Advanced C++

For : COP 3330.
Object oriented Programming (Using C++)
http://www.compgeom.com/~piyush/teach/3330

Source: Lutz Kettner.

Piyush Kumar

The user.

```
#define private     public
#define protected public
#define class       struct
```

Types of users

- We can distinguish between two kinds of protection a design can provide:
  - a user that makes occasional mistakes
  - a user that willingly tries to get around the protection mechanism

Const correctness

Revision.

```
                        // the pointer,     the data it refers to
                        // ---------------------------------------
      int*        p;
      int* const q;
const int*        r;
const int* const s;
```

```
                        // the pointer,     the data it refers to
                        // ---------------------------------------
      int*        p;    // non-const       non-const
      int* const q;     // const           non-const
const int*        r;    // non-const       const
const int* const s;     // const           const
```

Const declaration

```
struct A {
        const int i;
        A() : i(42) {}
};
```

This pointer and const.

```
struct C {
// hidden parameter: T* const this;
  void foo();
// hidden parameter: const T* const this;
  void bar() const;
};
```

Make Temporary Return Objects in C++ Classes Const

- L-values: can be used for the left side of an assignment, they are non-const.
- R-values: cannot be used for the left side of an assignment. . They are const.
- For example the post-increment operator requires an l-value, but is itself an r-value.

Make Temporary Return Objects in C++ Classes Const

- Thus, we cannot write:
  - int i; i++ ++; // second ++ forbidden!
  - Or i++++;
  - Error: error.cpp:5: error: non-lvalue in increment
- But:
  - struct A {
    A operator++ (int); // the post increment operator
    };
  - Now, lets try: A a; a++++; //compiles!

Make Temporary Return Objects in C++ Classes Const

- It works, because a++ returns a temporary object of type A.
- But it probably does not do what one would expect.
- Since the second ++ works on a temporary object, a itself gets only incremented once.
- We can forbid the second increment explicitly by making the return type, the type of the temporary object, const. This should be considered for all similar temporary return types.

```
struct A {
        // the post increment operator
        const A operator++ (int);
};
```

Empty Classes

C++ classes are often "empty"!

```
#include <iostream>

using namespace std;

struct X {
};

int main(){
      cout << sizeof(X) << endl;
      return 0;
}
```

Empty Classes

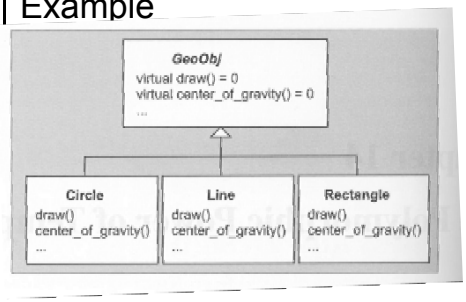
```
#include <iostream>

using namespace std;

struct X {
};

class Y:public X {
};

int main(){
      cout << sizeof(X) << endl;
      cout << sizeof(Y) << endl;
      return 0;
}
```

EBCO: Empty base class Optimization

Polymorphism

- Recap: Ability to associate different specific behaviors with a single generic notation.
- (Many forms or shapes)
- What you have seen:
  - Dynamic Polymorphism

Dynamic Polymorphism Example



Static Polymorphism

```
#include "coord.hpp"

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

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

Static Polymorphism

```
#include "statichier.hpp"
#include <vector>

// draw any GeoObj
template <typename GeoObj>
void myDraw (GeoObj const& obj)
{
    obj.draw();    // call draw() according to type of object
}

// process distance of center of gravity between two GeoObjs
template <typename GeoObj1, typename GeoObj2>
Coord distance (GeoObj1 const& x1, GeoObj2 const& x2)
{
    Coord c = x1.center_of_gravity() - x2.center_of_gravity();
    return c.abs(); // return coordinates as absolute values
}

// 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
    }
}
```

Static Polymorphism

```
int main()
{
    Line l;
    Circle c, c1, c2;

    myDraw(l);        // myDraw<Line>(GeoObj&) => Line::draw()
    myDraw(c);        // myDraw<Circle>(GeoObj&) => Circle::draw()

    distance(c1,c2);  // distance<Circle,Circle>(GeoObj1&,GeoObj2&)
    distance(l,c);    // distance<Line,Circle>(GeoObj1&,GeoObj2&)

    // std::vector<GeoObj*> coll;    // ERROR: no heterogeneous
                                     //      collection possible
    std::vector<Line> coll;   // OK: homogeneous collection possible
    coll.push_back(l);        // insert line
    drawElems(coll);          // draw all lines
}
```

Static Polymorphism

- All types must be determined at compile time.
- Heterogeneous collections can no longer be handled transparently.
- Generated code is potentially faster than dynamic polymorphism.

CRTP: Curiously recurring template pattern

- General class of techniques that consists of passing a derived class as a template argument to one of its own base classes.

```
// The Curiously Recurring Template Pattern (CRTP)
class derived : public base<derived> {
        // ...
};
```

CRTP


```
// The Curiously Recurring Template Pattern
// (CRTP)

template <typename Derived>
class CuriousBase {
        //…

};

class Curious : public CuriousBase<Curious> {
        // ...
        // Only valid if the size of CuriousBase<Curious>
        // can be determined independently of Curious.
};
```

CRTP: Alternative outline.

```
// The Curiously Recurring Template Pattern
// (CRTP)

template <typename Derived>
class CuriousBase {
        //…

};

template <typename T>
class CuriousT : public CuriousBase<CuriousT<T> > {
        // ...

};
```

CRTP: Alternative outline.

```
// The Curiously Recurring Template Pattern
// (CRTP)

template < template<typename> class Derived >
class MCuriousBase {
        //…

};

template <typename T>
class MoreCuriousT : public MCuriousBase<MoreCuriousT> {
        // ...

};
```

CRTP Concrete Example Counting Objects

A Generic solution to object counting.

```
#include <stddef.h>

template <typename CountedType>
class ObjectCounter {
 private:
  static size_t count;   // number of existing objects

 protected:
  // default constructor
  ObjectCounter() {
    ++count;
  }

  // copy constructor
  ObjectCounter (ObjectCounter<CountedType> const&) {
    ++count;
  }

  // destructor
  ~ObjectCounter() {
    --count;
  }

 public:
  // return number of existing objects:
  static size_t live() {
    return count;
  }
};

// initialize counter with zero
template <typename CountedType>
size_t ObjectCounter<CountedType>::count = 0;
```

CRTP Concrete Example Counting Objects

```
#include "objectcounter.hpp"
#include <iostream>

template <typename CharT>
class MyString : public ObjectCounter<MyString<CharT> > {
  //...
};

int main()
{
   MyString<char> s1, s2;
   MyString<wchar_t> ws;

   std::cout << "number of MyString<char>:    "
        << MyString<char>::live() << std::endl;
   std::cout << "number of MyString<wchar_t>: "
        << ws.live() << std::endl;
}
```

CRTP and the current assignment

- the graph knows the node and the edge class that are supposed to work together, and therefore the graph class passes itself as template argument to both types.

Another CRTP application

- Implement Inequality in terms of equality.

```
class A {
public:
    bool operator == (const A& a) const;
    bool operator != (const A& a) const {
        return ! (*this == a);
    }
    // ...
};
```

Another CRTP application

- Implement Inequality in terms of equality.

```
template <class T>
class Inequality {
public:
    bool operator != (const T& t) const {
        return ! (static_cast<const T&>(*this) == t);
    }
};

class A : public Inequality<A> {
public:
        bool operator == (const A& a) const;
};
```

More CRTP usage.

- The same technique can be used to implement a base class for iterators that contains all those small member functions that are defined in terms of a much smaller set of member functions.

Proxy Classes

Is this legal?

```
int data[10][20];

void processInput(int dim1, int dim2){
        int data[dim1][dim2];
        …
}

…

int *data = new int[dim1][dim2];
```

Proxy classes

- A dynamic two-dimensional array of integers could be declared in C++ as follows:

```
class Array2D {
    public:
      Array2D( int dim1, int dim2);
      // ...
};
```

Proxy Classes

- Of course, in a program we would like use the array similar to the builtin (static) two-dimensional arrays and access an element as follows:

```
int main()
{
        Array2D a(5,10);
        // ...
        int i = a[2][8];    // …(a[2])[8]…
}
```

Proxy Classes

- However, there is no operator[][] in C++.
- Instead, we can implement operator[] to return conceptually a one-dimensional array, where we can apply operator[] again to retrieve the element.

Proxy Classes

```
class Array1D {
public:
   Array1D( int dim);
   int operator[](int i);
   // ...
};
class Array2D {
public:
   Array2D( int dim1, int dim2);
   Array1D& operator[](int i);
   // ...
};
```

The intermediate class Array1D is called *proxy class*, also known as *surrogate* [Item 30, Meyers97].

Proxy classes

- Conceptually, it represents a one-dimensional array.
- In this application we surely do not want to copy the elements to actually create a one-dimensional array.
- The proxy class will just behave as if it is an one-dimensional array and internally it will use a pointer to the two-dimensional array to implement its operations.

Double Dispatch

- is a mechanism that dispatches a function call to different concrete functions depending on the runtime types of multiple objects involved in the call.
  - Lookup (Myers Item 31, More effective C++)

Smart pointers

I shot an arrow into the air,
It fell to earth, I know not where.

"The Arrow and the Song"
H. W. Longfellow

Smart Pointers: unique_ptr

- Typical pointer usage.

```
void f() {
        MyClass *ptrmyclass = new MyClass;
        // … perform some operators
        delete ptrmyclass;
}
```

- Source of trouble!

What if you forgot a return in the middle?

Smart Pointers: unique_ptr

- A return in the middle of the function.
- An exception thrown.
  - Or else the function has to catch all exceptions.
- How do we avoid resource leaks?
  - Recap: valgrind? ☺

Smart Pointers: auto_ptr

```
// Fixing the last program : Complicated.

void f(){
        MyClass *ptr = new MyClass;

        try {
        …
        }

        catch( … ){
                delete ptr;
                throw;  // rethrow the exception
        }

        delete ptr;
}
```

Smart Pointers: auto_ptr

```
#include <memory>      // header for unique_ptr

void f(){
        // create and initialize a unique_ptr
        std::unique_ptr<MyClass> ptr(new MyClass);
        // … perform some operators

}
```

- delete and catch are no longer necessary!
- The smart pointer can free the data to which it points whenever the pointer itself gets destroyed.
- A unique_ptr is a pointer that serves as an owner of the object to which it refers to.
- As a result , the object gets destroyed when its unique_ptr gets destroyed.
- A requirement of unique_ptr is that its object has only one owner.

unique_ptr

- Has much of the same interface as an ordinary pointer (operator *, operator ->)
- Pointer arithmetic (such as ++) is not defined.
- Note:

```
std::auto_ptr<MyClass> ptr1(new MyClass);  // OK
std::auto_ptr<MyClass> ptr1 = new MyClass; // Error
```

Misusing unique_ptrS.

- Cannot share ownerships.
- Do not do reference counting.
- Do NOT meet the requirements for container elements.
  - When a unique_ptr is copied/assigned the source unique_ptr gets modified! Because it transfers its value rather than copying it.

Unit Testing

- **unit testing** is a procedure used to validate that individual units of source code are working properly.
- Unit = Smallest testable part of an application
- In C++, Smallest unit = Class
- Goal: Isolate each part of the program and show individual parts are correct.