Inheritance

Tiziana Ligorio

Today's Plan

Recap

Useful C++ / OOP

Intro to Inheritance

Maybe More useful C++ / OOP

First a Recap and Review

OPP

Abstraction

Encapsulation

Information Hiding

Classes

Public Interface

Private Implementation

Constructors / Destructors

Interface

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

Implementation

```cpp
#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
```

```cpp
#include "SomeClass.hpp"

SomeClass::SomeClass()
{
    //implementation here

}


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

}


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

}


bool SomeClass::methodThree(int

someParameter)
{
    //implementation here

}
```

Review Some 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)
```

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

```cpp
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)
```

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

IS DIFFERENT FROM

```cpp
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
```

```
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:

```cpp
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

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

12

Operator Overloading

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

```cpp
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

```cpp
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

Type name (like `int`)

Possible values: like 0,1, 2, …

```
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

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

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

Basic Inheritance

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

    void setPaperSize(int size);
    void setOrientation(const string& orientation);
    void printDocument(const string& document);
private:
    // stuff here
}; //end Printer
```
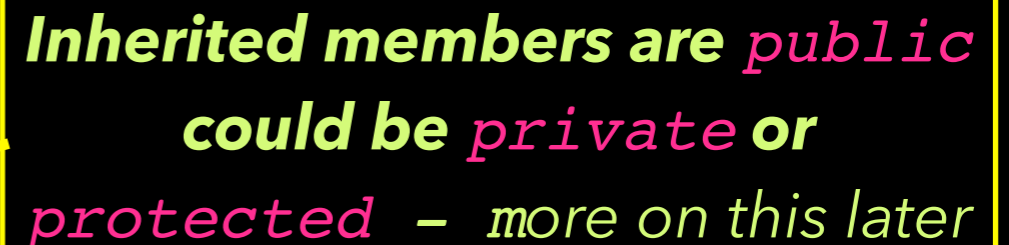
```cpp
class BatchPrinter
{
public:
    //Constructor, destructor
    void addDocument(const string& document);
    void printAllDocuments();
private:
    vector<string> documents;
}; //end BatchPrinter
```

19

Basic Inheritance

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

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


class BatchPrinter: public Printer   // inherit from printer
{
public:
    //Constructor, destructor
    void addDocument(const string& document);
    void printAllDocuments();
private:
    vector<string> documents;
}; //end BatchPrinter
```
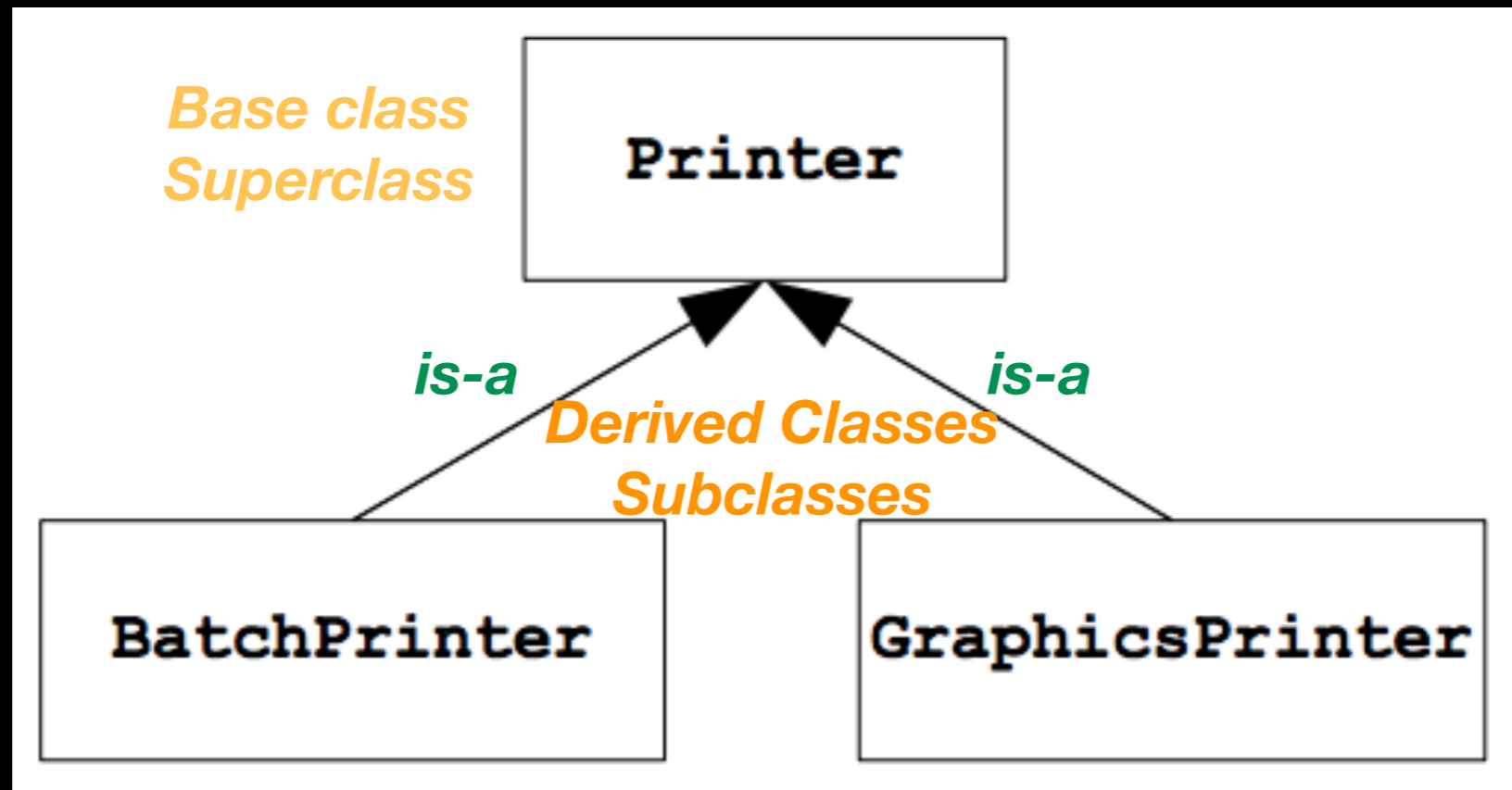
Inherited members are public could be private or protected - more on this later

Basic Inheritance

Base class
Superclass

Printer

is-a   *is-a*

Derived Classes
Subclasses

BatchPrinter   GraphicsPrinter

```
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

Overloading vs Overriding

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

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

<u>Overriding:</u> Rewrite function with *same signature* in derived class

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

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

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

```cpp
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
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 direct access to base class `private` members. However, can call public functions of the base class, which in turn do have access base classe's 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

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

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

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

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

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

main()

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

Constructors

```
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
```

29

Constructors

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


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

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


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

IMPLEMENTATION

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

}
main()
```

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

```cpp
DerivedClass my_derived_class;  ⬅
//PROBLEM!!! there is no default constructor to be called
//for BaseClass
```

30

Constructors

INTERFACE

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

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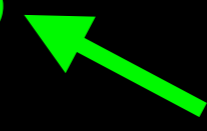
```
class DerivedClass: public BaseClass
{
public:
    DerivedClass();
    //stuff here

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

IMPLEMENTATION

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

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

Fix

main()

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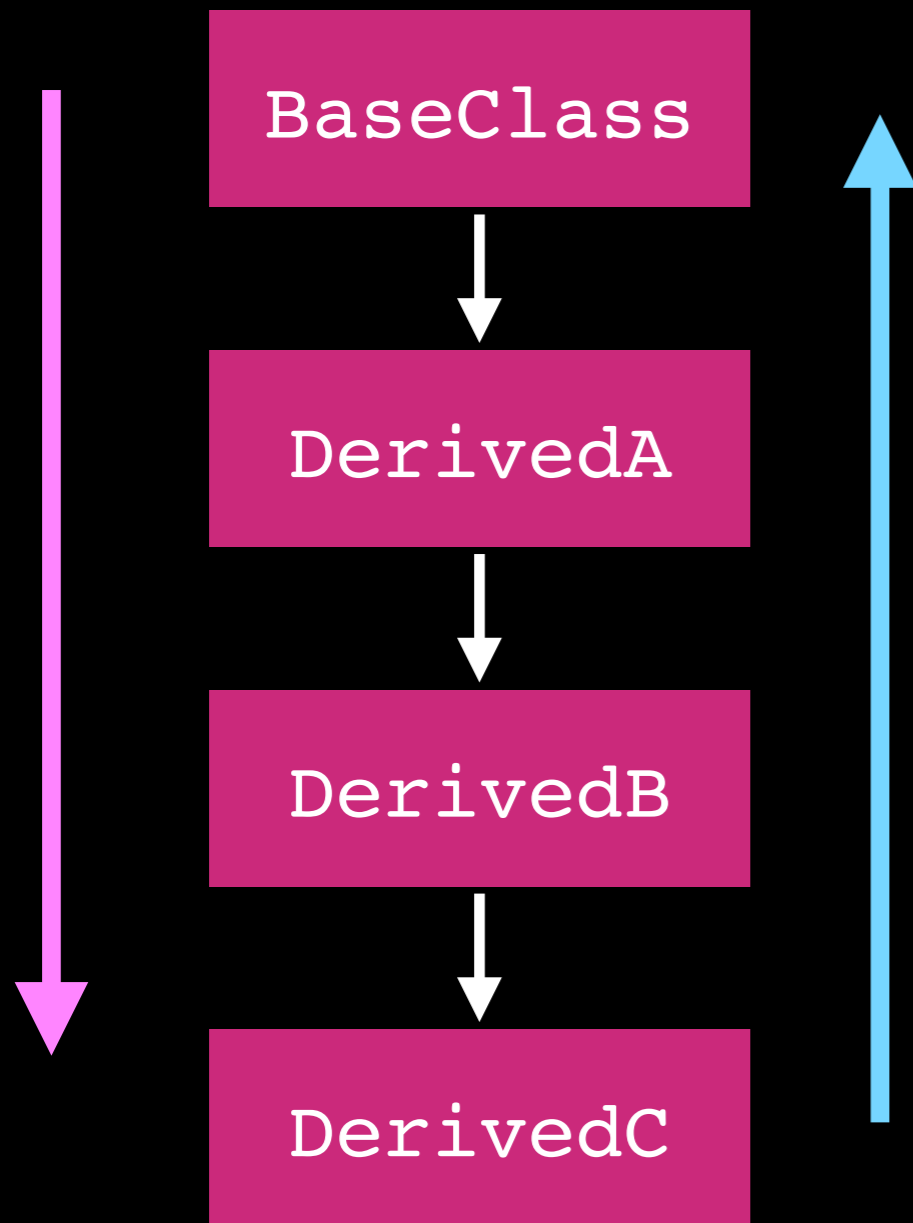
Destructors

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