Polymorphic Container Problem

- A serious complaint about STL attacks the very paradigm on which STL is based.

- One of STL's central tenants is that containers directly contain there objects, and this is the root of many of its efficiency claims.

- But in doing this, STL runs contrary to one of the very important tenants in object oriented programming: polymorphic references.

- For example, a program may contain a list of shapes, where shape is the polymorphic superclass of other concrete classes such as ellipse, rectangle and triangle

- Class shape may have a pure virtual method called draw(), while the concrete subtypes all implement the draw() method.

- The program may iterate the list of shapes calling the polymorphic draw()method which will execute the appropriate code block for each object in the list.

```
struct Point
{
  int x,y;

  Point(int x, int y) : x(x), y(y)
  {}
  void print()
  {
    cout << "X: " << x << " Y: " << y;
  }
};
```

```
struct Shape
{
  Point origin;

  Shape(Point p) : origin(p)
  {}

  virtual void draw()=0;
};
```

```
struct Ellipse : public Shape
{
  int radius;

  Ellipse(Point o, int r) : Shape(o), radius(r)
  {}

  virtual void draw()
  {
   cout << "Ellipse with Origin: ";
    origin.print();
    cout << " and Radius: " << radius << endl;
  }
};
```

```
struct Rectangle : public Shape
{
  int width, height;

  Rectangle (Point topleft, int w, int h): Shape(topleft), width(w), height(h)
  {}

  virtual void draw()
  {
    cout << "Rectangle with Origin: ";
    origin.print();
    cout << " and Width: " << width << " Height: " << height << endl;
  }
};
```

```
struct Shape
{
  Point origin;

  Shape(Point p) : origin(p)
  {}

  virtual void draw()=0;
};
```

```
  list <Shape> shapeList;

  shapeList.push_back(e);
  shapeList.push_back(r);

  foreach (Shape &s, shapeList)
  {
    s.draw();
  }
```

```
/usr/include/c++/4.2.1/bits/stl_list.h: In instantiation of 'std::_List_node<Shape>':
/usr/include/c++/4.2.1/bits/list.tcc:73:   instantiated from 'void std::_List_base<_Tp, _Alloc>::_M_clear() [with _Tp = Shape, _Alloc =
std::allocator<Shape>]'
/usr/include/c++/4.2.1/bits/stl_list.h:348:   instantiated from 'std::_List_base<_Tp, _Alloc>::~_List_base() [with _Tp = Shape, _Alloc =
std::allocator<Shape>]'
/usr/include/c++/4.2.1/bits/stl_list.h:408:   instantiated from here
/usr/include/c++/4.2.1/bits/stl_list.h:101: error: cannot declare field 'std::_List_node<Shape>::_M_data' to be of abstract type 'Shape'
../src/poly1.cpp:16: note:   because the following virtual functions are pure within 'Shape':
../src/poly1.cpp:22: note: virtual void Shape::draw()
```

- If class shape is pure virtual, the code will not compile because list will attempt to generate a shape object (and it can't because its has a pure virtual member).

```
struct Shape
{
  Point origin;

  Shape(Point p) : origin(p)
  {}

  virtual void draw()
  {
   origin.print();
  }
}
```

```
  Shape* shapes[2];

  shapes[0] = new Ellipse(Point (1,1), 10);
  shapes[1] = new Rectangle(Point(2,2), 20, 10);

  for (int i=0; i<2; i++)
  {
    shapes[i]->draw();
  }
```

```
X: 1 Y: 1X: 2 Y: 2
```

- If class shape is not pure virtual the code will compile, but it will fail at run time: When the list is iterated to draw all the shapes, it will be shape::draw() which is called each time instead of ellipse::draw() and rectangle::draw().

- This is because the insertions into the list entail a copy of the object: shapeList.push_back() will do call shape::operator=(&shape) with the ellipse object as a parameter.

Reconsider...

```
Ellipse   e(Point (1,1), 10);
Rectangle r(Point(2,2), 20, 10);

Shape shapes[2];

shapes[0] = e;
shapes[1] = s;

for (int i=0; i<2; i++)
{
  shapes[i].draw();
}
```

```
../src/poly1.cpp: In function 'void polytest()':
../src/poly1.cpp:62: error: invalid abstract type 'Shape' for 'shapes'
../src/poly1.cpp:16: note:   because the following virtual functions are pure within 'Shape':
../src/poly1.cpp:22: note: virtual void Shape::draw()
```

Pointers?

```
Shape* shapes[2];

shapes[0] = new Ellipse(Point (1,1), 10);
shapes[1] = new Rectangle(Point(2,2), 20, 10);

for (int i=0; i<2; i++)
{
  shapes[i]->draw();
}
```

```
Ellipse with Origin: X: 1 Y: 1 and Radius: 10
Rectangle with Origin: X: 2 Y: 2 and Width: 20 Height: 10
```

Pointers to local variables...

```
Ellipse   e(Point (1,1), 10);
Rectangle r(Point(2,2), 20, 10);

Shape* shapes[2];
shapes[0] = &e;
shapes[1] = &r;

for (int i=0; i<2; i++)
{
  shapes[i]->draw();
}
```

```
Ellipse with Origin: X: 1 Y: 1 and Radius: 10
Rectangle with Origin: X: 2 Y: 2 and Width: 20 Height: 10
```

- It becomes apparent that one level of indirection is needed to solve the problem.

- An obvious solution is to change the list of shapes to a list of pointers to shapes

```
list <Shape*> shapeList;

shapeList.push_back(&e);
shapeList.push_back(&r);

foreach (Shape *s, shapeList)
{
  s->draw();
}
```

- This solution seems to work, but it will likely lead to run time errors, for if variables e and are automatic, they will be destructed at end of their blocks.

- If the list's scope lives on past the end of this block, it will be left containing invalid objects (pointers to arbitrary places in or beyond the stack)

```
shapeList.push_back(new Ellipse(Point (1,1), 10));
shapeList.push_back(new Rectangle(Point(2,2), 20, 10));
```

- One possible solution is to insist that any object placed in such a list must be allocated from the heap:
- Besides the fact that unsuspecting programmers might not read the documentation and violate this rule, this solution leaves open another problem.
- When the list is destructed, it will leave all of its referenced objects on the heap without calling their destructors or deallocating their memory.
- One could always subclass list < shape* > and destroy the contained objects in the subclass's destructor, but this would defeat a lot of the convenience of using STL's containers: one would have to write a lot of simple constructors and a new destructor for every variation of an STL container.

A reasonable solution

- An alternative solution is to change the behavior of pointers rather than containers.

- One could make a new reference template class who's destructor would take care of destructing its referent.

```
template < class T >
class Ref2
  {
  public:
    Ref2(const T &s)                              {KillData = true;  t = s.clone();}
    Ref2(T *s)                                    {KillData = false; t = s;}
    Ref2(const Ref2 < T > &r)                     {KillData = true;
                                                   t = r.t?r.t->clone():NULL;}
    ~Ref2()                                       {if (t && KillData) delete t;}
    Ref2& operator= (const Ref2 < T > & r)        {if (t && KillData) delete t;
                                                   KillData = true;
                                                   t = r.t?r.t->clone():NULL;
                                                   return *this; }
    T* operator->() const                         {return t;}
    int operator< (const Ref2 < T > & r) const {return t?r.t?(*t) < (*r.t):false:true;}
    operator T&()   const                         {return *t;}
    operator T*()   const                         {return t;}
    T& operator*()  const                         {return *t;}
  protected:
    T *t;
  private:
    bool KillData;
  };
```

Shape & Decedents must implement clone

```
struct Shape
{
  Point origin;

  Shape(Point p) : origin(p)
  {}

  virtual void draw()=0;
  virtual Shape* clone() const {return 0;};
};
```

```
struct Ellipse : public Shape
{
  int radius;

  Ellipse(Point o, int r) : Shape(o), radius(r)
  {}

  virtual void draw()
  {
    //...
  }
  Shape* clone() const
  {
   return new Ellipse(origin, radius);
  }
};

struct Rectangle : public Shape
{
  int width, height;

  Rectangle (Point topleft, int w, int h): Shape(topleft), width(w), height(h)
  {}

  virtual void draw()
  {
    //...
  }
  Shape* clone() const
  {
    return new Rectangle(origin, width, height);
  }
};
```

- the T& constructor is called which sets KillData to true;
    - this means that when shapeList is destroyed,
    - it will call the destructors of the Ref2 objects which will in turn destroy the referent objects

```
list <Ref2 <Shape> > shapeList;
shapeList.push_back(Ellipse(Point (1,1), 10));
shapeList.push_back(Rectangle(Point(2,2), 20, 10));
```

- one can also signal the class not to destroy the object (if that is what is needed) by using the T* constructor instead:

```
static Ellipse persistentEllipse(Point(20,20), 22);
shapeList.push_back(&persistentEllipse);
```

- When using the iterators of shapeList, one must pay attention to the extra level of indirection:

```
list < Ref2 < Shape > >::iterator i;
for (i=shapeList.begin(); i!=shapeList.end(); i++)
  (*i)->draw();
```

```
foreach (Shape *s, shapeList)
{
  s->draw();
}
```

- One of problems with this solution is that it defeats a lot of the efficiency of the STL.
- The STL goes to great lengths to efficiently store its objects in blocks (separating memory allocation/deallocation and construction/destruction). T
- he Ref2class blatantly allocates and deallocates its referents one at a time!
- Run time efficiency is also compromised by the extra level of indirection.

Alternative:Pointer Container Library

```
#include <boost/ptr_container/ptr_list.hpp>
using namespace boost;
```

```
  ptr_list <Shape>  shapeList;

  shapeList.push_back(new Ellipse(Point (1,1), 10));
  shapeList.push_back(new Rectangle(Point(2,2), 20, 10));
```

```
ptr_list <Shape>::iterator i;
for (i=shapeList.begin(); i!=shapeList.end(); i++)
  i->draw();
```

```
foreach (Shape &s, shapeList)
{
  s.draw();
}
```