CS310

“Advanced C++:
Templates and Generic Programming”

Templates & Design
The Polymorphic Power of Templates

Hüseyin Akcan
huseyin.akcan@ieu.edu.tr
The Polymorphic Power of Templates

- **Polymorphism** is the ability to associate different specific behaviors with a single generic notation.

- Polymorphism is also a cornerstone of the object-oriented programming paradigm, which in C++ is supported mainly through class inheritance and virtual functions. We call this *dynamic polymorphism*.

- However, templates also allow us to associate different specific behaviors with a single generic notation, but this association is generally handled at compile time, which we refer to as *static polymorphism*. 
**Dynamic Polymorphism**

- The art of polymorphic design consists of identifying a common set of capabilities among related object types and declaring them as virtual function interfaces in a common base class.

![Diagram of GeoObj, Circle, Line, and Rectangle classes showing virtual function interfaces](image)
Dynamic Polymorphism

```cpp
// common abstract base class GeoObj for geometric objects.
class GeoObj {
    public:
        // draw geometric object:
        virtual void draw() const = 0;
        // return center of gravity of geometric object:
        virtual Coord center_of_gravity() const = 0;
        ...;
};

// concrete geometric object class Circle.
// - derived from GeoObj.
class Circle : public GeoObj {
    public:
        virtual void draw() const;
        virtual Coord center_of_gravity() const;
        ...;
};

// draw inhomogeneous collection of GeoObjs.
void drawElems (std::vector<GeoObj*> const& eles) {
    for (unsigned i=0; i<eles.size(); ++i) {
        eles[i]->draw(); // call draw() according to type of element.
    }
};

std::vector<GeoObj*> coll;
coll.push_back(&l);
coll.push_back(&c);
drawElems(coll);
```
Static Polymorphism

- Templates can also be used to implement polymorphism
- Concrete classes are defined independently from each other
- The polymorphic power is then enabled when templates are instantiated with the concrete classes.
- Instead of a hierarchy of geometric classes, we have several individual geometric classes
Static Polymorphism

// concrete geometric object class Circle.
// - not derived from any class.
class Circle {
    public:
        void draw() const;
        Coord center_of_gravity() const;
        ...
};

// concrete geometric object class Line.
// - not derived from any class.
class Line {
    public:
        void draw() const;
        Coord center_of_gravity() const;
        ...
};

// draw homogeneous collection of GeoObjs.
template <typename GeoObj>
void drawElems (std::vector<GeoObj> const& elems) {
    for (unsigned i=0; i<elems.size(); ++i) {
        elems[i].draw();  // call draw() according to type of element.
    }
}

Disadvantage: heterogeneous collections can no longer be handled transparently.
C++ Code Jam

- DrawElems function accepts a std::vector as the parameter.
- Modify the function to accept any kind of std container
  - Test for std::vector and std::list

Solution: poly.cpp
**Dynamic versus Static Polymorphism**

- Polymorphism implemented via inheritance is *bounded* and *dynamic*:
  - *Bounded* means that the interfaces of the types participating in the polymorphic behavior are predetermined by the design of the common base class (other terms for this concept are invasive or intrusive).
  - *Dynamic* means that the binding of the interfaces is done at run time (dynamically).

- Polymorphism implemented via templates is *unbounded* and *static*:
  - *Unbounded* means that the interfaces of the types participating in the polymorphic behavior are not predetermined (other terms for this concept are noninvasive or nonintrusive).
  - *Static* means that the binding of the interfaces is done at compile time (statically).
**Strengths and Weaknesses**

Dynamic polymorphism in C++ exhibits the following strengths:

- Heterogeneous collections are handled elegantly.
- The executable code size is potentially smaller (because only one polymorphic function is needed, whereas distinct template instances must be generated to handle different types).
- Code can be entirely compiled; hence no implementation source must be published (distributing template libraries usually requires distribution of the source code of the template implementations).

In contrast, the following can be said about static polymorphism in C++:

- Collections of built-in types are easily implemented. More generally, the interface commonality need not be expressed through a common base class.
- Generated code is potentially faster (because no indirection through pointers is needed a priori and nonvirtual functions can be inlined much more often).
- Concrete types that provide only partial interfaces can still be used if only that part ends up being exercised by the application.
Generic Programming: STL

• Static polymorphism leads to the concept of generic programming.

Generic programming is a subdiscipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization. Generic programming focuses on representing families of domain concepts.

• By far the most significant contribution in this area is the STL

The STL is a framework that provides a number of useful operations, called algorithms, for a number of linear data structures for collections of objects, called containers. Both algorithms and containers are templates. However, the algorithms are not member functions of the containers. Instead, the algorithms are written in a generic way so that they can be used by any container. To do this, the designers of STL identified an abstract concept of iterators that can be provided for any kind of linear collection.
template <class Iterator>
Iterator max_element (Iterator beg, // refers to start of collection.
        Iterator end)  // refers to end of collection.
{
    // use only certain Iterator's operations to traverse all elements.
    // of the collection to find the element with the maximum value.
    // and return its position as Iterator.
    ...
};

template <class T, ...>
class vector {
    public:
        typedef ... const_iterator; // implementation-specific iterator.
        ...
        const_iterator begin() const; // iterator for start of collection.
        const_iterator end() const; // iterator for end of collection.
        ...
};

template <class T, ...>
class list {
    public:
        typedef ... const_iterator; // implementation-specific iterator.
        ...
        const_iterator begin() const; // iterator for start of collection.
        const_iterator end() const; // iterator for end of collection.
        ...
};

// compute position of maximum value
pos = std::max_element(coll.begin(),coll.end());
Pre-STL containers

- Containers that exist before STL used polymorphism and inheritance. Thus each container is a separate class, extended from base Container class.

- This design has performance problems:
  - **Runtime:** Heavy use of polymorphism for each call
  - **Space:** Unable to add built-in types into containers