C++ Software Engineering

for engineers of other disciplines

Module 2

"C++ OOP"

1st Lecture: Classy OOP

ALTEN

Spring 2022

Gothenburg, Sweden

Object Oriented Programming

- OOP is a programming paradigm introduced in the 60s.
- *Result of an effort to conceptualise real-world systems:*
  - Based on notation of **objects**
  - Objects are combination of variables, functions, and data structure
- OOP Main Features (Principles):
  - Objects & Classes
  - Encapsulation & Abstraction
  - Composition, Inheritance & Delegation
  - Polymorphism
  - Class-based vs Prototype-based
  - Open Recursion
  - Dynamic Dispatch & Message Passing

"Programming paradigms are a way to classify programming languages […] some paradigms are concerned mainly with implications for the execution model of the language […] other paradigms are concerned mainly with the way that code is organized, […] others are concerned mainly with the style of syntax and grammar." https://en.wikipedia.org/wiki/Programming_paradigm

- C++ is a multi paradigm programming language, and it supports OOP to a great degree, but it is not considered a pure OO programming language: https://www.quora.com/Why-C++-is-not-considered-as-pure-Object-Oriented-programming
- OOP is mainly focused on how the code is organized: *grouping the code into units along with the state that is modified by the code.*

Object Oriented Programming

- OOP Main Features (Principles):
  - Objects & Classes
  - Encapsulation & Abstraction
  - Composition, Inheritance & Delegation
  - Polymorphism
  - Class-based vs Prototype-based
  - Open Recursion
  - Dynamic Dispatch & Message Passing

Object & Classes

- **Object**: a self-contained entity that contains attributes (data) and behavior (logic), and nothing more.

- **Class**: defines the structure and behavior of an object.
- Check [Difference between Class and Object]
- Check [Structure vs class in C++]
- Check [Static Members of a C++ Class]

Class Person		Object1	Object2
Attributes	`std::string Name;`	Mahshid	David
	`unsigned char Age;`	47	30
	`enum Gender;`	Female	Male
	`unsigned char Weight;`	55	80
Methods	`void gainWeight();`	Each object has "its own" methods	
	`void looseWeight();`		

data

logic

"A well-written object:
- *Has well-defined boundaries*
- *Performs a finite set of activities*
- *Knows only about its data and any other objects that it needs to accomplish its activities*

In essence, an object is a discrete entity that has only the necessary dependencies on other objects to perform its tasks." https://developer.ibm.com/technologies/java/tutorials/j-introtojava1

- A **`class`**, like a **`struct`**, could contain both member variables and member function. Each instant has its own *"copy"* of the information declared in the class.

Object & Classes


```cpp
#include<iostream>
enum Gender_t {Male,Female,Other};
class Person {
//Attributes
    std::string Name;
    unsigned char Age;
    Gender_t Gender;
//Methods
    void gainWeight();
    void looseWeight();
};
void SomeFunction() {
    Person MahshidOBJ;
    Person *DavidOBJ = new Person;

    MahshidOBJ.Gender = Gender_t::Female;
    DavidOBJ->
}
```

Age
gainWeight
Gender
looseWeight
Name

Class Person		Object1	Object2
Attributes	`std::string Name;`	Mahshid	David
	`unsigned char Age;`	47	30
	`enum Gender;`	Female	Male
	`unsigned char Weight;`	55	80
Methods	`void gainWeight();`	Each object has "its own" methods	
	`void looseWeight();`		

data: Attributes

logic: Methods

- A **class** is a compound datatype and can be used like other datatypes. Similar access operator as **struct** and **union** are used to access members of an object, depending on the memory the object is instantiated in: '**.**' for static and automatic memory, and '**->**' for dynamic memory.
- Any instantiation of a **class** creates an object of that **class** which contains a *"copy"* of the declared members for itself.

Instantiation & Destruction – In Theory

- Every `class` has two ***special*** methods:
  - *Constructor*: a method with the same name of the class. *Constructors* can be overloaded; providing the different possibilities for object instantiation.
  - *Destructor:* a method with the same name as the class with a prepending `` `~` ``. Destructors cannot be overloaded, and they do not accept any input arguments.
- Upon object initialization of an object an "*appropriate*" constructor is invoked.
- Upon object destruction, the destructor of the object is called.

```
class Person
{
  // Called upon instantiation
  Person();
  Person(/* args */);
  // Called upon destrcution
  ~Person();
};
```

- If no constructor is provided for the class, the compiler includes a *default constructor.*
- *Default constructors* are those which could be called without any arguments i.e. constructor does not receive any arguments or there is a default value for each input to the constructor.

Instantiation & Destruction – In Action

foo.h > ...

```cpp
#include<iostream>
class Foo {
//Attributes
public:
    std::string Name;
private:
    unsigned char *Buffer;
//Methods
public:
    Foo(std::string _name, size_t _size);
    ~Foo();
private:
    Foo();
};
```

foo.cpp > ...

```cpp
#include "foo.h"
Foo::Foo() {
    this->Buffer = nullptr;
}
Foo::Foo(std::string _name, size_t _size) {
    this->Name = _name;
    this->Buffer = new unsigned char[_size];
}
Foo::~Foo() {
    if (this->Buffer != nullptr) {
        delete [] this->Buffer;
    }
}
```

- Class definitions usually reside in header files, and implementation of the class methods are written in the `.cpp` files.
- In order to define the method of the class, outside the class, the name of the class together with scope operator '`::`' is used.
- `this` is a pointer to the very instance running the code – using it to access member from within the class is optional, as all the member of the class are visible from any point within the class.

Object initialization and constructor destructor

foo.h > ...

```cpp
#include<iostream>
class Foo {
//Attributes
public:
    std::string Name;
private:
    unsigned char *Buffer;
//Methods
public:
    Foo(std::string _name, size_t _size);
    ~Foo();
private:
    Foo();
};
```

foo.cpp > ...

```cpp
#include "foo.h"
Foo::Foo() {
    this->Buffer = nullptr;
}
Foo::Foo(std::string _name, size_t _size) {
    this->Name = _name;
    this->Buffer = new unsigned char[_size];
}
Foo::~Foo() {
    if (this->Buffer != nullptr) {
        delete [] this->Buffer;
    }
}
```

- `private` methods cannot be called from outside the `class`.

```
foo.cpp: In function ‘int main()’:
foo.cpp:3:9: error: ‘Foo::Foo()’ is private within this context
    3 |     Foo bar;
      |         ^~~
```

```cpp
#include "foo.h"
int main() {
    Foo bar;
    return 0;
}
```

Object initialization and constructor destructor

foo.h > ...

```cpp
#include<iostream>
class Foo {
//Attributes
public:
    std::string Name;
private:
    unsigned char *Buffer;
//Methods
public:
    Foo(std::string _name, size_t _size);
    ~Foo();
private:
    Foo();
};
```

foo.cpp > ...

```cpp
#include "foo.h"
Foo::Foo() {
    this->Buffer = nullptr;
}
Foo::Foo(std::string _name, size_t _size) {
    this->Name = _name;
    this->Buffer = new unsigned char[_size];
}
Foo::~Foo() {
    if (this->Buffer != nullptr) {
        delete [] this->Buffer;
    }
}
```

- Depending on how the object is instantiated, appropriate constructor is called – overloading constructors provides the possibility of initializing different objects upon instantiation.

```cpp
#include "foo.h"
int main() {
    Foo bar("TEST",1);
    return 0;
}
```

Object initialization and constructor destructor

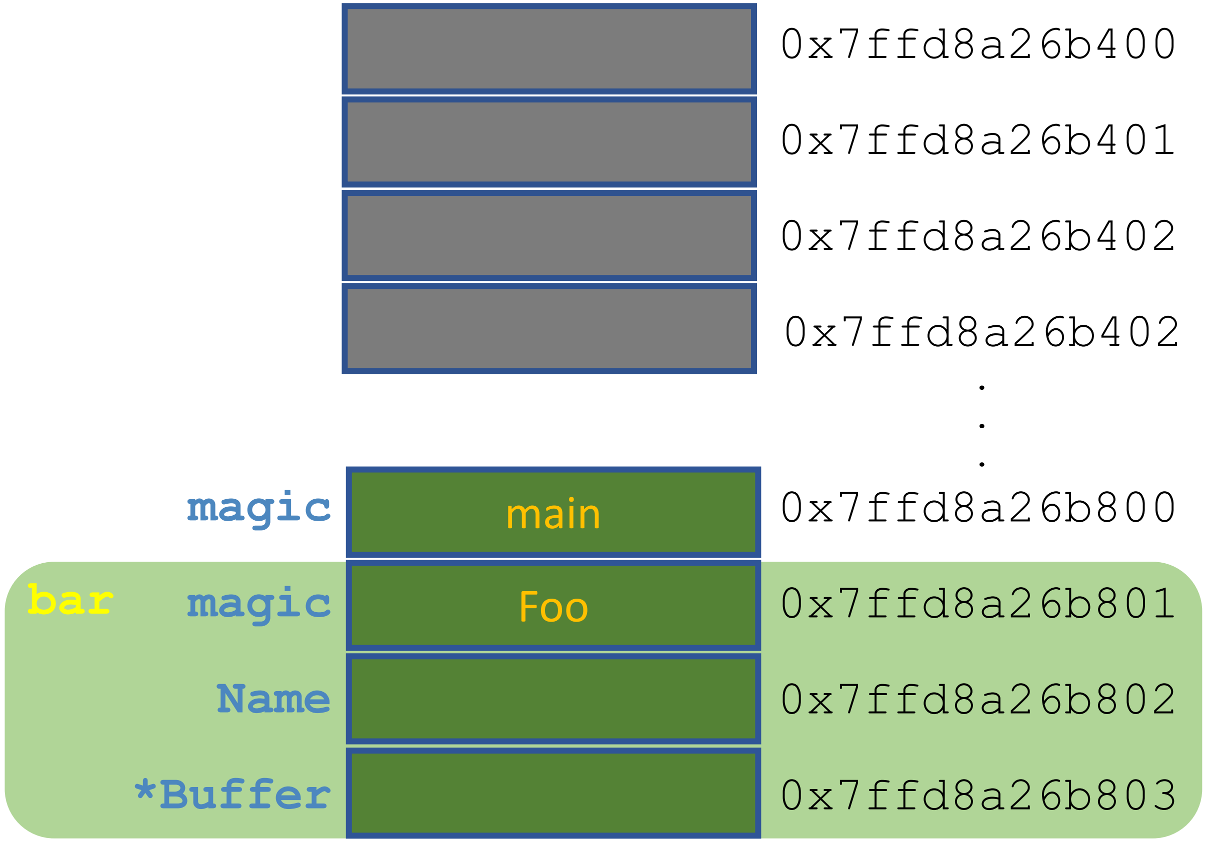
Illustrations in this lecture are simplified and they do not resemble the actual reality. The details would be explained in later modules of this course.

foo.cpp > ...

```cpp
#include "foo.h"
Foo::Foo() {
    this->Buffer = nullptr;
}
Foo::Foo(std::string _name, size_t _size) {
    this->Name = _name;
    this->Buffer = new unsigned char[_size];
}
Foo::~Foo() {
    if (this->Buffer != nullptr) {
        delete [] this->Buffer;
    }
}
```

```cpp
#include "foo.h"
int main() {
    Foo bar("TEST",1);
    return 0;
}
```

Object initialization and constructor destructor



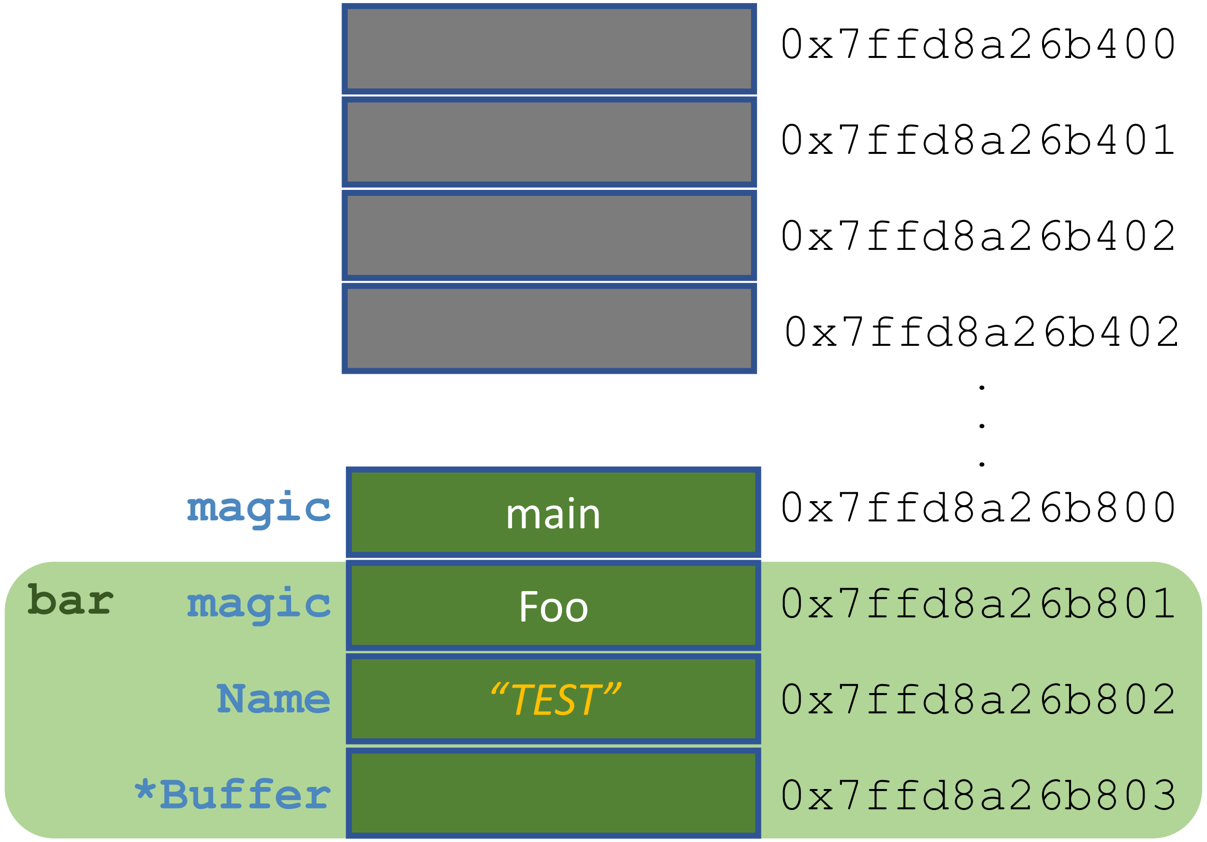
foo.cpp > ...

```cpp
#include "foo.h"
Foo::Foo() {
    this->Buffer = nullptr;
}
Foo::Foo(std::string _name, size_t _size) {
    this->Name = _name;
    this->Buffer = new unsigned char[_size];
}
Foo::~Foo() {
    if (this->Buffer != nullptr) {
        delete [] this->Buffer;
    }
}
```

```cpp
#include "foo.h"
int main() {
    Foo bar("TEST",1);
    return 0;
}
```

- Once an object is loaded into memory, necessary information such as address to its functions, are loaded into ram as well. The whole region allocated for an object in the above schematic is abstracted out to only illustrate the necessary fields: **`Name`** and **`*Buffer`**; and put the rest in the **`magic`** memory cell for the object.

Object initialization and constructor destructor

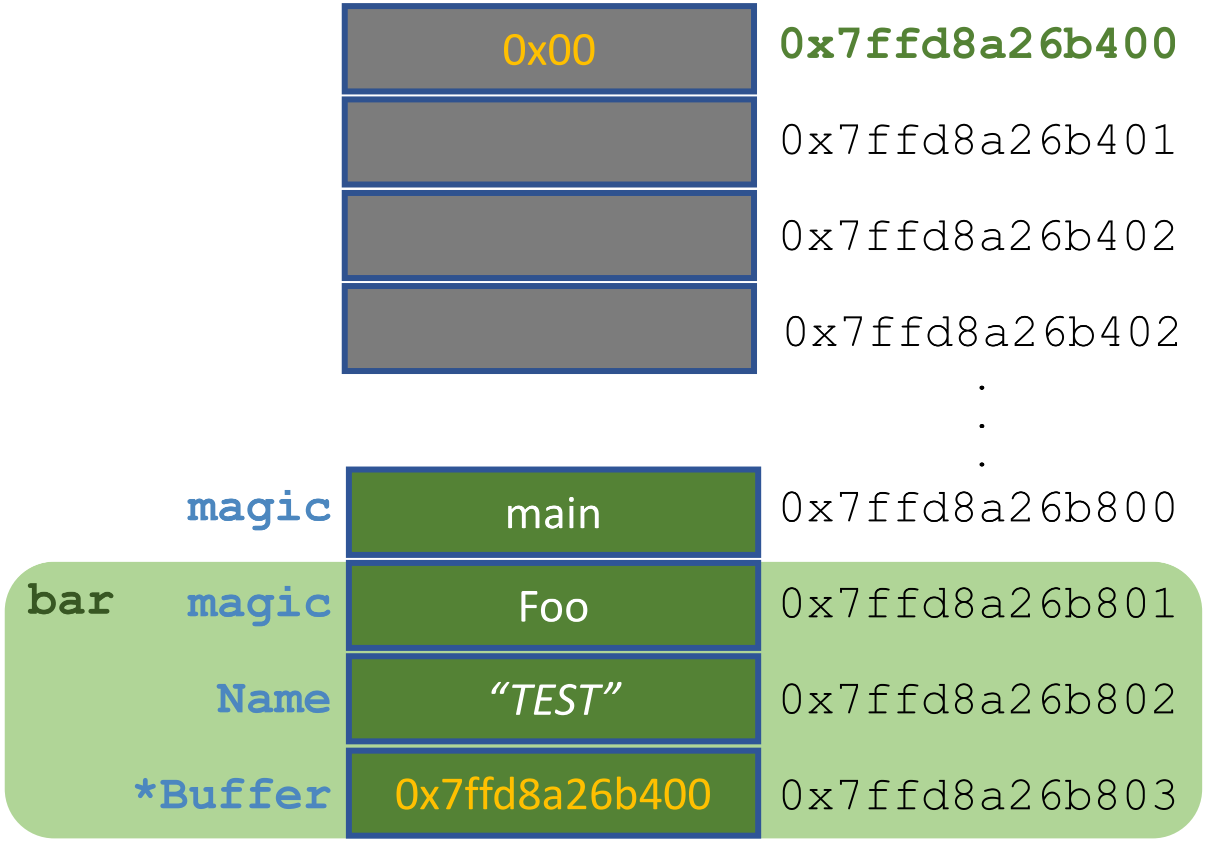


foo.cpp > ...

```cpp
#include "foo.h"
Foo::Foo() {
    this->Buffer = nullptr;
}
Foo::Foo(std::string _name, size_t _size) {
    this->Name = _name;
    this->Buffer = new unsigned char[_size];
}
Foo::~Foo() {
    if (this->Buffer != nullptr) {
        delete [] this->Buffer;
    }
}
```

```cpp
#include "foo.h"
int main() {
    Foo bar("TEST",1);
    return 0;
}
```

Object initialization and constructor destructor

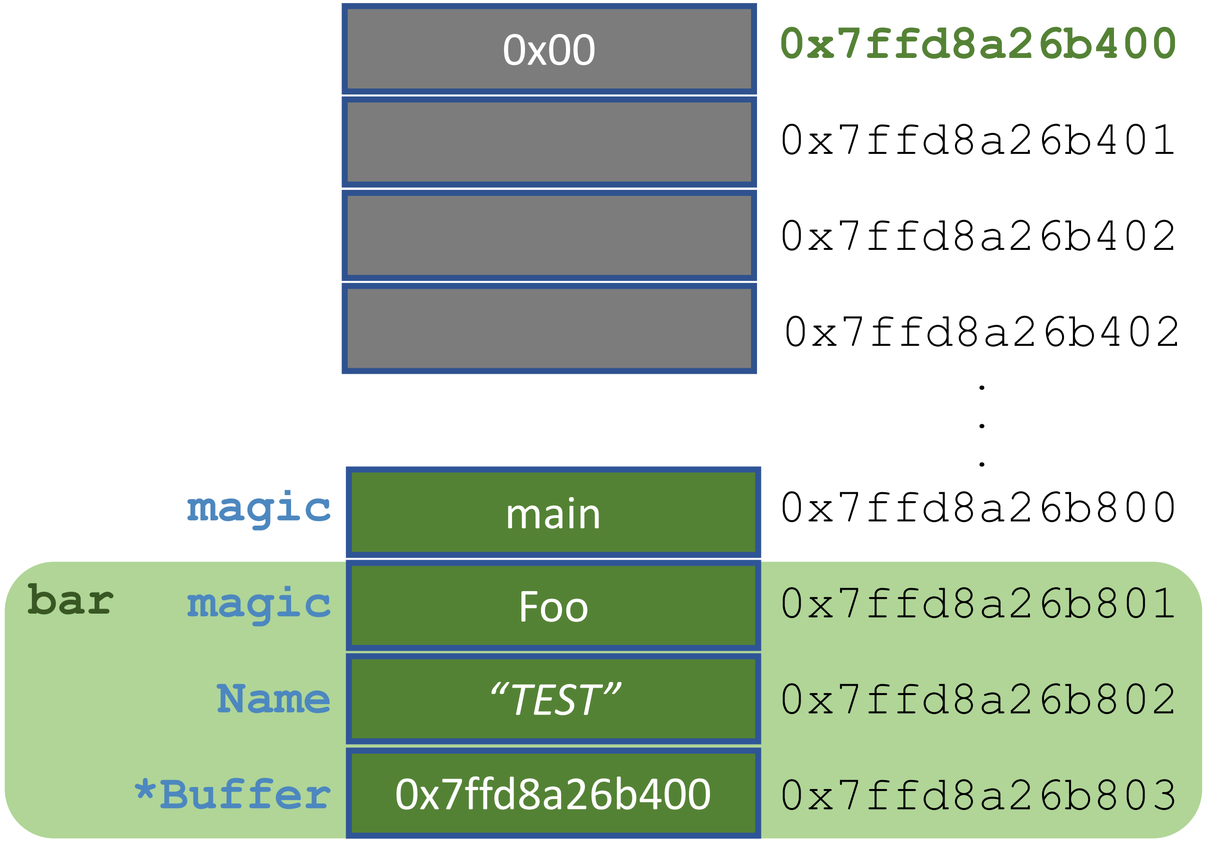


foo.cpp > ...

```cpp
#include "foo.h"
Foo::Foo() {
    this->Buffer = nullptr;
}
Foo::Foo(std::string _name, size_t _size) {
    this->Name = _name;
    this->Buffer = new unsigned char[_size];
}
Foo::~Foo() {
    if (this->Buffer != nullptr) {
        delete [] this->Buffer;
    }
}
```

```cpp
#include "foo.h"
int main() {
    Foo bar("TEST",1);
    return 0;
}
```

Object initialization and constructor destructor



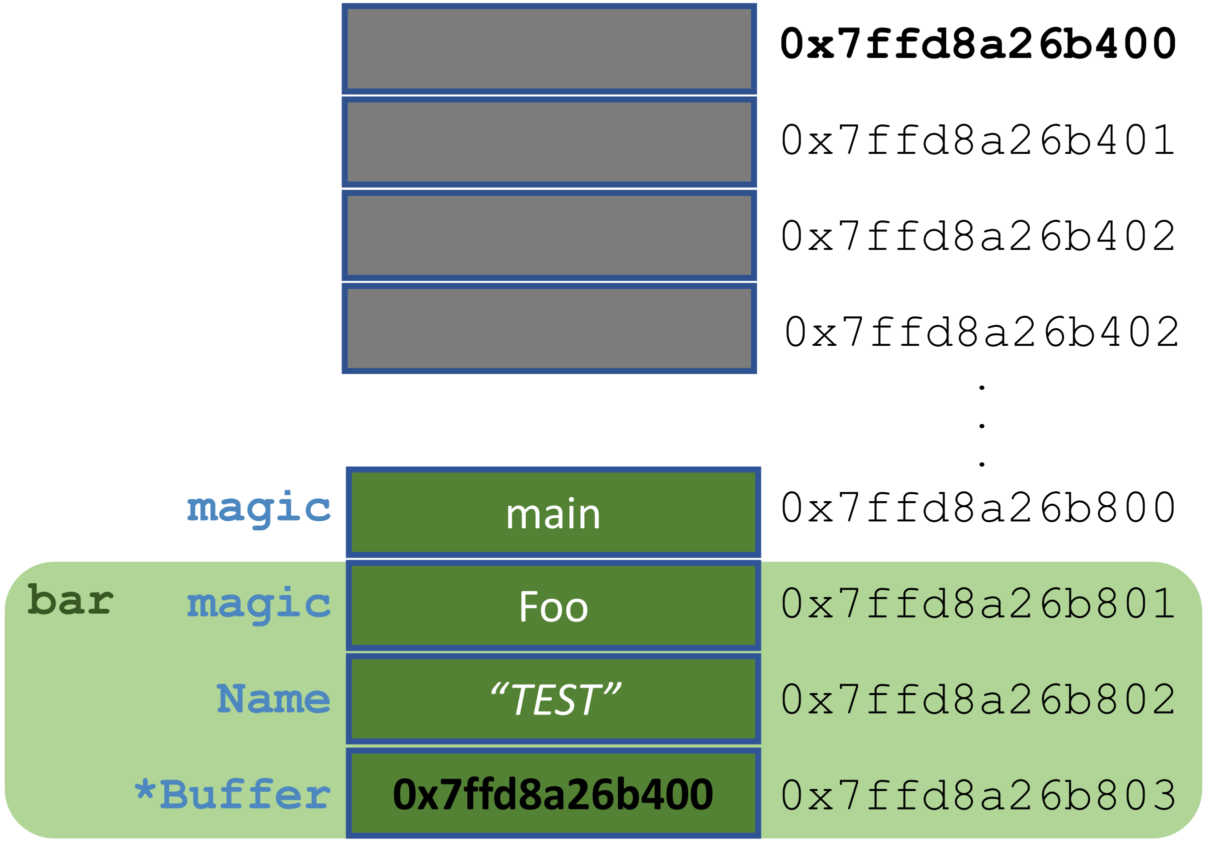
- Once the execution reaches the end of a function, the function’s frame is going to be removed from the *call stack*. Destructor for all the objects which are going out of scope would be invoked automatically.

foo.cpp > ...

```cpp
#include "foo.h"
Foo::Foo() {
    this->Buffer = nullptr;
}
Foo::Foo(std::string _name, size_t _size) {
    this->Name = _name;
    this->Buffer = new unsigned char[_size];
}
Foo::~Foo() {
    if (this->Buffer != nullptr) {
        delete [] this->Buffer;
    }
}
```

```cpp
#include "foo.h"
int main() {
    Foo bar("TEST",1);
    return 0;
}
```

Object initialization and constructor destructor



- *Dynamic memory* clean up is the developer's responsibility.

foo.cpp > ...

```cpp
#include "foo.h"
Foo::Foo() {
    this->Buffer = nullptr;
}
Foo::Foo(std::string _name, size_t _size) {
    this->Name = _name;
    this->Buffer = new unsigned char[_size];
}
Foo::~Foo() {
    if (this->Buffer != nullptr) {
        delete [] this->Buffer;
    }
}
```

```cpp
#include "foo.h"
int main() {
    Foo bar("TEST",1);
    return 0;
}
```

Object initialization and constructor destructor

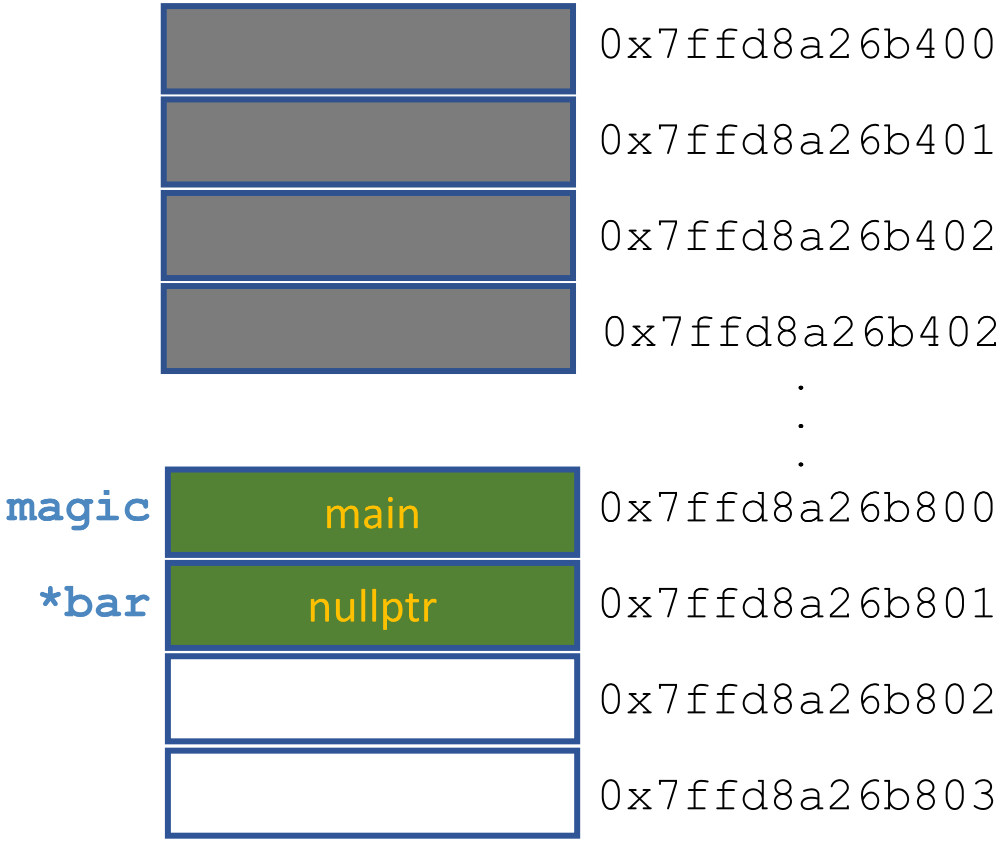
foo.cpp > ...

```cpp
#include "foo.h"
Foo::Foo() {
    this->Buffer = nullptr;
}
Foo::Foo(std::string _name, size_t _size) {
    this->Name = _name;
    this->Buffer = new unsigned char[_size];
}
Foo::~Foo() {
    if (this->Buffer != nullptr) {
        delete [] this->Buffer;
    }
}
```

```cpp
#include "foo.h"
int main() {
    Foo bar("TEST",1);
    return 0;
}
```

- All the allocated memory blocks are *freed* upon *normal termination* of the program.

Object initialization and constructor destructor

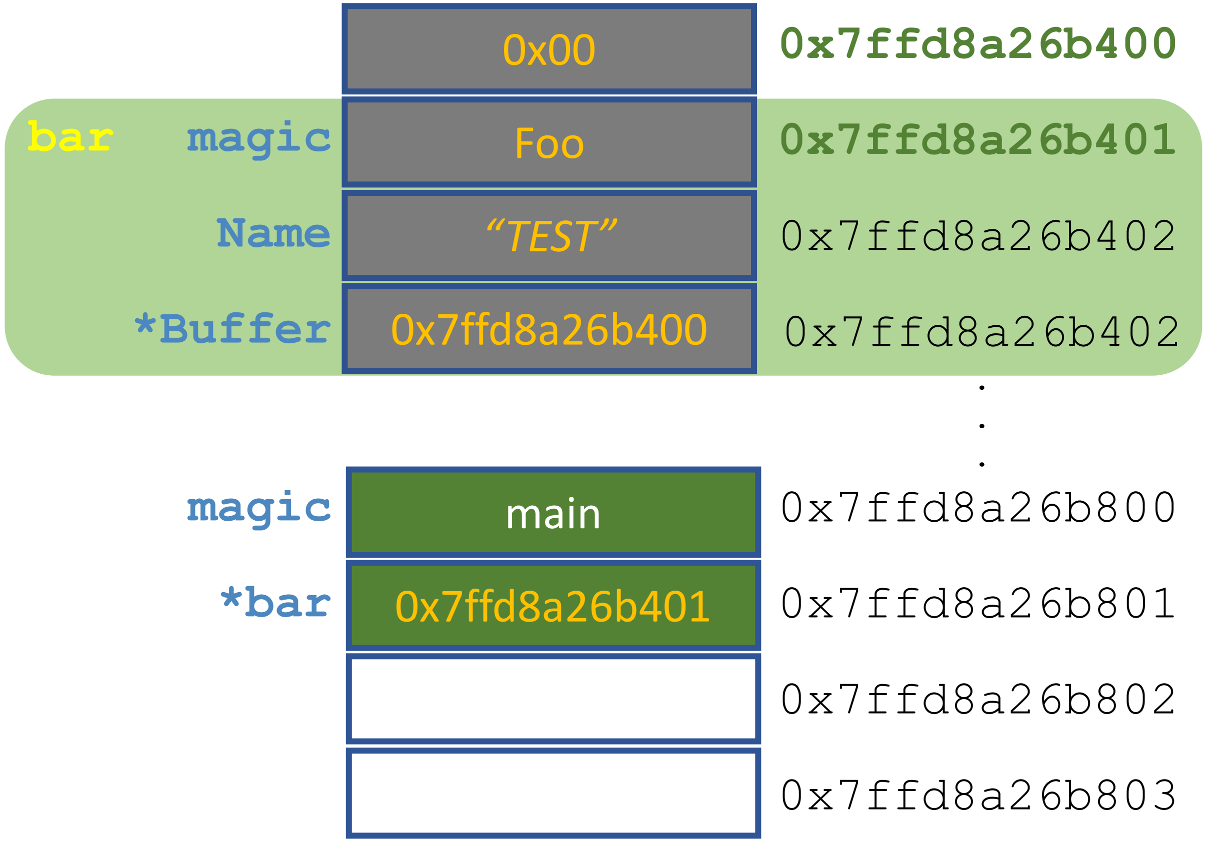


foo.cpp > ...

```cpp
#include "foo.h"
Foo::Foo() {
    this->Buffer = nullptr;
}
Foo::Foo(std::string _name, size_t _size) {
    this->Name = _name;
    this->Buffer = new unsigned char[_size];
}
Foo::~Foo() {
    if (this->Buffer != nullptr) {
        delete [] this->Buffer;
    }
}
```

```cpp
#include "foo.h"
int main() {
    Foo *bar = new Foo("TEST",1);
    return 0;
}
```

Object initialization and constructor destructor



- Operator **new** invokes the constructor of the **class**. Objects instantiated this way are allocated in *dynamic memory*.

foo.cpp > ...

```cpp
#include "foo.h"
Foo::Foo() {
    this->Buffer = nullptr;
}
Foo::Foo(std::string _name, size_t _size) {
    this->Name = _name;
    this->Buffer = new unsigned char[_size];
}
Foo::~Foo() {
    if (this->Buffer != nullptr) {
        delete [] this->Buffer;
    }
}
```

```cpp
#include "foo.h"
int main() {
    Foo *bar = new Foo("TEST",1);
    return 0;
}
```

Object initialization and constructor destructor

ALTEN

bar magic — 0x7ffd8a26b400
0x7ffd8a26b401
Name — 0x7ffd8a26b402
*Buffer — 0x7ffd8a26b402

. . .

0x7ffd8a26b800
0x7ffd8a26b801
0x7ffd8a26b802
0x7ffd8a26b803

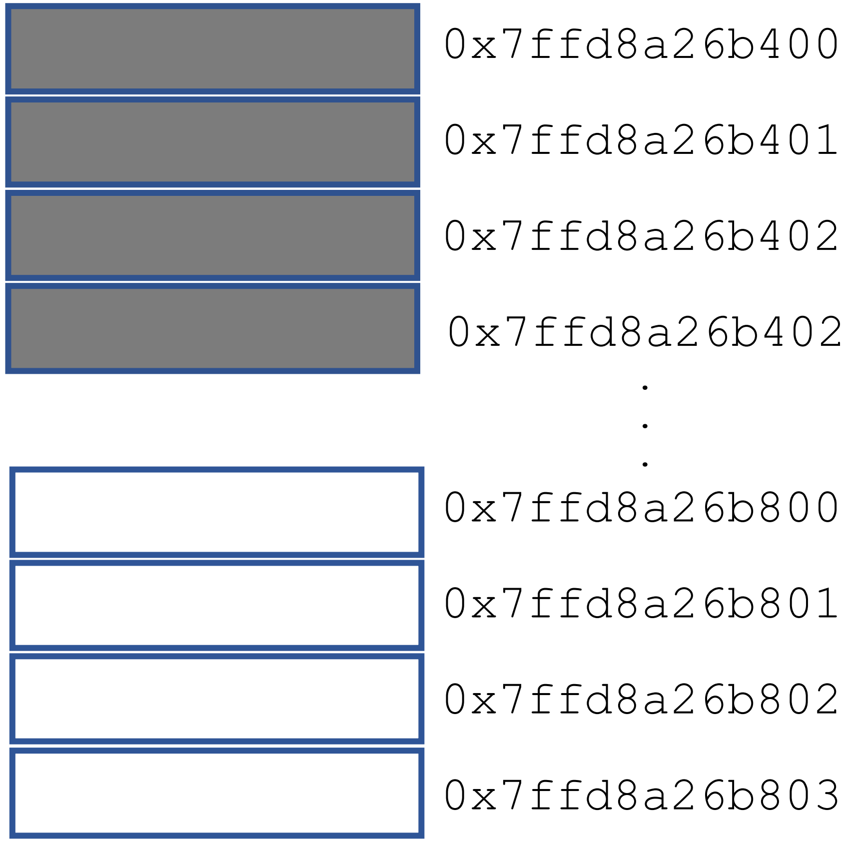
- Destructor for objects declared in *dynamic memory* ***would not be*** invoked once the pointer goes out of scope.

foo.cpp > ...

```cpp
#include "foo.h"
Foo::Foo() {
    this->Buffer = nullptr;
}
Foo::Foo(std::string _name, size_t _size) {
    this->Name = _name;
    this->Buffer = new unsigned char[_size];
}
Foo::~Foo() {
    if (this->Buffer != nullptr) {
        delete [] this->Buffer;
    }
}
```

```cpp
#include "foo.h"
int main() {
    Foo *bar = new Foo("TEST",1);
    return 0;
}
```

Object initialization and constructor destructor



foo.cpp > ...

```cpp
#include "foo.h"
Foo::Foo() {
    this->Buffer = nullptr;
}
Foo::Foo(std::string _name, size_t _size) {
    this->Name = _name;
    this->Buffer = new unsigned char[_size];
}
Foo::~Foo() {
    if (this->Buffer != nullptr) {
        delete [] this->Buffer;
    }
}
```

```cpp
#include "foo.h"
int main() {
    Foo *bar = new Foo("TEST",1);
    // SOME CODE
    delete bar;
    return 0;
}
```

- Destructor for objects declared in *dynamic memory* ***would be*** invoked if the pointer is **delete**d.

Object Oriented Programming

- OOP Main Features (Principles):
  - Objects & Classes
  - Encapsulation & Abstraction
  - Composition, Inheritance & Delegation
  - Polymorphism
  - Class-based vs Prototype-based
  - Open Recursion
  - Dynamic Dispatch & Message Passing

Encapsulation

- Class Member Functions
- A technique to employ ***Information Hiding*** principle.
- Encapsulation binds data and function that manipulate it together.
- C++ provides access specifers for this purpose:

"Information hiding is the ability to prevent certain aspects of a class from being accessible [...] using programming language features." https://en.wikipedia.org/wiki/Information_hiding

Keyword	Description
`public`	Accessible to everyone who has access to the class
`private`	Accessible only to the members of a class and its friends.
`protected`	Accessible only to the members and friends of a class, and members (and friends until C++17) of derived classes.

- Default access specifier for members defined in a `class` is `private`.
- Encapsulation helps protecting data while reduces complexity by *hiding* unnecessary information.

```
#include<iostream>
enum Gender_t {Male,Female,Other};
class Person {
//Attributes
public:
    std::string Name;
    unsigned char Age;
    Gender_t Gender;
private:
    unsigned charWeight;
//Methods
public:
    void gainWeight();
    void looseWeight();
};
```

Object Oriented Programming

- OOP Main Features (Principles):
  - Objects & Classes
  - Encapsulation & Abstraction
  - Composition, Inheritance & Delegation
  - Polymorphism
  - Class-based vs Prototype-based
  - Open Recursion
  - Dynamic Dispatch & Message Passing

Abstraction vs. Encapsulation

- Abstraction is another form of *information hiding*.
- Check Abstraction vs. Encapsulation in more detail.

Encapsulation	Abstraction
Process to *contain* information	Process to *gain* information
Performed at implementation level	Performed at design level
Hides data from *outsiders*	Hides *background* details
Implemented using access modifiers: `public`/`private`/`protected`	Implemented "*mostly*" using abstract and interface classes

- *"Getter/Setter"*s are the classic method of encapsulation. The data is *encapsulated* from *outside* the class, and only means of *interaction* with the data is through the *"Getter/Setter"* methods a.k.a. *interface.*

```
class Person
{
  public:
    void          set_age(unsigned char age) {this->Age = age;}
    unsigned char get_age(unsigned char age) {return this->Age;}
  private:
    unsigned char Age;
};
```

Abstraction vs. Encapsulation


```cpp
class Abstraction {
private:
    int x,y,z;
public:
    Abstraction () = delete;
    Abstraction (int _x,int _y,int _z):x(_x),y(_y),z(_z) { }
    int sum() { return x+y+z; }
};
```

```cpp
class Encapsulation {
private:
    int x,y,z;
public:
    Encapsulation() = default;
    void SetX(int _x) { x = _x; }
    void SetY(int _y) { y = _y; }
    void SetZ(int _z) { z = _z; }
    int X() { return x; }
    int Y() { return y; }
    int Z() { return z; }
};
```

Exercises !

Kristen is a contender for valedictorian of her high school. She wants to know how many students (if any) have scored higher than her in the exams given during this semester.

Create a class named with the following specifications:
•An instance variable named to hold a student's exam scores.
•A *void input()* function that reads integers and saves them to .
•An *int calculateTotalScore()* function that returns the sum of the student's scores.

Reference : [HackerRank]

Object Oriented Programming

- OOP Main Features (Principles):
  - Objects & Classes
  - Encapsulation & Abstraction
  - Composition, Inheritance & Delegation
  - Polymorphism
  - Class-based vs Prototype-based
  - Open Recursion
  - Dynamic Dispatch & Message Passing

Object Composition

> *"In computer science, object composition is a way to combine objects or data types into more complex ones."* https://en.wikipedia.org/wiki/Object_composition

```
#include<iostream>
enum Gender_t {Male,Female,Other};
class Behaviour;
class Health;
/* more */
class Person {
//Attributes
public:
    std::string Name;
    unsigned char Age;
    Gender_t Gender;
    Behaviour PersonalBehaviour;
    Health PersonalHealth;
      ....
```

```
typedef std::__cxx11::basic_string<char> std::string
```

Object Oriented Programming

- OOP Main Features (Principles):
  - Objects & Classes
  - Encapsulation & Abstraction
  - Composition, Inheritance & Delegation
  - Polymorphism
  - Class-based vs Prototype-based
  - Open Recursion
  - Dynamic Dispatch & Message Passing

Inheritance

- A technique to extend *characteristic* of a class: A *derived* class *inherits* **ALL** members of the *base* class.

```
class DerivedClassName: AccessSpecifier BaseClassName
```

- Access specifier, specifies the access level of the *inherited* members.

Keyword	Description
`public`	Inherited members keep the same access specifier from the *base* class.
`private`	All `public` and `protected` inherited members become `private`.
`protected`	All `public` and `protected` inherited members become `protected`.

- Default inheritance is `private` for classes, if not specified.
- Members declared as `protected` are accessible to derived classes, but not others.
- Inheritance access specifier only makes sense for `public` and `protected` members, since `private` members are not accessible to derived classes.

```
#include <iostream>
class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
};

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
};
```

Inheritance

- Derived class inherits the **`private`** members of the *base* class yet **cannot** access them i.e. **`private`** members are not accessible from within derived classes.

- Members declared **`protected`** are not accessible to public.

Keyword	Description For Specifying Inheritance Access Level
`public`	Inherited members keep the same access specifier from the *base* class.
`private`	All **`public`** and **`protected`** inherited members become **`private`**.
`protected`	All **`public`** and **`protected`** inherited members become **`protected`**.
Keyword	**Description For Specifying Member Access Level**
`public`	Accessible to everyone who has access to the class
`private`	Accessible only to the members of a class and its friends.
`protected`	Accessible only to the members and friends of a class, and members (and friends until C++17) of derived classes.

```
class Base {
  int x;
protected:
  int y;
public:
  int z;
  void increaseX() {this->x++;}
}
class Derived : public Base {
 void testAccess() {
   this->z = 0; // OK
   this->y = 0; // OK
   this->x = 0; // NOT OK
 }
};
void SomeFunction() {
  Base b;
  b.x = 0; // NOT OK
  b.y = 0; // NOT OK
  b.z = 0; // OK
}
```

Inheritance


```cpp
class Polygon {
private:
    int width, height;
protected:
    void set_values (int a, int b) { width=a; height=b;}
    int Width() {return this->width;}
    int Height() {return this->height;}
public:
    Polygon (int a, int b) { width=a; height=b;}
    void print() {
        std::cout << this->width << " " << this->height << std::endl;
    }
};
```

```cpp
class Square: public Rectangle {
    friend void modifySquare(Square&);
public:
    Square(int a) : Rectangle(a,a) {}
};
```

```cpp
class Rectangle: /*protected*/ Polygon {
    friend void modifyRectangle(Rectangle&);
public:
    Rectangle(int a,int b) : Polygon(a,b) {}
private:
    int area () { return Width() * Height(); }
};
```

```cpp
void modifySquare(Square &s) {
    std::cout << "Modifying Rectangle with Height and Width:";
    s.print();
    std::cout << "Hence the area is: " << s.area() << std::endl;

    s.set_values(s.Width()*2,s.Height()*2);

    std::cout << "After modification new Height and Width:";
    s.print();
    std::cout << "Hence the area is: " << s.area() << std::endl
}
```

Friendship

- Friends can access **private** members of the class they are "*friend*" with.
- Check This pointer
- A **class** could have a function as a **friend**.
- A **class** could have a class as a **friend**.

Keyword	Description For Specifying Inheritance Access Level
public	Inherited members keep the same access specifier from the *base* class.
private	All **public** and **protected** inherited members become **private**.
protected	All **public** and **protected** inherited members become **protected**.
Keyword	**Description For Specifying Member Access Level**
public	Accessible to everyone who has access to the class
private	Accessible only to the members of a class and its friends.
protected	Accessible only to the members and friends of a class, and members (and friends until C++17) of derived classes.

```
class Bar;

class Foo {
  int x;
public:
  friend void printFoo(const Foo&);
  void printBar(const Bar &_b) {
    std::cout << _b.x << std::endl;
  }
};

class Bar {
  friend class Foo;
  int x;
};

void printFoo(const Foo &_foo) {
  std::cout << _foo.x << std::endl;
}
```

Friendship


```cpp
#include <iostream>

class Bar;
class Foo {
    int x = -1;
    friend void externalPrint(const Foo&);
public:
    void printBar(const Bar &_b); /*{
        std::cout << _b.x << std::endl;//F
    }*/
    Foo() = default;

};
class Bar {
    friend class Foo;
    int x  = 2;
public:
    Bar() = default;
};
```

```cpp
#include "acc.h"

void externalPrint(const Foo &_f) {
    std::cout << "External print, printing FOO: "<< _f.x << std::endl;
}

void Foo::printBar(const Bar &_b) {
    std::cout << "Printing BAR from within FOO: "<< _b.x << std::endl;
}

int main() {
    Foo f;
    Bar b;
    externalPrint(f);
    f.printBar(b);
    return 0;
}
```

Diamond & Virtual


```cpp
class CatDog : public Cat, public Dog {
public:
    CatDog() = default;
};

int main() {
    CatDog catDog;

    catDog.Woof();
    catDog.Meow();
    catDog.eat();
    //catDog.Weight = 3;

    return 0;
}
```

```cpp
class Animal {
public:
    Animal() = default;
    void eat() {
        std::cout << "Eating." << std::endl;
    }
    int Weight;
};
class Dog : public virtual Animal {
public:
    Dog() = default;
    void Woof() {
        std::cout << "WOOF!" << std::endl;
    }
};
class Cat : public virtual Animal {
public:
    Cat() = default;
    void Meow() {
        std::cout << "MEOW!" << std::endl;
    }
};
```

Exercises !

- [Rectangle Area]

Object Oriented Programming

- OOP Main Features (Principles):
  - Objects & Classes
  - Encapsulation & Abstraction
  - Composition, Inheritance & [Delegation]
  - Polymorphism
  - Class-based vs Prototype-based
  - Open Recursion
  - Dynamic Dispatch & Message Passing

- C++ does not have *out-of-the-box* delegation keyword, yet provides other means to achieve the same. We will cover some of them in later modules of this course.

Assignment

- A car service shop needs to keep track of the records of services they provide to their customer. Create a system for them so they could keep ATLEAST the below records:
    - Date the customer visited
    - Services performed (at least 2 different services)
    - Parts changed (at least 2 different parts)
    - Payment (method & amount)

- The purpose is to practice OO concepts we learnt – don't focus on features, put your time on defining objects.