Introductory Robot Programming

ENPM809Y

Lecture 07 – Object-oriented Programming (OOP) – Part I

Zeid Kootbally

zeidk@umd.edu

Fall 2020

Overview I

Overview II

Overview III

Highlights

- Two lectures on object-oriented programming.
- We will first learn about the concepts of class and object.
- We will see what is object oriented programming and the advantages this programming paradigm brings on the table.
- You will learn about inheritance and polymorphism in the next lecture.
- Before we start, create a new project and copy files from ELMS to this project.
  - ➤ CLion:Lecture7

Procedural-oriented Programming

- In procedural-oriented programming, emphasis is given to procedure calls (function calls in Python).
  - Functions: set of statements which performs a particular task.
    - Assignments.
    - Tests (selection statements).
    - Loops (iteration statements).
    - Definition of other functions.
    - Invocation of other functions.
    - . . .

Procedural-oriented Programming | Limitations

- Although procedural-oriented programs are extremely powerful, they do have some limitations.
- New changes to large programs are done with hacks.
- As programs get larger, they tend to turn into "spaghetti-code"[1] (unstructured source code).
  - more difficult to understand.
  - more difficult to maintain.
  - more difficult to debug.
  - more difficult to reuse.
  - fragile and easier to break.

[1]https://en.wikipedia.org/wiki/Spaghetti_code

Object-Oriented Programming (OOP)

- In traditional programming, programs are lists of instructions to the computer that define data and then work with that data (via statements and functions).
- Data and the functions that work on that data are separate entities that are combined together to produce the desired result.
- Because of this separation, traditional programming often does not provide a very intuitive representation of reality. Its up to the programmer to manage and connect the properties (variables) to the behaviors (functions) in an appropriate manner.

```
PickPart("fanuc_robot","red_part");
```

Object-Oriented Programming (OOP)

- What is object-oriented programming (OOP)?
  - If you take a look around you, you will spot many objects: books, chairs, tables, buildings, food, and even you (objects do not have to be physical objects).
  - Objects have two major components to them:
    1. A list of relevant properties (e.g., weight, color, size, solidity, shape, etc).
    2. Some number of behaviors that they can exhibit (e.g., move, fall, etc).
  - These properties and behaviors are inseparable.

Object-Oriented Programming (OOP)

- OOP provides us with the ability to create objects that tie together both properties and behaviors into a self-contained, reusable package.

```
fanuc_robot.PickPart("red_part");
```

  - This reads more clearly: You can tell who the subject is (a Fanuc robot) and what behavior is being invoked (picking a red part).
  - Rather than being focused on writing functions, we are focused on defining objects that have a well-defined set of behaviors.
  - This is why the paradigm is called object-oriented.

Object-Oriented Programming (OOP)

- Object-oriented programming (OOP) is developed by:
  1. retaining all the best features of procedural method.
     - e.g., functions.
  2. adding new concepts to these features to facilitate efficient programming.
     - e.g., classes and objects.
- OOP methods bring in many features that make possible an entirely new way of approaching a program.
- 4 pillars of OOP:
  - Abstraction.
  - Encapsulation.
  - Inheritance.
  - Polymorphism.

OOP | Advantages

- Modularity: OOP provides a clear modular structure for programs.
  - Good for defining abstract datatypes where implementation details are hidden and the unit has a clearly defined interface.
- Code reusability through inheritance: We can eliminate redundant code and extend the use of existing classes.
- Flexibility through Polymorphism: You can "shape-shift" methods created in the Base class to make them work with objects from Derived classes.
- Effective problem solving: OOP languages allow you to break down your software into bite-sized problems that you then can solve (one object at a time).

OOP | Limitations

- OOP is not a solution for everything.
    - It will not make bad code better.
    - Not suitable for all types of problems.
    - Not everything decomposes into classes.
- Learning curve.
    - Usually a steeper learning curve than procedural programming.
    - Many OO languages, many variations of OO concepts.
- Design.
    - More upfront design is necessary to create good models for your domain.

OOP | Classes

- A class in OOP is a template/blueprint that describes the behavior/state that objects of its type support.
- Classes are a way of grouping together related data (attributes) and functions (methods) which act upon that data.
- Can hide attributes and methods.
- Provides a public interface.
- Each class should be designed and programmed to accomplish one and only one thing.
    - Many classes are usually used to build an entire application.

OOP | Classes | Example

- Consider the following blueprint for a Saab vehicle [2]
- You can imagine this blueprint as being a class (SaabModel).

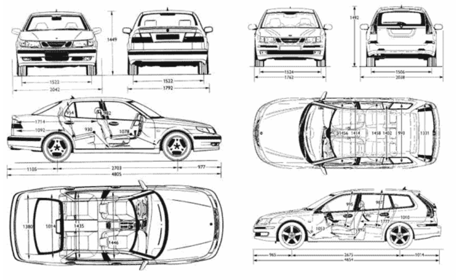
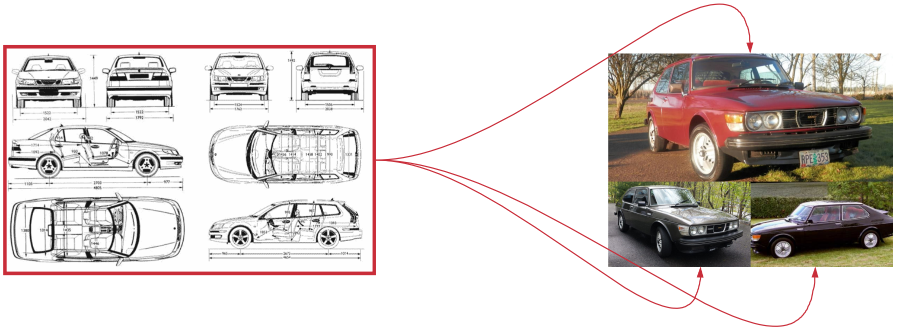


Figure: Saab model blueprint.

[2]https://www.saabplanet.com/saab-blueprints/

OOP | Objects

- From this blueprint you can manufacture many cars which can be seen as the objects of your class SaabModel.
- Each one of these cars is unique with its own ID (e.g., VIN number).



OOP | Objects

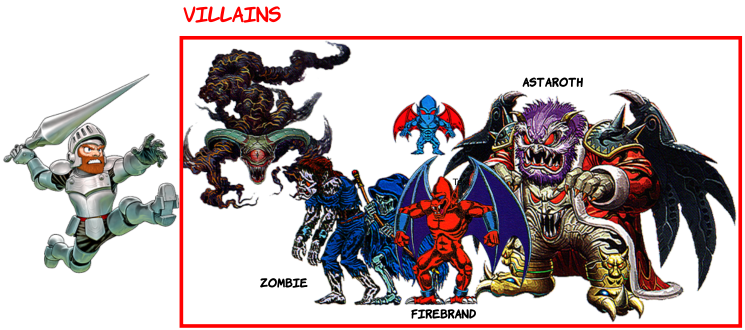
- An object is a specific instance of a class and each object has its own identity.
- A class is an "object factory" where multiple objects can be created from the same class.
- Each object has its own identity.
- Note: The term "object" is overloaded a bit, and this causes some amount of confusion. In traditional programming, an object is a piece of memory to store values. In object-oriented programming, an "object" implies that it is both an object in the traditional programming sense, and that it combines both properties and behaviors. From this point forward, when we use the term "object", we will be referring to "objects" in the object-oriented sense.

OOP | Domain

- You want to use OOP in your program? Does it make sense?
- When we design our own classes, we have to decide how we are going to group things together, and what our objects are going to represent.
- Sometimes we write objects which map very intuitively onto things in the real world.
    - For example, if we are writing code to simulate chemical reactions, we might have Atom objects which we can combine to make a Molecule object.
    - However, it is not always necessary, desirable or even possible to make all code objects perfectly analogous to their real-world counterparts.

OOP | Domain

- In this lecture we would like to model "villain" characters from a video game.
- Each villain will have a "name", a "health points", and a "number of lives".
- For now, each villain can only do one thing: "take damages". When a villain takes damages, its "health points" deplete. When "health points" is 0 then the villain loses one "life".



Classes

- In C++, OOP is typically done via the `class` keyword, which is used to define a new user-defined type called a class.
- It turns out that the C++ Standard Library is full of classes that have been created for your benefit, e.g., `std::string`, `std::vector`, and `std::array` are all class types.
- So when you create an object of any of these types, you are instantiating a class object. And when you call invoke a function using these objects, you are calling a member function.

```cpp
int main(){
    std::string s {"Hello, world!"}; //--instantiate a string class object
    std::array<int, 3> a {1, 2, 3}; //--instantiate an array class object
    std::vector<double> v {1.1, 2.2}; //--instantiate a vector class object

    std::cout << "length: " << s.length() << '\n'; //--call a method
}
```

Classes

- We will start with a simple `Villain` class and edit it as we learn new OOP concepts.

```
class Villain{//--start with capital letter
public:
    std::string name_;
    double health_;
    int life_;
    void TakeDamage(double damage){
        health_ -= damage;
    }
    void Print(){
        std::cout << "n="<< name_ << ", h=" << health_ << ", l=" << life_;
    }
};
int main() {
    Villain astaroth{"Astaroth", 100, 2};
    astaroth.life_ = 3;
    astaroth.Print();
}
```

- Note: Per Google C++ Style Guide, we use a trailing underscore for attributes.

Classes

- You can access members (data and functions) of a class with the dot operator.
    - For example, when we call `astaroth.Print()` we are telling the compiler to call the `Print()` method, associated with the `astaroth` object.
- In the code below, `name_`, `health_`, `life_` refer to the associated object. When we call `astaroth.Print()`, the compiler interprets `name_` as `astaroth.name_`, `health_` as `astaroth.health_`, and `life_` as `astaroth.life_`

```
void Print(){
    std::cout << "n="<< name_ << ", h=" << health_ << ", l=" << life_;
}
```

- The associated object is essentially implicitly passed to the method. For this reason, it is often called the implicit object.
- Key Point: With non-member functions, we have to pass data to the function to work with. With member functions (methods), we can assume we always have an implicit object of the class to work with.

Classes | Access Specifiers

- The following code will not compile. This is because by default, all members of a class are private. Private members are members of a class that can only be accessed by other members of the class. Because `main()` is not a member of `Villain`, it does not have access to `Villain` private members.

```
class Villain{//--start with capital letter
    std::string name_;
    double health_;
    int life_;
    void Print(){
        std::cout << "n="<< name_ << ", h=" << health_ << ", l=" << life_;
    }
};
int main() {
    Villain astaroth{"Astaroth", 100, 2};
    astaroth.life_ = 3;
    astaroth.Print();
}
```

- If you want class members to be accessible in `main()` you have to make them public.

Classes | Access Specifiers

- C++ provides 3 different access specifier keywords: `public`, `private`, and `protected`.
- `public` and `private` are used to make the members that follow them public members or private members respectively. `protected`, works much like private does and is used in inheritance.
- A `class` usually uses multiple access specifiers to set the access levels of each of its members. There is no limit to the number of access specifiers you can use in a class.
- In general, attributes are usually made `private` and methods are usually made `public`.
  - The group of `public` members of a class are often referred to as a public interface. Because only public members can be accessed from outside of the class, the public interface defines how programs using the class will interact with the class.
- See Google C++ Style Guide on `class` format.

Classes | Accessors and Mutators

- An accessor (or getter) is a `public` method that allows the user to access `private` attributes.
  - The return type of the accessor is the type of the attribute you are retrieving.
  - Accessors do not need to have parameters.
  - Getters should provide "read-only" access to data. Therefore, they should return by value or const reference (not by non-const reference). Getters should also be `const`.
- A mutator (or setter) is a `public` method which allows the user to modify `private` attributes.
  - Mutators do not have a return type (`void`).
  - Mutators have one parameter, which is the value that will be assigned to the attribute.
- See Google C++ Style Guide on how to name accessors and mutators.

Classes | Encapsulation

- Why did we make attributes `private` and methods `public`?
- Modern life: 1) You use a remote control to turn your TV on/off, 2) You drive your car to work, and 3) You use a smartphone to take a picture.
- All three of these things provide a simple interface for you to use (a button, a gas pedal, a button on your touch screen) to perform an action.
- However, how these devices actually operate is hidden away from you.
- This separation of interface and implementation is extremely useful because it allows us to use objects without understanding how they work.
- This vastly reduces the complexity of using these objects, and increases the number of objects we are capable of interacting with.

Classes | Encapsulation

- In OOP, encapsulation (also called information hiding) is the process of keeping the details about how an object is implemented hidden away from users of the object.
- Instead, users of the object access the object through a public interface. In this way, users are able to use the object without having to understand how it is implemented.
- In C++, encapsulation is implemented via access specifiers.
  - Typically, all attributes of the class are made `private` (hiding the implementation details), and most methods are made `public` (exposing an interface for the user).

Classes | Encapsulation | Benefits

1. Reduce the complexity of your programs: With a fully encapsulated `class`, you only need to know what methods are publicly available to use the `class`, what arguments they take, and what values they return. It does not matter how the class was implemented internally.
2. Help protect your data and prevent misuse.

```cpp
class MyArray{
public:
    int m_array_[10];
};
int main(){
    MyArray my_array;
    my_array.m_array_[12] = 5;//--error
}
```

Classes | Encapsulation | Benefits

3. Easier to change: Consider the following code.

```cpp
class Villain{
public:
    int life_;
};
int main(){
    Villain astaroth;
    astaroth.life_ = 3;
}
```

- What would happen if we decided to rename `life_` or change its type?

```cpp
class Villain{
public:
    void set_life(int life){life_ = life;}
private:
    int life_;
};
int main(){
    Villain astaroth;
    astaroth.set_life(3);
}
```

Constructors

- When all members of a `class` are `public`, we can use aggregate initialization to initialize the `class` directly using an initialization list or uniform initialization.

```
class Villain{
public:
    std::string name_;
    double health_;
    int life_;
};
int main(){
    Villain astaroth = {"Astaroth", 100, 3};//--initialization list
    Villain firebrand{"FireBrand", 200, 1};//--uniform initialization
}
```

- However, as soon as we make any attributes `private`, we are no longer able to initialize classes in this way. If you can not directly access a variable (because it is `private`), you should not be able to directly initialize it.
- We can initialize a class with private attributes through constructors.

Constructors

- A constructor is a special kind of class method that is automatically called when an object of that class is instantiated.
- Constructors are typically used to initialize member variables of the class to appropriate default or user-provided values, or to do any setup steps necessary for the class to be used (e.g. open a file or database).
- Unlike normal member functions, constructors have specific rules:
  - Constructors must have the same name as the class (with the same capitalization).
  - Constructors have no return type (not even `void`).

Constructors | Implicit Constructor

- If your class has no constructors, the compiler will automatically generate a public default constructor for you (implicit constructor).

```
class Villain{
private:
    std::string name_;
    double health_;
    int life_;
};
int main(){
    Villain astaroth ;
}
```

- The `class Villain` has no constructor. Therefore, the compiler will generate an implicit constructor that allows us to create a `Villain` object without arguments.
- An implicit constructor does not initialize any of the attributes. If attributes are objects themselves (e.g., `std::string name_`), the constructors of those attributes will be called.

Constructors | Default Constructor

- A constructor that takes no parameters (or has parameters that all have default values) is called a default constructor. The default constructor is called if no user-provided initialization values are provided.

```
class Villain{
public:
    Villain(){
        name_ = "Villain";
        health_ = 100;
        life_ = 1;
    }
private:
    std::string name_;
    double health_;
    int life_;

};
int main(){
    Villain astaroth;//--default constructor called
}
```

Constructors | Constructors with Parameters

- While the default constructor is great for ensuring our classes are initialized with reasonable default values, often times we want instances of our class to have specific values that we provide.
- Fortunately, constructors can also be declared with parameters.
- Note: We are allowed to overload constructors.
- Question: Can we merge these two constructors into one?

```cpp
class Villain{
public:
    Villain(){
        name_ = "Villain";
        health_ = 100;
        life_ = 1;
    }
    Villain(std::string name, double health, int life=1){
        name_ = name;
        health_ = health;
        life_ = life;
    }
private:
    ...
};
int main(){
    Villain villain;//--default constructor called
    Villain astaroth{"Astaroth", 300, 3};
    Villain firebrand{"Firebrand", 100};
}
```

Constructors | Constructor Notes

- Constructors do not create objects. The compiler sets up the memory allocation for the object prior to the constructor call.
- Constructors actually serve two purposes.
    - They determine who is allowed to create an object. That is, an object of a class can only be created if a matching constructor can be found.
    - They can be used to initialize objects. It is valid to have a constructor that does no initialization at all. Much like it is a best practice to initialize all local variables, it is also a best practice to initialize all attributes on creation of the object.

Constructors | Member Initialization List

- So far, all attributes have been set in the constructor body (assignments).

```
class Villain{
public:
    Villain(std::string name, double health, int life=1){
        name_ = name;
        health_ = health;
        life_ = life;
    }
};
```

- This is similar to the following:

```
std::string name_;
double health_;
int life_;

name_ = name;
health_ = health;
life_ = life;
```

Constructors | Member Initialization List

- However, some types of data (e.g. `const` and reference variables) must be initialized on the line they are declared.

```
class Villain{
public:
    Villain(){
        name_ = "Villain;
    }
private:
    const std::string name_;
};
```

- which is similar to:

```
const std::string name_;
name_ = "Villain";
```

- Assigning values to `const` or reference member variables in the body of the constructor is clearly not possible in some cases.
- To solve this problem, C++11 introduced member initialization list to initialize class attributes (rather than assigning values to them after they are created).

Constructors | Member Initialization List

- Previous version.

```
Villain(std::string name, double health, int life=1){
    name_ = name;
    health_ = health;
    life_ = life;
}
```

- With the member initialization list.

```
Villain(std::string name, double health, int life=1):
name_{name}, health_{health}, life_{life}
{
    //--no need to do assignments here
}
```

Constructors | Initialization list order

- Variables in the initializer list are not initialized in the order that they are specified in the initializer list. Instead, they are initialized in the order in which they are declared in the class.
- The following recommendations should be observed:
    - Do not initialize member variables in such a way that they are dependent upon other member variables being initialized first (in other words, ensure your member variables will properly initialize even if the initialization ordering is different).
    - Initialize variables in the initializer list in the same order in which they are declared in your class. This is not strictly required so long as the prior recommendation has been followed.

Constructors | Non-static member initialization

- When writing a class that has multiple constructors, having to specify default values for all members in each constructor results in redundant code. If you update the default value for a member, you need to touch each constructor.
- Since C++11, it is possible to give normal non-static class attributes a default initialization value directly:

```cpp
class Villain{
public:
    Villain(std::string name):
    name_{name}{
    }

    Villain(std::string name, double health, int life=1):
name_{name}, health_{health}, life_{life}
{
    //--no need to do assignments here
}
```

```cpp
private:
    std::string name_{"Villain"};
    double health_{100};
    int life_{1};
};

int main(){
    Villain astaroth{"Astaroth"};
    astaroth.Print();
    Villain firebrand{"Firebrand", 100};
    firebrand.Print();
}
```

Constructors | Delegating Constructor

- Often, the code for overloaded constructors is very similar.

```
//--default constructor
Villain()
: name_{"Villain"}, health_{100}, life_{1}{}

//--constructor with 1 parameter
Villain(std::string name)
: name_{name}, health_{100}, life_{1}{}

//--constructor with 4 parameters
Villain(std::string name, double health, int life)
: name_{name}, health_{health}, life_{life}{}
```

- The syntax for the initialization lists is almost the same, only the values used to initialize the attributes are different.

Constructors | Delegating Constructor

- Duplicated code in overloaded constructors makes maintenance difficult.
  - If you want to add more members or change the type of existing members, you have to make the same changes three times in the `Villain` class.
- Delegating constructors (since C++11) promote code reuse.
  - Concentrate on common initialization steps in a constructor, known as the target constructor.
  - Other constructors (delegating constructors) can call the target constructor to do the initialization.

Constructors | Delegating Constructor

- Target constructor.

```cpp
//--constructor with 4 parameters
Villain(std::string name, double health, int life)
    : name_{name}, health_{health}, life_{life}{}
```

- Delegating constructors.

```cpp
//--default constructor
Villain():
    Villain("Villain", 100, 0){}
//--constructor with 1 parameter
Villain(std::string name):
    Villain(name, 100, 0){}
```

Destructor

- A destructor is another special kind of class member function that is executed when an object of that class is destroyed. Whereas constructors are designed to initialize a class, destructors are designed to help clean up.
- When an object goes out of scope normally, or a dynamically allocated object is explicitly deleted using the `delete` keyword, the class destructor is automatically called (if it exists) to do any necessary clean up before the object is removed from memory. For simple classes (those that just initialize the values of normal member variables), a destructor is not needed because C++, will automatically clean up the memory for you.
- However, if your class object is holding any resources (e.g. dynamic memory, or a file or database handle), or if you need to do any kind of maintenance before the object is destroyed, the destructor is the perfect place to do so, as it is typically the last thing to happen before the object is destroyed.

Destructor

- Like constructors, destructors have specific naming rules:
    - The destructor must have the same name as the class, preceded by a tilde.
    - The destructor can not take arguments. This implies that only one destructor may exist per class, as there is no way to overload destructors since they can not be differentiated from each other based on arguments.
    - The destructor has no return type.
- Generally you should not call a destructor explicitly (as it will be called automatically when the object is destroyed).

Header and Source Files

- Each C++ `class` is normally split up into two files.
  - The header file (.h, .hh, .hpp) contains class definitions (attributes, methods, and macros).
  - The source file (.cc, .cpp) contains the implementation of `class` methods.
  - In this course we use .h and .cpp extensions. You are free to use any default extensions that come with your IDE.
- Separating header and source files:
  - Speeds up the compile time.
  - Keeps your code organized.
  - Allows you to separate interface from implementation.

Header and Source Files

- Header files allow you to make the interface (`class` definition) visible to other source (.cpp) files, while keeping the implementation (method definitions) in its own source file.
- header files are included and not compiled.
- source files are compiled and not included.
- The `#include` statement is a copy/paste operation. The preprocessor replaces the `#include` line with the actual contents of the file being included.

Header and Source Files | Include Guards

- Prepocessor does not have brains of its own.
- If you tell it to include the same file more than once, then that is exactly what it will do.
- Multiple inclusions of the same file will result in compiler errors.

```
//main.cpp
#include "Villain/villain.h"
#include "Villain/villain.h"
```

- You will probably not include the same file multiple times in a row, however, when your program gets bigger and more complex, you may not realize you are including the same file multiple times.

Header and Source Files | Include Guards

- An include guard is a technique which uses a unique identifier to avoid the problem of double inclusion when dealing with the `#include` directive.
- In the following example, `__MYCLASS_H__` is a unique identifier, chosen by the user.
- *Best practice*: Have this identifier to be the name of your file.

```
//myclass.h
#ifndef __MYCLASS_H__
#define __MYCLASS_H__

class MyClass{
    ...
};
#endif
```

Header and Source Files | Include Guards

- `#pragma once`
  - Non-standard but widely supported preprocessor directive designed to cause the current source file to be included only once in a single compilation.
  - It acts the same way as the include guards shown in the previous slide.

```
//myclass.h
#pragma once

class MyClass{
    ...
};
```

Header and Source Files | Namespaces

- Namespaces subdivide the global scope into distinct, named scopes, and so are useful for preventing name collisions in the global scope.
- Namespaces should have unique names based on the project name, and possibly its path.
- See Google style guide on namespaces.

Header and Source Files | Domain Header File

```
//--villain.h
#pragma once
#include <string>

namespace game{
    class Villain{
    ...
    };//--class Villain
}//--namespace game
```

Header and Source Files | Domain Source File | Version 1.0

```
//--villain.cpp
#include "villain.h"

using namespace game;

void Villain::TakeDamage(double damage){
    //--code goes here
}
void Villain::Print(){
    //--code goes here
}
```

Header and Source Files | Domain Source File | Version 1.1

```
//--villain.cpp
#include "villain.h"

namespace game{
    void Villain::TakeDamage(double damage){
        //--code goes here
    }
    void Villain::Print(){
        //--code goes here
    }
    }//--namespace game
```

Header and Source Files | Domain Source File | Version 1.2

```
//--villain.cpp
#include "villain.h"

void game::Villain::TakeDamage(double damage){
    //--code goes here
}
void game::Villain::Print(){
    //--code goes here
}
```

Next Class | 10/22

- Lecture08: Object Oriented Programming – Part II.
- Quiz on raw and smart pointers.
- Assignment due on Saturday.
- Stay safe!