Polymorphism
Learning Objectives

• Pointers to base class
• Virtual members
• Abstract base class
• Pure virtual function
Pointers to base class

```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;
    }
};

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';
  return 0;
}
```
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;}
};

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()<<endl;
    cout<<ppoly2->area()<<endl;
}
```

In function 'int main()':
35:19: error: 'class Polygon' has no member named 'area'
36:19: error: 'class Polygon' has no member named 'area'
Example 1.1

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

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

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()<<endl;
  cout<<ppoly2->area()<<endl;
}
```
#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); 
        }
};

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';
};
Pure Virtual Functions

• Virtual functions with no definition
• start with virtual keyword and ends with =0;

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

• Pure virtual functions can be given a small definition in the Abstract Class, which you want all the derived classes to have.
• Pure virtual function must be defined outside the class definition.
Example 2

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

class Base //Abstract base class
{
public:
  virtual void show() = 0; //Pure Virtual Function
};

void Base :: show() //Pure Virtual definition
{
  cout << "Pure Virtual definition\n";
}

class Derived : public Base
{
public:
  void show()
  { cout << "Implementation of Virtual Function in Derived class"; }
};

int main()
{
  Base *b;
  Derived d;
  b = &d;
  b->show();
}
```
Abstract base class

• contain at least one pure virtual function
• used to provide an interface for its sub classes
• classes inheriting an Abstract base class must provide definition to the pure virtual function, otherwise, they will also become abstract class
Abstract base classes

```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(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';
  return 0;
}
Example 3

```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()=0;
};

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

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

ing 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()<<endl;
  // cout<<ppoly2->area()<<endl;
  // cout<<ppoly3->area()<<endl;
}
Virtual Function

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

class A
{
    public:
        virtual void show()
        {
            cout << "Base class\n";
        }
};

class B : public A
{
    private:
        virtual void show()
        {
            cout << "Derived class\n";
        }
};

int main()
{
    A *a;
    B b;
    a = &b;
    a -> show();
}
```
Learning Objectives

• Function templates
• Class templates
• Templates and Inheritance
Function template

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

template <class T>
T addnum(T a, T b)
{
    return a + b;
}

int main()
{
    int x = 7, y = 43, z;
    z = addnum(x, y);
    cout << z << endl;
}
```
Function Template with Multiple Parameters

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

template <class first, class second>
first smaller(first a, second b){
  return (a<b?a:b);
}

int main()
{
  int x=89;
  double y=56.78;
  cout<<smaller(x,y);
}
```
```cpp
#include <iostream>
using namespace std;

template <class T>
class AddNumber{
  T first;
  T second;

  public:
  T addNum(T a, T b);
};

template <class T>
T AddNumber<T>::addNum(T a, T b)
{
  first=a;
  second=b;
  return first+second;
}

int main()
{
  AddNumber<int> obj;
  cout<< obj.addNum(4,5)<<endl;

  AddNumber<double> obj1;
  cout<<obj1.addNum(4.5,5.5)<<endl;
}
Class Template

template <class T, class U>
class AddNumber{
  T first;
  U second;

  public:
  T addNum(T a, U b);
};

template <class T, class U>
T AddNumber<T,U>::addNum(T a, U b)
{
  first=a;
  second=b;
  return first+second;
}

int main()
{
  AddNumber<int, int> obj;
  cout<< obj.addNum(4,5)<<endl;
  AddNumber<double, double> obj1;
  cout<<obj1.addNum(4.5,5.5)<<endl;
  AddNumber<double, int> obj2;
  cout<<obj2.addNum(4.5,5)<<endl;
}
Template Specialization

template <class T, class U>
T AddNumber<T,U>::addNum(T a, U b)
{
    first=a;
    second=b;
    return first+second;
}

template<>  
class AddNumber<string,string>  
{
    string first;
    string second;
    public:
    string addNum(string a, string b);
};

string AddNumber<string,string>::addNum(string a, string b)
{
    first=a;
    second=b;
    return (a + " " + b);
}

int main()
{
    AddNumber<int, int> obj;
    cout<< obj.addNum(4,5)<<endl;

    AddNumber<double, double> obj1;
    cout<<obj1.addNum(4.5,5.5)<<endl;

    AddNumber<double, int> obj2;
    cout<<obj2.addNum(4.5,5)<<endl;

    AddNumber<string,string> obj3;
    cout<<obj3.addNum("Hello", "World")<<endl;
}
Template and Inheritance

```cpp
class printValue : public AddNumber<double, double>
{
  public:
    void display()
    {
      cout << (first + second) << endl;
    }
};

int main()
{
  AddNumber<int, int> obj;
  cout << obj.addNum(4, 5) << endl;

  AddNumber<double, double> obj1;
  cout << obj1.addNum(4.5, 5.5) << endl;

  AddNumber<double, int> obj2;
  cout << obj2.addNum(4.5, 5) << endl;

  AddNumber<string, string> obj3;
  cout << obj3.addNum("Hello", "World") << endl;

  printValue obj4;
  obj4.addNum(40, 50);
  obj4.display();
}```
Template and Inheritance

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

template <class T>
class A
{
  public:
    A(T a): x(a)\{
    protected:
    T x;
  };

template <class T>
class B: public A<int>
{
  public:
    B():A(100)\{cout<<x<<endl;\}
};

int main()
{
  B<int>test;
}
```
Example 1

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

template <typename T>
void fun(const T& x)
{
    static int count = 0;
    cout << "x = " << x << " count = " << count << endl;
    ++count;
    return;
}

int main()
{
    fun<int> (1);
    cout << endl;
    fun<int>(1);
    cout << endl;
    fun<double>(1.1);
    cout << endl;
    return 0;
}
```
```cpp
#include <iostream>
using namespace std;

template <class T>
class Test
{
private:
    T val;
public:
    static int count;
    Test() { count++; }
};

template<class T>
int Test<T>::count = 0;

int main()
{
    Test<int> a;
    Test<int> b;
    Test<double> c;
    cout << Test<int>::count << endl;
    cout << Test<double>::count << endl;
}
```
#include <iostream>
using namespace std;

template <class T>
T max (T &a, T &b)
{
    return (a > b)? a : b;
}

template <>
int max <int> (int &a, int &b)
{
    cout << "Called ";
    return (a > b)? a : b;
}

int main ()
{
    int a = 10, b = 20;
    cout << max <int> (a, b);
}
#include <iostream>
using namespace std;

template <class T>
T maxF(T x, T y)
{
    return (x > y)? x : y;
}

int main()
{
    cout << maxF(3, 7) << endl;
    cout << maxF(3.0, 7.0) << endl;
    cout << maxF(3, 7.0) << endl;
    return 0;
}