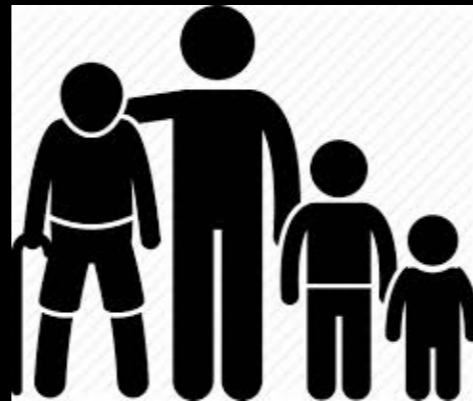


Inheritance



Tiziana Ligorio
Hunter College of The City University of New York

Announcements

- No IDE integration in GitHub: must have **local environment** — code, debug and test locally

Today's Plan



Recap

Useful C++ / OOP

Inheritance



Recap

OOP

Abstraction

Encapsulation

Information Hiding

Classes

Public Interface

Private Implementation

Constructors / Destructors

Interface

SomeClass.hpp
(same as **SomeClass.h**)

Include Guards:

Implementation

Tells linker "include only if it has not been included already by some other module"

```
#ifndef SOME_CLASS_H
#define SOME_CLASS_H

#include <somelibrary>
#include "AnotherClass.h"

class SomeClass
{
public:
    SomeClass(); //Constructor
    int methodOne();
    bool methodTwo();
    bool methodThree(int
                      someParameter);

private:
    int data_member_one_;
    bool data_member_two_;
}; //end SomeClass

#endif
```

```
#include "SomeClass.hpp"

SomeClass::SomeClass()
{
    //implementation here
}

int SomeClass::methodOne()
{
    //implementation here
}

bool SomeClass::methodTwo()
{
    //implementation here
}

bool SomeClass::methodThree(int
someParameter)
{
    //implementation here
}
```

Some (Perhaps Review) Useful Concepts

Default Arguments

```
void point(int x = 3, int y = 4);  
  
point(1,2); // calls point(1,2)  
point(1);   // calls point(1,4)  
point();    // calls point(3,4)
```

Order Matters!
Parameters without default arguments must go first.

Default Arguments

```
void point(int x = 3, int y = 4);  
  
point(1,2); // calls point(1,2)  
point(1);   // calls point(1,4)  
point();    // calls point(3,4)
```

Order Matters!
Parameters without default arguments must go first.

Similarly:

```
Person(int id, string first = "", string last = "");  
  
Person(143); // calls Person(143,"", "")  
Person(143, "Gina"); // calls Person("143","Gina", "")  
Person(423, "Nina", "Moreno"); // calls Person(423,"Nina","Moreno")
```


Default Arguments

```
void point(int x = 3, int y = 4);  
  
point(1,2); // calls point(1,2)  
point(1);   // calls point(1,4)  
point();    // calls point(3,4)
```

Order Matters!
Parameters without default arguments must go first.

```
Animal(std::string name = "", bool domestic = false, bool predator = false);
```

IS DIFFERENT FROM

```
Animal(std::string name, bool domestic = false, bool predator = false);
```

Overloading Functions

Same name, different parameter list (different function prototype)

```
int someFunction()  
{  
  
//implementation here  
  
} // end someFunction
```

```
int someFunction(string  
some_parameter )  
{  
    //implementation here  
  
} // end someFunction
```

Overloading Functions

Same name, different parameter list (different function prototype)

```
int someFunction()  
{  
    //implementation here  
} // end someFunction
```

```
int someFunction(string  
some_parameter )  
{  
    //implementation here  
} // end someFunction
```

```
int main()  
{  
    int x = someFunction();  
    int y = someFunction(my_string);  
    //more code here  
} // end main
```

Friend Functions

Functions that are **not members** of the class but **CAN access private members** of the class

Friend Functions

Functions that are **not members** of the class but **CAN** access **private members** of the class

Violates Information Hiding!!!

Yes, so don't do it unless appropriate and controlled



Friend Functions

DECLARATION:

```
class SomeClass
{
    public:
        // public member functions go here
        friend returnType someFriendFunction( parameter list);
    private:
        int some_data_member_;

}; // end SomeClass
```

IMPLEMENTATION (SomeClass.cpp):

Not a member function

```
returnType someFriendFunction( parameter list)
{
    // implementation here
    some_data_member_ = 35; //has access to private data
}
```

Operator Overloading

Desirable operator (=, +, -, == ...) behavior may not be well defined on objects


```
class SomeClass
{
    public:
        // public data members and member functions go here
        friend bool operator==(const SomeClass& object1,
                               const SomeClass& object2);

    private:
        // private members go here
}; // end SomeClass
```

Operator Overloading

IMPLEMENTATION (SomeClass.cpp):

Not a member function



```
bool operator==(const SomeClass& object1,  
               const SomeClass& object2)  
{  
    return ( (object1.memberA_ == object2.memberA_) &&  
            (object1.memberB_ == object2.memberB_) && ... );  
}
```


Enum

A user defined datatype that consist of integral constants

Why? Readability

```
enum season {SPRING, SUMMER, AUTUMN, WINTER };
```

```
enum animal_type {MAMMAL, FISH, BIRD};
```

Enum

A user defined datatype that consist of integral constants

Type name (like `int`)

Possible values:
like 0,1, 2, ...

Why? Readability

```
enum season {SPRING, SUMMER, AUTUMN, WINTER};
```

```
enum animal_type {MAMMAL, FISH, BIRD};
```

Enum

A user defined datatype that consist of integral constants

Type name (like `int`)

Possible values:
like 0,1, 2, ...

Why? Readability

```
enum season {SPRING, SUMMER, AUTUMN, WINTER };
```

```
enum animal_type {MAMMAL, FISH, BIRD};
```

By default = 0, 1, 2, ...

To change default:

```
enum ta_role {MAMMAL = 5, FISH = 10, BIRD = 20};
```

Inheritance



From General to Specific

What if we could *inherit* functionality from one class to another?

We can!!!

Inherit **public** members of another class

Basic Inheritance

```
class Printer
{
public:
    //Constructor, destructor

    void setPaperSize(const int size);
    void setOrientation(const string& orientation);
    void printDocument(const string& document);
private:
    // stuff here
}; //end Printer
```

Basic Inheritance

```
class Printer
{
public:
    //Constructor, destructor

    void setPaperSize(const int size);
    void setOrientation(const string& orientation);
    void printDocument(const string& document);
private:
    // stuff here
}; //end Printer
```

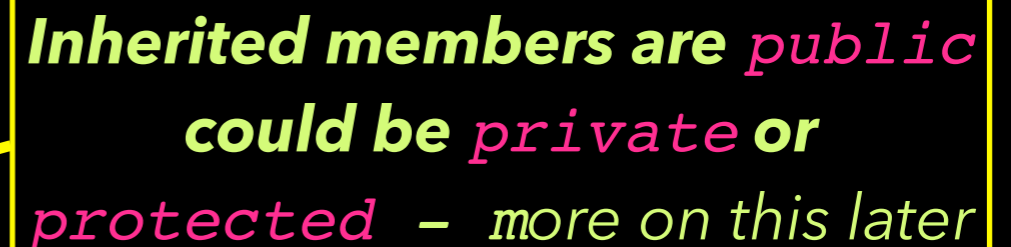
```
class BatchPrinter
{
public:
    //Constructor, destructor
    void addDocument(const string& document);
    void printAllDocuments();
private:
    vector<string> documents;
}; //end BatchPrinter
```

Basic Inheritance

```
class Printer
{
public:
    //Constructor, destructor

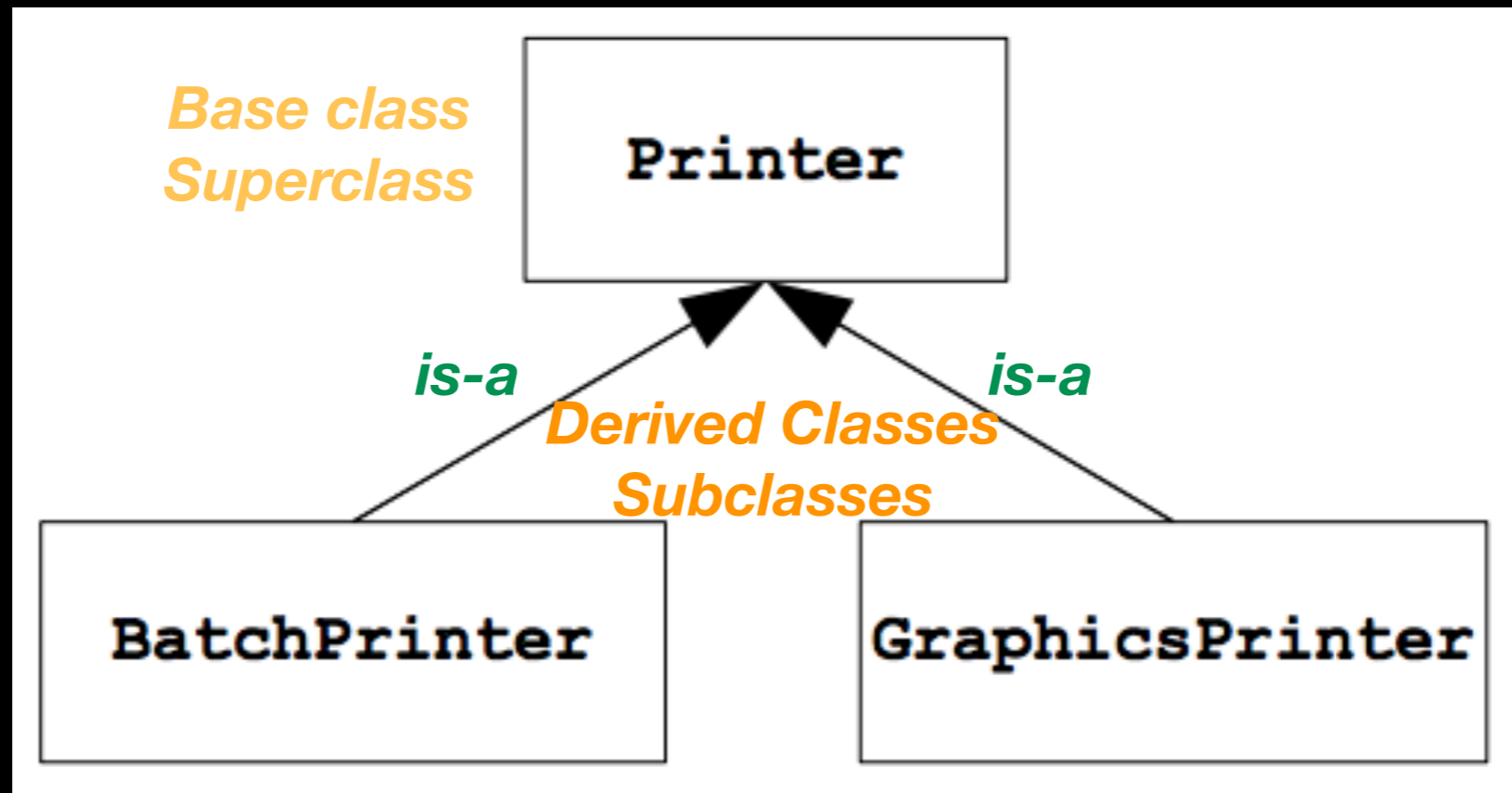
    void setPaperSize(const int size);
    void setOrientation(const string& orientation);
    void printDocument(const string& document);
private:
    // stuff here
}; //end Printer
```

Inherited members are *public* could be *private* or *protected* - more on this later



```
class BatchPrinter: public Printer // inherit from printer
{
public:
    //Constructor, destructor
    void addDocument(const string& document);
    void printAllDocuments();
private:
    vector<string> documents;
}; //end BatchPrinter
```


Basic Inheritance



```
void initializePrinter(Printer& p)
{
    //some initialization function
}
```

```
BatchPrinter batch;
initializePrinter(batch); //legal because batch is-a printer
```

Think of argument types as specifying **minimum requirements**

Lecture Activity

Can you think up more Printer subclasses

Can you think of multiple levels of inheritance
(specialization of a specialization)

**Lecture 3 Assignment on Gradescope
Login and submit NOW!!!**

Overloading vs Overriding

Overloading (independent of inheritance): Define new function with same name but different parameter list (different signature or prototype)

```
int someFunction(){ }  
int someFunction(string some_string){ }
```

Overriding: Rewrite function with *same signature* in derived class

```
int BaseClass::someMethod(){ }  
int DerivedClass::someMethod(){ }
```

```
class Printer
{
public:
    //Constructor, destructor

    void setPaperSize(const int size);
    void setOrientation(const string& orientation);
    void printDocument(const string& document);
private:
    // stuff here
}; //end Printer
```

```
class GraphicsPrinter: public Printer    // inherit from printer
{
public:
    //Constructor, destructor
    void setPaperSize(const int size);
    void printDocument(const Picture& picture); //some Picture object
private:
    //stuff here
}; //end GraphicsPrinter
```

Overrides setPaperSize()

Overloads printDocument()

main()

```
Printer base_printer;  
GraphicsPrinter graphics_printer  
Picture picture;  
// initialize picture here  
string document;  
// initialize document here
```

```
base_printer.setPaperSize(11); //calls Printer function
```

Printer

setPaperSize(int)
setOrientation(string)
printDocument(string)

GraphicsPrinter

setPaperSize(int)
printDocument(Picture)

main()

```
Printer base_printer;  
GraphicsPrinter graphics_printer  
Picture picture;  
// initialize picture here  
string document;  
// initialize document here
```

```
base_printer.setPaperSize(11); //calls Printer function  
graphics_printer.setPaperSize(60); // Overriding!!!
```

Printer

```
setPaperSize(int)  
setOrientation(string)  
printDocument(string)
```

GraphicsPrinter

```
setPaperSize(int)  
printDocument(Picture)
```

main()

```
Printer base_printer;  
GraphicsPrinter graphics_printer;  
Picture picture;  
// initialize picture here  
string document;  
// initialize document here
```

```
base_printer.setPaperSize(11); //calls Printer function  
graphics_printer.setPaperSize(60); // Overriding!!!  
graphics_printer.setOrientation("landscape"); //inherited
```

Printer

```
setPaperSize(int)  
setOrientation(string)  
printDocument(string)
```

GraphicsPrinter

```
setPaperSize(int)  
printDocument(Picture)
```

main()

```
Printer base_printer;
GraphicsPrinter graphics_printer;
Picture picture;
// initialize picture here
string document;
// initialize document here

base_printer.setPaperSize(11); //calls Printer function
graphics_printer.setPaperSize(60); // Overriding!!!
graphics_printer.setOrientation("landscape"); //inherited

graphics_printer.printDocument(string); //calls Printer inherited function
```

Printer

setPaperSize(int)
setOrientation(string)
printDocument(string)

GraphicsPrinter

setPaperSize(int)
printDocument(Picture)

main()

```
Printer base_printer;
GraphicsPrinter graphics_printer;
Picture picture;
// initialize picture here
string document;
// initialize document here

base_printer.setPaperSize(11); //calls Printer function
graphics_printer.setPaperSize(60); // Overriding!!!
graphics_printer.setOrientation("landscape"); //inherited

graphics_printer.printDocument(string); //calls Printer inherited function
graphics_printer.printDocument(picture); // Overloading!!!
```

Printer

setPaperSize(int)
setOrientation(string)
printDocument(string)

GraphicsPrinter

setPaperSize(int)
printDocument(Picture)

protected access specifier

```
class SomeClass
{
    public:
        // public members available to everyone

    protected:
        // protected members available to class members
        // and derived classes

    private:
        // private members available to class members ONLY
}; // end SomeClass
```

Important Points about Inheritance

Derived class inherits all **public** and **protected** members of base class

Does not have access to base class **private** members

Does not inherit constructor and destructor

Does not inherit assignment operator

Does not inherit friend functions and friend classes

Constructors

A class needs user-defined constructor if must initialize data members

Base-class constructor **always** called before derived-class constructor

If base class has only parameterized constructor, derived class **must supply constructor** that calls base-class constructor explicitly

Constructors

INTERFACE

```
class BaseClass
{
public:
    //stuff here

private:
    //stuff here
}; //end BaseClass
```

```
class DerivedClass: public BaseClass
{
public:
    DerivedClass();
    //stuff here

private:
    //stuff here
}; //end DerivedClass
```

IMPLEMENTATION

```
DerivedClass::DerivedClass()
{
    //implementation here
}
```

main()

```
DerivedClass my_derived_class;
//BaseClass compiler-supplied default constructor called
//then DerivedClass constructor called
```

Constructors

INTERFACE

```
class BaseClass
{
public:
    BaseClass();
    //may also have other
    //constructors
private:
    //stuff here
}; //end BaseClass
```

```
class DerivedClass: public BaseClass
{
public:
    DerivedClass();
    //stuff here
private:
    //stuff here
}; //end DerivedClass
```

IMPLEMENTATION

```
BaseClass::BaseClass()
{
    //implementation here
}
```

```
DerivedClass::DerivedClass()
{
    //implementation here
}
```

main()

```
DerivedClass my_derived_class;
//BaseClass default constructor called
//then DerivedClass constructor called
```

Constructors

INTERFACE

```
class BaseClass
{
public:
    BaseClass(int value);
    //stuff here

private:
    int base_member_;
}; //end BaseClass
```

```
class DerivedClass: public BaseClass
{
public:
    DerivedClass();
    //stuff here

private:
    //stuff here
}; //end DerivedClass
```

IMPLEMENTATION

```
BaseClass::
BaseClass(int value):
base_member_(value)
{
    //implementation here
}
main()
```

```
DerivedClass::DerivedClass()
{
    //implementation here
}
```

DerivedClass my_derived_class; ←

//PROBLEM!!! there is no default constructor to be called
//for BaseClass

Constructors

INTERFACE

```
class BaseClass
{
public:
    BaseClass(int value);
    //stuff here

private:
    int base_member_;
}; //end BaseClass
```

```
class DerivedClass: public BaseClass
{
public:
    DerivedClass();
    //stuff here

private:
    static const int INITIAL_VAL = 0;
}; //end DerivedClass
```

IMPLEMENTATION

```
BaseClass::
BaseClass(int value):
base_member_(value)
{
    //implementation here
}
```

main()

```
DerivedClass::DerivedClass():
BaseClass(INITIAL_VAL)
{
    //implementation here
}
```

Fix



```
DerivedClass my_derived_class;
// BaseClass constructor explicitly called by DerivedClass
//constructor
```


Destructors

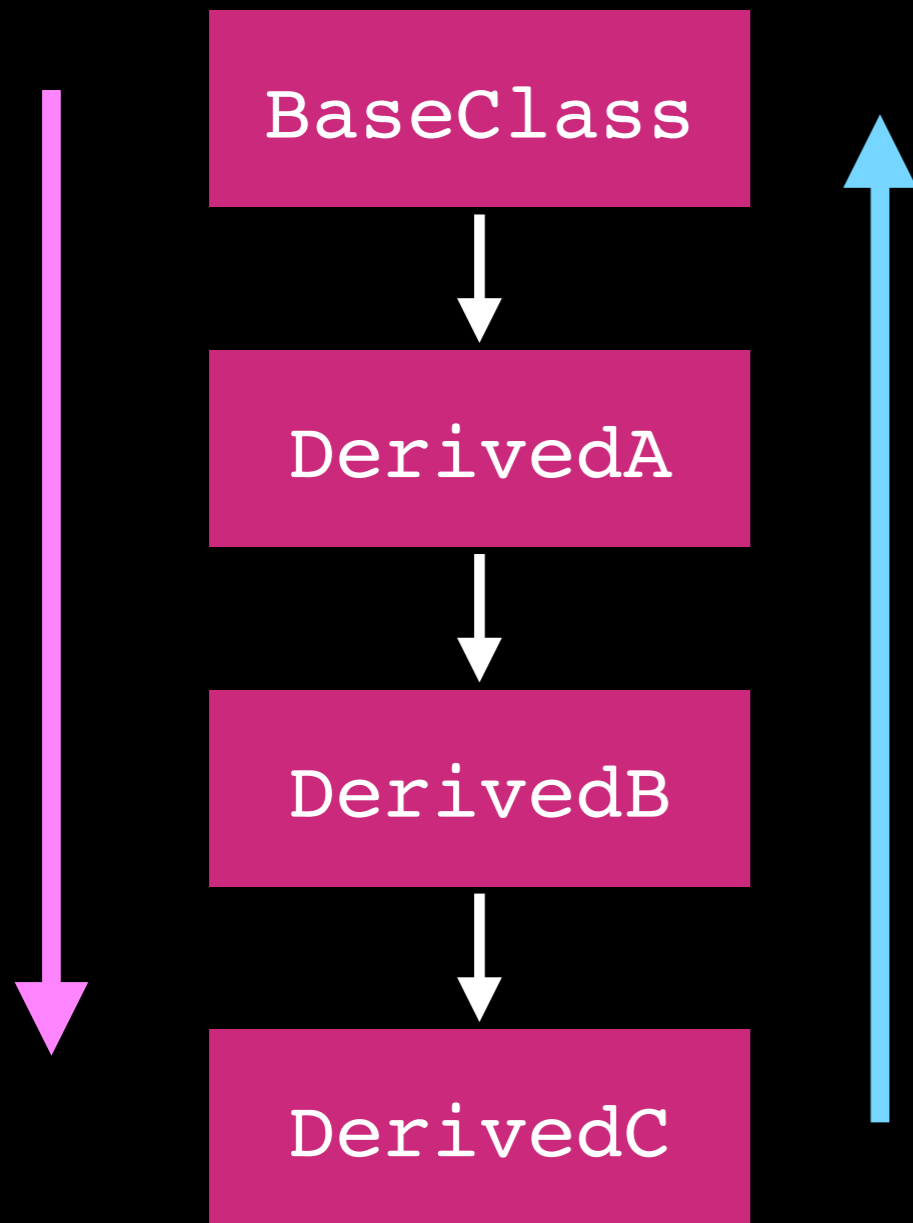
Destructor invoked if:

- program execution left scope containing object definition
- `delete` operator was called on object that was created dynamically

Destructors

Derived class destructor **always causes base class** destructor to be called implicitly

Derived class destructor is called **before base class** destructor



Order of calls to **constructors when instantiating a **DerivedC** object:**

```
BaseClass()  
DerivedA()  
DerivedB()  
DerivedC()
```

Order of calls to **destructors when instantiating a **DerivedC** object:**

```
~DerivedC()  
~DerivedB()  
~DerivedA()  
~BaseClass()
```

Basic Inheritance

No runtime cost

In memory `DerivedClass` is simply `BaseClass` with extra members tacked on the end

Basically saving to re-write `BaseClass` code

Address	1000	baseX	←- BaseClass members
	1004	baseY	
	1008	derX	←- DerivedClass -specific members
	1012	derY	

Project 2

Post subclasses (General, Promotional, Poll)

Derived class default constructor will **IMPLICITLY** call Base class constructor

Derived class parameterized constructor **MUST EXPLICITLY** call Base class constructor

Questions?

Recap



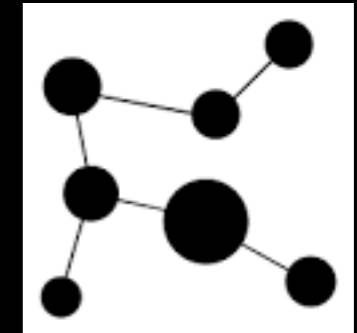
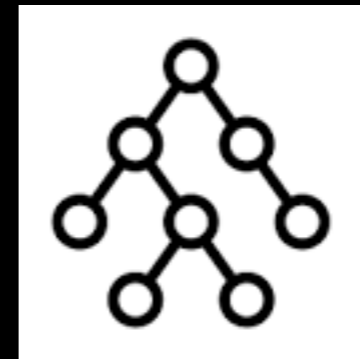
Recap

Useful C++ / OOP

Inheritance

Next Time

Abstract Data Types



Templates

