  - Notice that the ppoly pointers:

```
Polygon * ppoly1 = new Rectangle (4,5);
Polygon * ppoly2 = new Triangle (4,5);
```

are declared being of type "pointer to Polygon", but the objects allocated have been declared having the derived class type directly (Rectangle and Triangle)

Polymorphism

- **Generic programming** refers to performing operations on different types using a single piece of code
  - *Examples include the application of searching and sorting algorithms to different data types*

- In C++ it is normally done using templates

Polymorphism

- Polymorphism is a key feature of object oriented programming

- Complex systems are able to be easily extended

  1. *The extendibility is provided by defining new classes within an inheritance hierarchy*

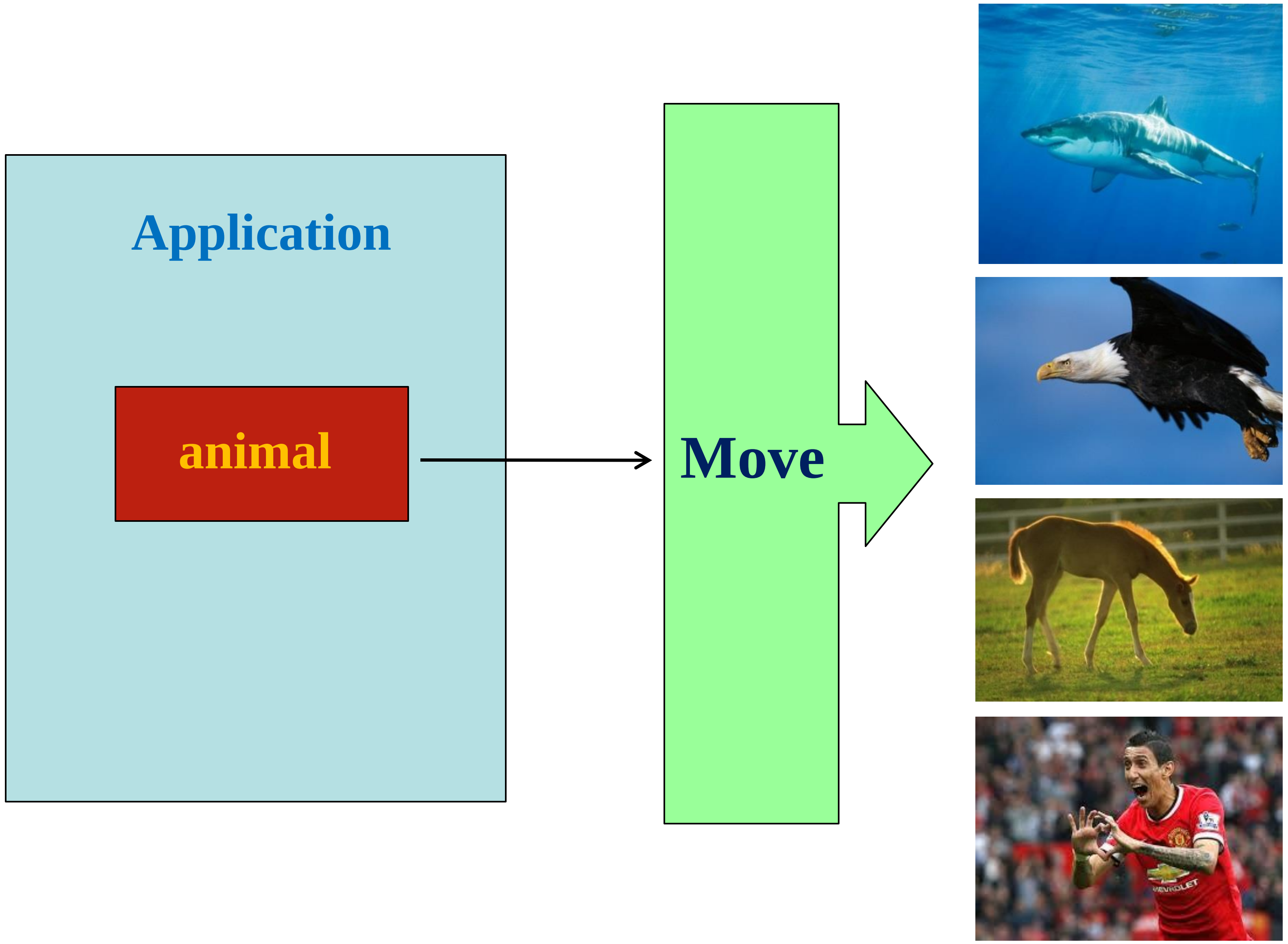
  2. *Objects of these new classes are accessed through a base class reference*

  3. *These objects add new behaviours to the system through a common interface to the application (the base class virtual functions)*

Polymorphism

- For example we could extend the list of animals to which we can send 'move' messages in our video game application
  - *Each animal is responsible for its own movement code which can easily 'plug-in' to the main application*
  - *Thus the application is easily extended with minimal changes to the main application code*

Polymorphism



Polymorphism - Simmary

- We have looked at how we can extend existing classes through the idea of inheritance
- We have seen how, by accessing derived classes through a base class pointer, object behaviour is determined at run time through polymorphism
- We have looked at the significance of polymorphism in object orientation
    - *Object oriented applications are easily extended with additional code mainly confined to new derived classes*

Object Oriented Programming

- Questions?!

mentor.hamiti@universitetiaab.com