Polymorphism

Polymorphism is the ability of different functions to be invoked with the same name.

There are two forms:

Static polymorphism is the common case of overriding a function by providing additional definitions with different numbers or types of parameters. The compiler matches the parameter list to the appropriate function.

Dynamic polymorphism relies on parent classes to define **virtual** functions which child classes may redefine.

When this virtual member function is called for an object of the parent class, the execution dynamically chooses the appropriate function to call - the parent function if the object really is the parent type, or the child function if the object really is the child type.

Pointers to base class

One of the key features of derived classes is that a pointer to a derived (ie. child) class is type-compatible with a pointer to its base (ie. parent or super) class.

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

Example

Recall the following example code from last week:

```
class CPolygon {
  protected:
    int width, height;
  public:
    void set_values(int a, int b) {
      width=a;
      height=b;
    } // end of method set_values
}; // end of class CPolygon

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

class CTriangle: public CPolygon {
  public:
    int area() {
```

```
      return (width*height / 2);
    } // end of method area
}; // end of class CTriangle
```

Now consider the following code that *exercises* it:

```
main () {
  CRectangle rect;
  CTriangle trgl;

  CPolygon* ppoly1 = &rect;
  CPolygon* ppoly2 = &trgl;

  // polymorhpic access of CPolygon::set_values
  ppoly1->set_values(4,5);
  ppoly2->set_values(4,5);

  cout << rect.area() << endl;
  cout << trgl.area() << endl;
} // end of main
```

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 assignments.

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.

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 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 useful!

Abstract base classes

Abstract base classes are something very similar to the CPolygon class of our previous example.

The only difference is that in our previous example we have defined a valid area() function with a minimal functionality for object instances of

class CPolygon (like the object poly), whereas in an abstract base class we could leave the method area() without implementating it at all.

This is done by appending =0 (equal to zero) to the function declaration.

Any method with such a definition is known as a *pure virtual function.*

Example

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

Notice how we appended =0 to virtual **int** area() method *instead* of specifying an implementation for the function.

Abstract Classes

Any class that contains at least one pure virtual function is an **abstract** class.

The main difference between an abstract class and a regular polymorphic class is that in abstract class, at least one of its members lacks an implementation (and so we cannot create object instances of it).

Note: a class that cannot instantiate objects is not totally useless:

we can create pointers to it and take advantage of all its *polymorphic* abilities!

Example

Consider the following changes to our CPolygon class:

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

CRectangle and CTriangle have the usual implementation.

Now consider the following code that exercises these new class definitions:

```
main () {
  CPolygon* ppoly1 = new CRectangle;
  CPolygon* ppoly2 = new CTriangle;
  ppoly1->set_values(4,5);
  ppoly2->set_values(4,5);
  ppoly1->printarea();
  ppoly2->printarea();
  delete ppoly1;
  delete ppoly2;
}
```

Notice that the ppoly pointers are declared to be of type pointer to CPolygon.

However the objects dynamically allocated have been declared having the derived class type directly!

Function templates

Function templates are special functions that can operate with **generic** types.

This allows us to create a function template whose *functionality* can be adapted to more than one variable type or class without repeating the code for each type.

For example, to create a template function that returns the larger of two Thingy object instances we could use:

```
template <class Thingy>
Thingy GetMax (Thingy a, Thingy b) {
  if (a > b) {
    return a;
  } else {
    return b;
  } // end of if-then-else
} // end of template function GetMax
```

Here we have created a template function with Thingy as its template parameter.

This template parameter represents a type that has not yet been specified, but that can be used in the template function as if it were a regular type.

To get GetMax to compare two integer values we can write:

```
int x,y;
GetMax<int> (x,y);
```

When the compiler encounters this call to a template function, it uses the template to *automatically* generate a function replacing each appearance of Thingy by the type passed as the *actual* template parameter (**int** in this case) and then calls it.

This process is automatically performed by the compiler and is invisible to the programmer.

Class templates

We also have the possibility to write *class templates*, so that a class can have members that use template parameters as types:

```
template <class T>
class pair {
    T values[2];
  public:
    pair(T first, T second) {
      values[0] = first;
      values[1] = second;
    } // end of constructor
}; // end of template class pair<T>
```

The class that we have just defined serves to store two elements of any valid type.

So, if we wanted to declare an object of this class to store two integer values of type **int** with the values 115 and 36 we would write:

pair<**int**> myobject (115, 36);