Programming C++
Lecture 4

Howest, Fall 2014

Instructor: Dr. Jennifer B. Sartor
Jennifer.sartor@elis.ugent.be

Few things from assign1

- Playing with pointers, or re-assigning boards
- Follow instructions ***exactly***
  - Function definitions
  - Input and output
  - Pointer arithmetic and calling delete

To remember for assign2

- Use a preprocessor wrapper (#ifndef BOOK_H) in .h file
- Make methods that don't change member variables const
- Use member initializer syntax (required to initialize member variables that are objects, but NOT for pointers) (also required to call base class constructor from derived class)
- Write a destructor, operator= and copy constructor if necessary (when member variable dynamically allocated)
- Make sure member variables are private or protected

Input to Main Function

```
int main(int argc, char* argv[]) {
    cout << "Number of arguments is " << argc << endl;
    for (int i = 0; i < argc; i++) {
        cout << "Argument " << i << " is " << argv[i] << endl;
    }
    ifstream inputFile(argv[1], ios::in);

}
```

Base Class Example

```
class Member {
public:
    Member(string name);
    Member( Member const &);
    Member& operator= (Member const &);
    ~Member();

    string getName() const;
    void setName(string name);
    void print() const;
private:
    string myName;
};
```

Derived Class Example

```
#include "Member.h"
class Employee : public Member {
public:
    Employee(string name, double money);
    Employee( Employee const &);
    Employee& operator= (Employee const &);
    ~Employee ();

    double getSalary() const;
    void setSalary(double money);
    void print() const;
private:
    double salary;
};
```

Employee Constructor

```
#include “Employee.h”
Employee::Employee( string name, double money )
    : Member(name)   //base class initializer syntax
{
    salary = money;
}
```

- C++ requires derived class constructor to call base class constructor to initialize inherited base class data members (if not explicit, default constructor would be called).

Constructor/Destructor Order

- When we instantiate a derived class:
  1. Base class's member object constructors execute (if they exist)
  2. Base class constructor executes
  3. Derived class's member object constructors execute
  4. Derived class constructor executes
- Destructors called in reverse order.
- Base class constructors, destructors and overloaded assignment operators are not inherited by derived classes. However derived class can call base class's version of these.

Instantiating Objects Example

```
1. #include "Member.h"
2. #include "Employee.h"
3. Member m1("Jill");
4. Employee e1("Jack", 65000);
5. Member *mPtr = &m1;
6. cout << mPtr->getName();  //what does this print?
7. mPtr->print();            //and this?
```

Instantiating Objects Example

```
1. #include "Member.h"
2. #include "Employee.h"
3. Member m1("Jill");
4. Employee e1("Jack", 65000);
5. Member *mPtr = &m1;
6. cout << mPtr->getName();  //Jill
7. mPtr->print();            //Jill
```

Instantiating Objects Example

```
1. #include "Member.h"
2. #include "Employee.h"
3. Member m1("Jill");
4. Employee e1("Jack", 65000);
5. Member *mPtr = &m1;
6. Employee *ePtr = &e1;
7. cout << ePtr->getName() << ePtr->getSalary();  //result?
8. ePtr->print();          //what function does this call?
```

Instantiating Objects Example

```
1. #include "Member.h"
2. #include "Employee.h"
3. Member m1("Jill");
4. Employee e1("Jack", 65000);
5. Member *mPtr = &m1;
6. Employee *ePtr = &e1;
7. cout << ePtr->getName() << ePtr->getSalary();  //Jack 65000
8. ePtr->print();      //Employee.print which calls Member.print
```

Instantiating Objects Example

1. #include “Member.h”
2. #include “Employee.h”
3. Member m1(“Jill”);
4. Employee e1(“Jack”, 65000);
5. Member *mPtr = &m1;
6. Employee *ePtr = &e1;
7. mPtr = &e1; //is this ok? Base class pointer to derived class?

Instantiating Objects Example

1. #include “Member.h”
2. #include “Employee.h”
3. Member m1(“Jill”);
4. Employee e1(“Jack”, 65000);
5. Member *mPtr = &m1;
6. Employee *ePtr = &e1;
7. mPtr = &e1;   //Yes, valid; all Employees are Members
8. ePtr = &m1;   //this valid? Derived class pointer to base class?

Instantiating Objects Example

```
1. #include "Member.h"
2. #include "Employee.h"
3. Member m1("Jill");
4. Employee e1("Jack", 65000);
5. Member *mPtr = &m1;
6. Employee *ePtr = &e1;
7. mPtr = &e1;    //Yes, valid;  all Employees are Members
8. ePtr = &m1;    //No, not all Members are Employees;
                  //compiler error
```

Instantiating Objects Example

```
1. #include "Member.h"
2. #include "Employee.h"
3. Member m1("Jill");
4. Employee e1("Jack", 65000);
5. Member *mPtr = &m1;
6. mPtr = &e1;        //yes, this is valid;  all Employees are Members
7. cout << mPtr->getName();     //what does this print?
8. cout << mPtr->getSalary();   //this ok?
9. mPtr->print();               //what function does this call?
```

Instantiating Objects Example

```
1. #include "Member.h"
2. #include "Employee.h"
3. Member m1("Jill");
4. Employee e1("Jack", 65000);
5. Member *mPtr = &m1;
6. mPtr = &e1;
7. cout << mPtr->getName();        //Jack
8. cout << mPtr->getSalary();      //compiler error
9. mPtr->print();              //calls Member's print: Jack
```

Introducing Polymorphism

- Member *mPtr = &e1;   mPtr->print();
- By default, the method that is called depends on the type of the handle, not the type of the object
- Polymorphism enables the compiler to call the more specific method, i.e. call based on the type of object dynamically.
- Because all derived class objects ARE base class objects, 1 base class pointer can enable calls to any number of derived class methods.
  - Program "in the general" rather than "in the specific"

Polymorphism!

- Member *mPtr = &e1;   mPtr->print();
- To get the Employee print function to be called, the method has to be declared **virtual** (in the .h)
- For virtual functions, the type of the object being pointed to determines function call, not type of handle.
  - At execution time we determine what function to call (not compile time), so it is done dynamically.
  - This is called **dynamic binding**

Polymorphism!

- Member *mPtr = &e1;   mPtr->print();
- Dynamic binding with virtual functions only works with pointer and reference handles (you need a level of indirection).
  - Member m1("Jill");
  - m1.print(); resolved at compile time => static binding!
- Base class declares functions as virtual, and implicitly for all derived classes that function is virtual (whether declared thus or not – virtualness is inherited).
- Derived class function can override/redefine base class regular or virtual function, or takes on base class's implementation if not defined

Base Class Example

```
class Member {
public:
    Member(string name);
    Member( Member const &);
    Member& operator= (Member const &);
    ~Member();

    string getName() const;
    void setName(string name);
    virtual void print() const;
private:
    string myName;
};
```

Derived Class Example

```
#include "Member.h"
class Employee : public Member {
public:
    Employee(string name, double money);
    Employee( Employee const &);
    Employee& operator= (Employee const &);
    ~Employee ();

    double getSalary() const;
    void setSalary(double money);
    virtual void print() const;  //keywork here unnecessary, but good practice.
private:
    double salary;
};
```

Instantiating Objects Example

```
1. #include "Member.h"
2. #include "Employee.h"
3. Member m1("Jill");
4. Employee e1("Jack", 65000);
5. Member *mPtr = &m1;
6. mPtr = &e1;
7. cout << mPtr->getName();          //Jack
8. mPtr->print();     //calls Employee's print: Jack 65000
```

Kinds of Assignments

- Base class pointer -> base class object = FINE
  - Invokes base class functionality
- Derived class pointer -> derived class object = FINE
  - Invokes derived class functionality
- Base class pointer to derived class object = FINE
  - Will invoke base class functionality unless functions declared virtual, then will invoke derived class functionality
- Derived class pointer to base class object = COMPILER ERROR (unless explicit cast)

Base class is a Derived class?

- Derived class pointer -> base class object
  - Could downcast?
  - DANGEROUS!

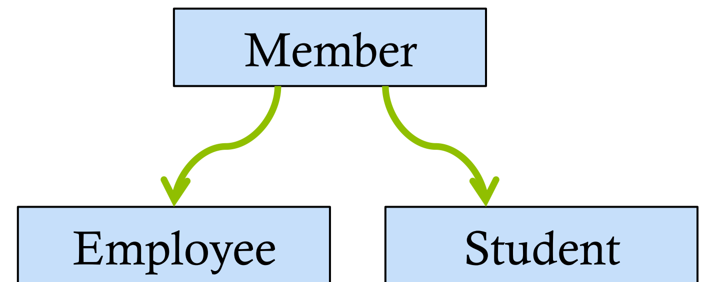
```
Member *mPtr;
…
Employee *ePtr = static_cast< Employee* > (mPtr);
ePtr->getSalary();
```

- We will see a safe way to do this – with dynamic cast.

Derived Class Example2

```
#include "Member.h"
class Student: public Member {
public:
    Student(string name, int id);
    Student(Student const &);
    Student& operator= (Student const &);
    ~Student ();

    int getUniqueID( ) const;
    void setUniqueID (int id);
    virtual void print( ) const;
private:
    int uniqueID;
};
```



Example of Polymorphism

```
vector < Member* > members(4);

members[0] = new Employee("Alice", 60000); //name & salary

members[1] = new Student("Bob", 987654);  //name & uniqueID

for (size_t i = 0; i < members.size(); i++) {
    members[i]->print();  //polymorphic behavior here

}
```

Example of Polymorphism

```
vector < Member* > members(4);

members[0] = new Employee(“Alice”, 60000); //name & salary

members[1] = new Student(“Bob”, 987654);  //name & uniqueID

for (size_t i = 0; i < members.size(); i++) {
    members[i]->print();
    //what if we want to change salary here – give everyone a raise?

}
```

Example of Polymorphism

```
vector < Member* > members(4);
members[0] = new Employee("Alice", 60000);
members[1] = new Student("Bob", 987654);  //name & uniqueID
for (size_t i = 0; i < members.size(); i++) {
    Employee *ePtr = dynamic_cast < Employee* > (members[i]);
    if (ePtr != 0) { //if downcast succeeded, we have Employee*
        ePtr->setSalary((ePtr->getSalary()) * 1.1);
    }
    members[i]->print();
}
```

Memory Management

```
vector < Member* > members(4);
members[0] = new Employee("Alice", 60000);
members[1] = new Student("Bob", 987654);  //name & uniqueID
for (size_t i = 0; i < members.size(); i++) {
    Employee *ePtr = dynamic_cast < Employee* > (members[i]);
    if (ePtr != 0) { //if downcast succeeded
        ePtr->setSalary((ePtr->getSalary()) * 1.1);
    }
    members[i]->print();
}
for (size_t i = 0; i < members.size(); i++) {
    delete members[i];
}
```

Destructors

- What happens if we call delete on a base class pointer to a derived class object?
  - Call base class destructor?
  - Derived class destructor?
  - Error?

Destructors

- What happens if we call delete on a base class pointer to a derived class object?
  - Call base class destructor?
  - Derived class destructor?
  - Error?
- This is undefined and can cause compiler warnings. BAD

Virtual Destructors

- When virtual methods exist, declare destructor **virtual** in base class.
- All derived classes destructors are then by default virtual as well (even though they have different names).
- Enables proper destruction of derived classes from base class pointers (behavior undefined if destructor not virtual)
- Constructors CANNOT be virtual.

Base Class Example

```
class Member {
public:
    Member(string name);
    Member( Member const &);
    Member& operator= (Member const &);
    virtual ~Member();

    string getName() const;
    void setName(string name);
    virtual void print() const;
private:
    string myName;
};
```

Abstract Classes

- An abstract class provides a common public interface for its class hierarchy. It is usually the base class.
- Class is made abstract by declaring 1 or more of its virtual functions to be "pure" in .h, no implementation in .cpp
  - virtual void print() const = 0;
  - Abstract classes are never instantiated (lack implementation)
- Abstract classes provide a framework but are incomplete. Derived classes must define missing pieces.

Pure Virtual

- Every concrete derived class *must* override all base-class pure virtual functions with concrete implementations.
  - If not overridden, derived class is abstract (can't be instantiated).
- A virtual-only function in base class has an implementation and gives derived class an option to override (as with regular functions).

Abstract Classes

- Abstract class can have data members and concrete functions (constructors/destructors) which go by normal inheritance rules.
- Can use pointers to abstract classes to use polymorphic functionality on all concrete derived classes.
  - Useful with container classes (vector of abstract base class)
  - Can use iterator to iterate over items in container class

Abstract Class Example

```
// Base class
class Shape  {
public:   // pure virtual function providing interface framework.
  virtual int getArea() = 0;
  void setWidth(int w) {
    width = w;
  }
  void setHeight(int h) {
    height = h;
  }
protected:
  int width;
  int height;
};
```

Abstract Class Example

```
// Derived class
class Rectangle: public Shape
{
public:
  int getArea() {
    return (width * height);
  }
};
```

```
// Derived class
class Triangle: public Shape
{
public:
  int getArea(){
    return (width * height)/2;
  }
};
```