CSE 165/ENGR 140
Intro to Object Orient Program
Lecture 22 – Multiple Inheritance
Announcement

- Lab #9 Open tomorrow
  - Due 4/23 at 11:59PM

- Lab Re-submission
  - Due 4/19 at 11:59PM

- Final project (in UCMCROPS)
  - Due date: 5/11 (Monday) at 11:59PM
  - Presentation date: 5/12 (Tuesday) at 3:00PM

- Reading assignment:
  - Ch. 9 (Vol. 2)
Multiple Inheritance

- Is it really necessary?
  - There is still debate about the need of multiple inheritance.
  - Consider it as a non-essential feature that is useful in some cases.
Multiple Inheritance

- Useful to solve cases where containers are designed to work with objects that must derive from a given base class.
Deciding to use multiple inheritance or not will lead to different possible designs for classes.
In C++, inheritance “inherits everything”
- Everything in a base class will also be part of the derived class.
- Is there a way to inherit the interface only?
  - The only way is to declare methods in the base class, creating an “interface class”.
  - Interface classes should be made abstract (with every method being a pure virtual method).
class Printable {
public:
    virtual ~Printable() { cout<<"~Printable()\n"; }
    virtual void print(ostream&) const = 0;
};

class Stringable {
public:
    virtual ~Stringable() { cout<<"~Stringable()\n"; }
    virtual string toString() const = 0;
};

class X : public Printable, public Stringable {
    int i;
public:
    X ( int x ) { i=x; }
    void print ( ostream& o ) const { o<<"out:"<<i; }
    string toString() const { ostringstream os; os<<"stout:"<<i;
        return os.str();
    }
};
Interface Inheritance 2/2

```cpp
void testPrintable ( const Printable& p ) {
    p.print(cout);
    cout << endl;
}

void testStringable(const Stringable& s) {
    string buf = s.toString();
    cout << buf << endl;
}

int main() {
    X x(7);
    testPrintable(x);
    testStringable(x);
}
```

**Output:**
```
out:7
stout:7
~Stringable()
~Printable()
```

- **Questions:**
  - What will happen with the two virtual destructors?
  - What will be printed?
  - How would you implement a similar example with templates?
class X {
    int i;
public:
    X ( int x ) { i=x; }
    void print ( ostream& o ) const { o<<"out:"<<i; }
    string toString() const { ostringstream os; os<<"stout:"<<i; return os.str(); }
};

template<class Printable>
void testPrintable ( const Printable& p ) {
    p.print(cout);
    cout << endl;
}

template<class Stringable>
void testStringable(const Stringable& s) {
    string buf = s.toString();
    cout << buf << endl;
}

int main() {
    X x(7);
    testPrintable(x);
    testStringable(x);
}
Implementation Inheritance

- Once again, in C++, inheritance “inherits everything”
  - Everything in a base class will also be part of the derived class

- We can use multiple inheritance to add capabilities to other classes.
  - Create *mixing classes* for added capabilities.
  - Mixing classes cannot be instantiated by themselves.
Implementation Inheritance 1/2

class Database {
public:
    Database ( const string& s ) { dbid=s; cout<<"Database\n"; }
    virtual ~Database() { cout<<"~Database\n"; }
    void open() { /* open connection */ }
    void close() { /* close connection */ }
private:
    string dbid;
};

class Countable { // a mixing class
public:
    long attach() { return ++count; }
    long detach() { return (--count>0)? count : (delete this, 0); }
    long refCount() const { return count; }
protected:
    Countable() { count=0; cout<<"Countable\n"; } // instantiated by derived classes
    virtual ~Countable() { cout<<"~Countable\n"; }
private:
    long count;
};

how will a self-delete work in multiple inheritance?
Implementation Inheritance 2/2

class DBConnection : public Database, public Countable {
public:
    DBConnection ( const string& dbStr ) : Database(dbStr)
    { cout<<"DBConnection\n"; open(); attach(); }
    ~DBConnection() { cout<<"~DBConnection\n"; close(); }
    DBConnection* getRef() { attach(); return this; }
private:
    // Disallow copy
    DBConnection(const DBConnection&);
    DBConnection& operator=(const DBConnection&);
};

int main() {
    cout << "=== Direct delete case ===\n";
    DBConnection* c = new DBConnection ("MyDatabase");
    delete c;
    cout << "=== Self delete case 1 ===\n";
    c = new DBConnection ("MyDatabase");
    c->detach();
    cout << "=== Self delete case 2 ===\n";
    DBConnection* c1 = new DBConnection ("MyDatabase");
    DBConnection* c2 = c1->getRef();
    c1->detach();
    c2->detach();
}
Implementation Inheritance Output

=== Direct delete case ===
Database
Countable
DBConnection
~DBConnection
~Countable
~Database

=== Self delete case 1 ===
Database
Countable
DBConnection
~DBConnection
~Countable
~Database

=== Self delete case 2 ===
Database
Countable
DBConnection
~DBConnection
~Countable
~Database