DATA STRUCTURE AND PROGRAMMING II

Introduction to
Object-oriented programming (OOP)

Outline

- Introduction
- Class and Object
- Inheritance
  - Overloading Vs. Overriding methods
- Encapsulation
  - Access modifier/accessibility types: public, protected, private
- Polymorphism
- Abstraction, interface

Introduction

- Data (variable)
- Behavior (function)

❑ What is OOP? OOP: Object-Oriented Programming

- OOP is the principle for programming approach based on the concept of classes and objects.
  - OOP allows us to organize software as a collection of objects that consist of both data and behavior.
- The core of OOP is to create an ***object*** that has certain ***properties*** *(variable)* and ***methods*** (function)
  - **Properties**: characteristics of an object
  - **Methods**: actions that can be performed by an object

Example:

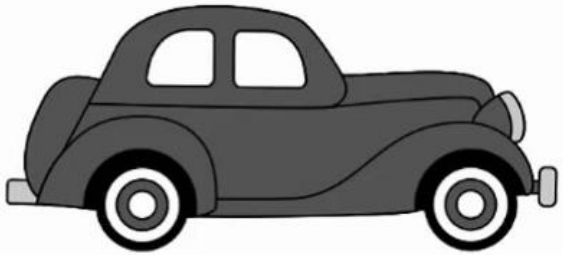
- A car is an object which has certain **properties**

such as *color*, year, price, model, etc.

- It also has certain **methods** such as

start, drive, park, brake and so on.

Properties
Make
Model
Color
Year
Price



Methods
Start
Drive
Park

Introduction

❑ What is OOP?

- The main principles of OOP are
  - Class, object, overloading and overriding
  - Inheritance,
  - Encapsulation,
  - Abstraction,
  - Polymorphism

Introduction

❑ Why OOP?

- OOP has the following main advantages
  - ✓ OOP makes it easy to maintain and modify existing code (code reusability)
  - ✓ It provides a clear structure for programs
  - ✓ It helps to manage the complexity of large software systems
  - ✓ Objects can also be reused within or across applications
  - ✓ The reuse of software also lowers the cost of development

References

❑ Sources

- Book
  - Learning Object-Oriented Programming: Explore and crack the OOP code in Python, JavaScript, and C# (OOP books using Python, C#, JavaScript)
    - http://library1.org/_ads/F485A9E07966E40D96382FF767A0271D
  - “Object-Oriented Programming Using C++”, Joyce Farrell, (4th edition)
    - http://library1.org/_ads/E599B867E1FE1C14C5E4ABA7E3F6942D
- Online document:
  - https://www.w3schools.com/cpp/cpp_oop.asp
  - https://www.tutorialspoint.com/cplusplus/cpp_classes_objects.htm
  - https://www.programiz.com/cpp-programming

Class and Object

Introduction

❑ Class and Object?

- **A class** a template/blueprint for defining something.
- **An object** is a specific item that belongs to a class.
  - An object is called an **instance of a class**.
- A class defines *characteristics* of its objects and *methods* that can be applied to its objects.
  - A class provides a way to group data and the functions that use the data.
- The concept of using classes provides a useful way to organize objects
  - It is especially useful because classes are reusable.

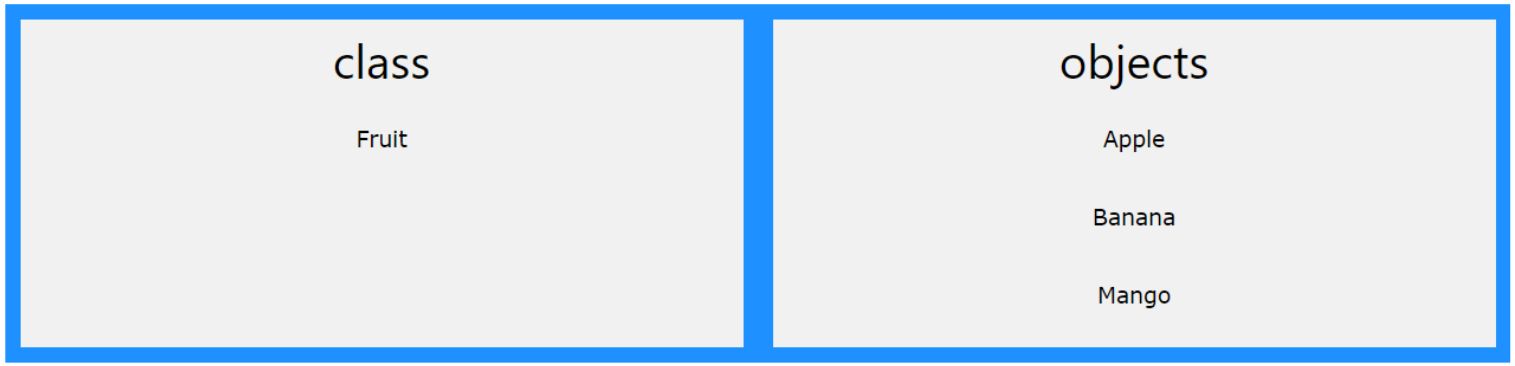
Remark: Classes are similar to structures in that both provide a means to group data and behaviors together so you can create objects. However, classes provides much more features than structures.

Class Vs. Object

C++ What are Classes and Objects?

Classes and objects are the two main aspects of object-oriented programming.

Look at the following illustration to see the difference between class and objects:



Another example:

Introduction

❑ Structure Vs. Class

- C++ supports two ways to create your own complex data types:
  - **structure**
  - **class**

```
struct Employee
{
    int idNumber;
    double hourlySalary;
};
```

```
class Employee
{
    int idNumber;
    double hourlySalary;
};
```

Figure 1-13 A simple `Employee` `struct` and `class`

Introduction

❑ Example: Structure

```
#include<iostream>
using namespace std;
struct Employee
{
     int idNumber;
     double hourlySalary;
};
int main()
{
    Employee oneStaffMember;
    oneStaffMember.idNumber = 2345;
    oneStaffMember.hourlySalary = 12.95;
    cout << "ID number is " << oneStaffMember.idNumber << endl;
    cout << "Hourly rate is " << oneStaffMember.hourlySalary << endl;
    return 0;
}
```

Figure 1-15 A structure and a `main()` method that uses the structure

Introduction

❑ Example: Class

```
class Student
{
    int idNum;
    string lastName;
    double gradePointAverage;
};
```

Figure 7-1 A `Student` class

```
class Student
{
    private:
        int idNum;
        string lastName;
        double gradePointAverage;
    public:
        void displayStudentData();
};
```

Figure 7-3 `Student` class that includes one function definition

```
class Student
{
    private:
        int idNum;
        string lastName;
        double gradePointAverage;
    public:
        void displayStudentData();
        void setIdNum(int);
        void setLastName(string);
        void setGradePointAverage(double);
};
```

Figure 7-5 `Student` class with set functions for private data

Remark: You cannot assign a value to a field in a class definition.
E.g.:, in the Student class, you cannot write idNum = 123;.
A class definition is only a description of a type; you cannot assign values to fields until you create an object.

Introduction

❑ Advantages of Using Class and Object

- ✓ When you create an object from the class, you automatically create all the related fields.
- ✓ You gain the ability to pass an object into a function as a parameter, or receive an object from a function as a returned value. Most importantly it can automatically pass or receive all the individual fields that each object contains.
- ✓ You think about class and object then and manipulate them similarly to the way you use real-life classes and objects.

Introduction

❑ Accessibility types (access modifier)

- The accessibility is used to define whether or not a variable/function can be accessed by the others. The accessibility types are
  - **public** : everyone can access
  - **private** : can only be accessed within its class
  - **protected** : can be accessed within its class or child class. Protect from everyone except its child class (use in *inheritance*)

Remark: For most C++ classes, data is private or protected, and most functions are public

Introduction

❑ REMARKS

- ✓ By default, the data of the class are private.
- ✓ By default, the methods of the class are public.
- ✓ To make a data to be public, use ***public:***
  - ✓ string lastName;
  - ✓ **public:** string lastName;
- ✓ Similarly, for the private and protected, use ***private:*** or ***protected:***

OOP

❑ Examples

Parameter name is optional in prototype

```
#include<iostream>
using namespace std;

class Student{
        int idNum;
        string lastName;
        double gpa;

        void displayStudentData();
        void setIdNum(int);
        void setLastName(string);
        void setGPA(double);
        double getGPA();
};
```

Example 1: Create a class

```
#include<iostream>
using namespace std;

class Student{
        private: int idNum;
        public: string lastName;
        private: double gpa;

        void displayStudentData();
        void setIdNum(int);
        void setLastName(string);
        void setGPA(double);
        double getGPA();
};
```

Example 2: Create a class and specify accessibility (alternative way)

```
class Student{

    public:
        int idNum;
        string lastName;
    private: double gpa;
    public:
        void displayStudentData();
        void setIdNum(int);
        void setLastName(string);
        void setGPA(double);
        double getGPA();
};
```

Example 3: Create a class and specify accessibility (alternative way)

OOP

❑ Examples

Example 4:
Create a class and define the implementation of the methods

1. Define characteristics
2. Define methods
3. Implementation of the defined methods

Parameter name is optional.

```
#include<iostream>
using namespace std;

class Student{
        int idNum;
        string lastName;
        private: double gpa;

    public:
        void displayStudentData();
        void setData(string name, int id, double GPA);
        double getGPA();
};
void Student::setData(string name, int id, double GPA){
        lastName = name;
        idNum = id;
        gpa = GPA;
}
void Student::displayStudentData(){
        cout<<lastName<<" "<<idNum<<" "<<gpa;
        cout<<"\n";
}
double Student::getGPA(){
        return gpa;
}
```

1 2 3

```
main(){
        Student s1, s2, s3;
        s1.setData("Dara", 1, 3.5);
        s2.setData("Sok", 2, 3);
        s3.setData("Sao", 3, 3);
        s1.displayStudentData();
        s2.displayStudentData();
        s3.displayStudentData();
        cout<<s1.getGPA()<<endl;
        cout<<s2.getGPA()<<endl;
        cout<<s3.getGPA()<<endl;
        cout<<s3.lastName<<endl;
        cout<<s3.idNum<<endl;
        //cout<<s3.gpa;

}
```

```
Dara 1 3.5
Sok 2 3
Sao 3 3
3.5
3
3
```

Q&A

Practice

❑ Exercise on Class and Object

1) Write a C++ program to create a class for a customer. The class *Customer* has some characteristics/properties such as *customer id, name, sex, phone*. Make the *customer id* as a private and the others (*name, sex, phone)* as public. In addition, the class *Customer* has 2 public methods such as **`setCustomerData(int id, string name, char sex, string phone)`** [for initializing the data], and **`displayACustomerInfo()`** [for displaying customer's information]. Then,

- Write code (implementation) for the method **`setCustomerData`**
- Write code (implementation) for the method **`displayACustomerInfo`**
- Create 5 objects from the class *Customer*.
- Set data for each object by using the method **`setCustomerData`** with the information from any five students of your classmates.
- Display the 1st, 3rd and 5th object by using the method **`displayACustomerInfo`**

Inheritance

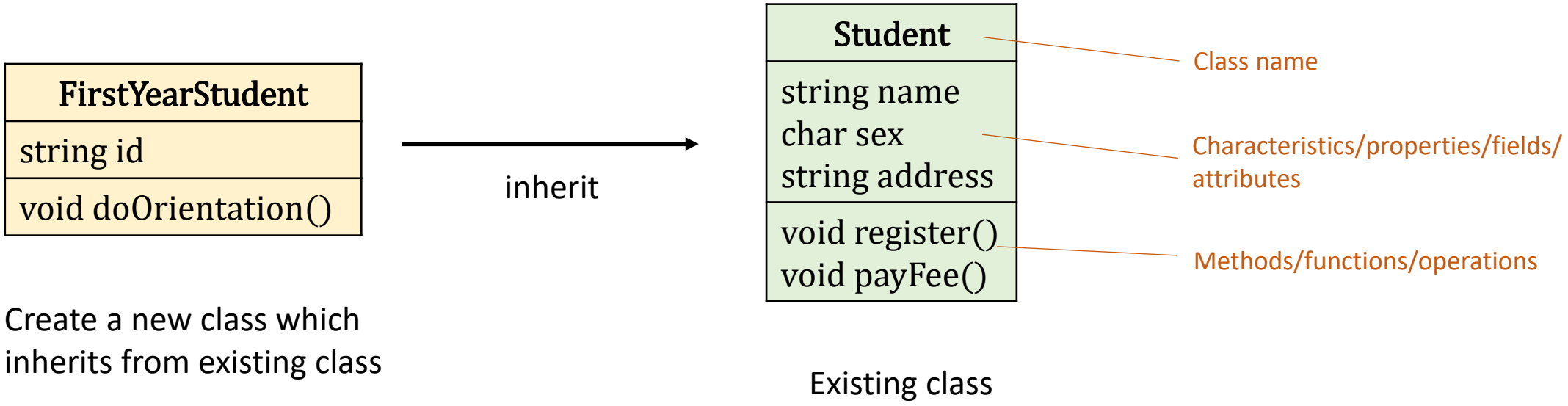
Outline

- Introdution to inheritance
- Advantages of inheritance
- Using inheritance
- Create and use a derived/child/subclass class

Introduction

❑ Inheritance

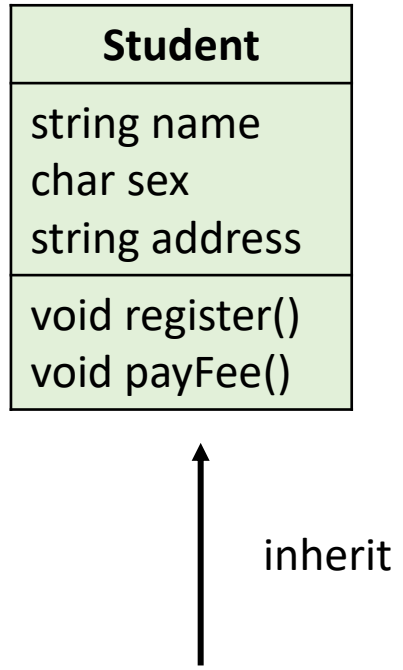
- **Inheritance** means you *create a class* that get their characteristics (data) and methods (functions) *from existing a class*



Introduction

❑ Inheritance

- Suppose you have written several programs with a class named *Student*
  - You already defined the characteristics and methods
- If you need to write a program using a new class named *FirstYearStudent*
  - You just need to inherit from the class Student
  - The class *FirstYearStudent* may require some additional data members and functions



- **Remark**: Observe the example of inheritance on the right
  - The class ***Student*** is called a ***parent class, base class, superclass*** or ***ancestor***.
  - The class ***FirstYearStudent*** is called a ***child class, derived class, subclass*** or ***descendant***.

Advantages of using Inheritance

❑ Inheritance

- ✓ Save time
  - Because you **don't have to start from scratch** when you want to create a class that has similar characteristics and methods to your existing class
  - What you need to do is just to expand on an existing class
- ✓ In a child class, you **can extend and revise a parent class without corrupting the existing parent class's features**
  - You don't have to modify a parent class to get it to work correctly with your requirement
- ✓ When you create a child class, **much of the code has been tested** and it makes this new child class more reliable

Introduction

❑ Example

```
class Person
{
   private:
      int idNum;
      string lastName;
      string firstName;
   public:
      void setFields(int, string, string);
      void outputData();
};
void Person::setFields(int num, string last, string first)
{
   idNum = num;
   lastName = last;
   firstName = first;
}
void Person::outputData()
{
   cout << "ID #" << idNum << " Name: " <<
       firstName << " " << lastName << endl;
}
```

Figure 10-1 The `Person` class

```
class Customer : public Person
{
     // other statements go here
};
```

Figure 10-2 The `Customer` class shell

The class *Customer* is inherited from the existing class *Person* (Inheritance)

```
class Customer : public Person

class Customer : protected Person

class Customer : private Person
```

Using different accessibility types

Accessibility Type (access modifier)

- **private** data and functions
  - can be accessed only within a class
- **protected** data and functions
  - can be accessed within its class or child class
- **public** data and functions
  - can be accessed anywhere

Creating a *child class* with different types of accessibility

- Using public
  - Base class members that are `public` remain `public` in the derived class.
  - Base class members that are `protected` remain `protected` in the derived class.
  - Base class members that are `private` are inaccessible in the derived class.
- Using protected
  - Base class members that are `public` become `protected` in the derived class.
  - Base class members that are `protected` remain `protected` in the derived class.
  - Base class members that are `private` are inaccessible in the derived class.
- Using private
  - Base class members that are `public` become `private` in the derived class.
  - Base class members that are `protected` become `private` in the derived class.
  - Base class members that are `private` are inaccessible in the derived class.

Introduction

❑ Example

```
class Person
{
   private:
      int idNum;
      string lastName;
      string firstName;
   public:
      void setFields(int, string, string);
      void outputData();
};
void Person::setFields(int num, string last, string first)
{
   idNum = num;
   lastName = last;
   firstName = first;
}
void Person::outputData()
{
   cout << "ID #" << idNum << " Name: " <<
       firstName << " " << lastName << endl;
}
```

Figure 10-1 The `Person` class

```
class Customer : public Person
{
   private:
      double balanceDue;
   public:
      void setBalDue(double);
      void outputBalDue();
};
void Customer::setBalDue(double bal)
{
   balanceDue = bal;
}
void Customer::outputBalDue()
{
   cout << "Balance due $" << balanceDue << endl;
}
```

Figure 10-3 The `Customer` class

```
int main()
{
   Customer cust;
     // the next two functions are defined
     // in the base class Person
   cust.setFields(215, "Santini", "Linda");
   cust.outputData();
     // the next two functions are defined
     // in the derived class Customer
   cust.setBalDue(147.95);
   cust.outputBalDue();
   return 0;
}
```

Figure 10-4 Function that uses a `Customer` object

Examples

```
#include<iostream>
using namespace std;
class Person{
    private:
        int idNum;
        string lastName;
        string firstName;
    public:
        void setFields(int, string,
string);
        void outputData();
};
void Person::setFields(int num, string
last, string first){
    idNum = num;
    lastName = last;
    firstName = first;
}
void Person::outputData(){
    cout<<"ID #"<<idNum<<"\nName:
"<<firstName<<" "<<lastName<<endl;
}
```

Example 1: Create a class

```
class Customer : public Person{
    //other statements go here
};

int main(){
    Customer c1;
    c1.setFields(1, "Sok", "Dara");
    c1.outputData();
}
```

1

2

Example 2: Create an inheritance class

1. Create a new class with inheritance to the existing class
2. Using the child class

```
ID #1
Name: Dara Sok
```

Examples

```
#include<iostream>
using namespace std;

class Person{
    private:
        int idNum;
        string lastName;
        string firstName;
    public:
        void setFields(int, string,
string);
        void outputData();
};
void Person::setFields(int num, string
last, string first){
    idNum = num;
    lastName = last;
    firstName = first;
}
void Person::outputData(){
    cout<<"ID #"<<idNum<<"\nName:
"<<firstName<<" "<<lastName<<endl;
}
```

Example 1: Create a class

```
class Customer : public Person{
    //other statements go here
    private:
        double balanceDue;
    public:
        void setBalDue(double);
        void outputBalDue();
};
```

2

```
void Customer::setBalDue(double bal){
    balanceDue = bal;
}
void Customer::outputBalDue(){
    cout<<"Balance due $
"<<balanceDue<<endl;
}
```

Example 2: Create an inheritance class then add some more characteristics and methods

```
int main(){
    Customer c1;
    c1.setFields(1, "Sok",
"Dara");
    c1.outputData();
    c1.setBalDue(147.99);
    c1.outputBalDue();
}
```

Example 3: Main program

```
ID #1
Name: Dara Sok
Balance due $ 147.99
```

2

More characteristics and methods are added to the child class

Q & A

❑ Practice exercise on Class and Object

1) Write a C++ program to create a class for a customer. The class *Customer* has some characteristics/properties such as *customer id, name, sex, phone*. Make the *customer id* as a private and the others (*name, sex, phone)* as public. In addition, the class *Customer* has 2 public methods such as **`setCustomerData(int id, string name, char sex, string phone)`** [for initializing the data], and **`displayACustomerInfo()`** [for displaying customer's information]. Then,

- Write code (implementation) for the method **`setCustomerData`**
- Write code (implementation) for the method **`displayACustomerInfo`**
- Create 5 objects from the class *Customer*.
- Set data for each object by using the method **`setCustomerData`** with the information from any five students of your classmates.
- Display the 1st, 3rd and 5th object by using the method **`displayACustomerInfo`**

Practice exercise

❑ Practice

1) Write a C++ program to create a class Computer. This class has three protected characteristics such as id, size of hdd, and size of ram. In addition, this class has two public methods, i.e i) *setSpec* which assign data to it, and ii) *displaySpec* which display characteristics' information of this computer.

2) Next, create two subclasses (Laptop and Desktop). Each subclass has some additional private characteristics such as price, model, and year. These two subclasses also have two more public methods *setData* and *showDetail*, for assigning all required data and shows all detail information.

3) Create a main program to run and test your code by

- Create two objects obj1 from Laptop class and obj2 from Desktop class
- Set data (id: 1, size of hdd: 1T, size of ram: 8GB) to obj1 using *setData* method
- Set data (id: 2, size of hdd: 2T, size of ram: 16GB) to obj2 using *setData* method
- Display all detail info of obj1 and obj2 via their showDetail methods
- Display specification of the computer obj1 using *displaySpec* method

Class

❑ Examples

Example 4:
Create a class and define the implementation of the methods

1. Define characteristics
2. Define methods
3. Implementation of the defined methods

```
#include<iostream>
using namespace std;

class Student{
        int idNum;
        string lastName;
        private: double gpa;

    public:
        void displayStudentData();
        void setData(string name, int id, double GPA);
        double getGPA();
};
void Student::setData(string name, int id, double GPA){
        lastName = name;
        idNum = id;
        gpa = GPA;
}
void Student::displayStudentData(){
        cout<<lastName<<" "<<idNum<<" "<<gpa;
        cout<<"\n";
}
double Student::getGPA(){
        return gpa;
}
```

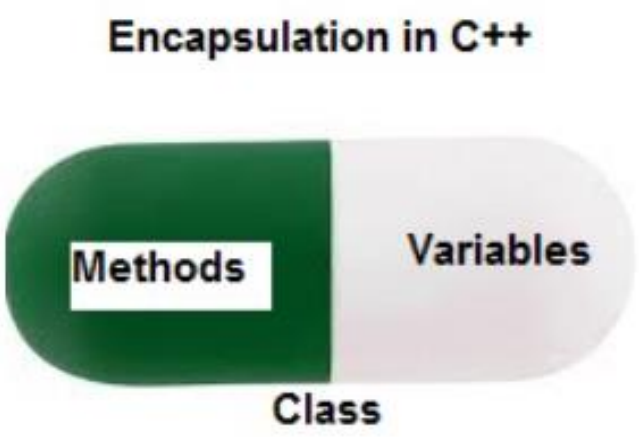
1 2 3

```
main(){
        Student s1, s2, s3;
        s1.setData("Dara", 1, 3.5);
        s2.setData("Sok", 2, 3);
        s3.setData("Sao", 3, 3);
        s1.displayStudentData();
        s2.displayStudentData();
        s3.displayStudentData();
        cout<<s1.getGPA()<<endl;
        cout<<s2.getGPA()<<endl;
        cout<<s3.getGPA()<<endl;
        cout<<s3.lastName<<endl;
        cout<<s3.idNum<<endl;
        //cout<<s3.gpa;

}
```

```
Dara 1 3.5
Sok 2 3
Sao 3 3
3.5
3
3
```

Encapsulation



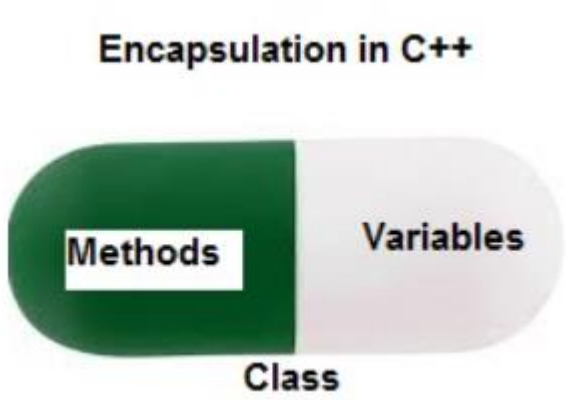
Outline

- Learn about encapsulation
- Advantages of encapsulation
- Using encapsulation

Introduction

❑ Encapsulation

- **Encapsulation** is the process of combining variables and functions into a class then you make some variables as private.
- In simple word, encapsulation is a principle that you store variables (characteristics/data) in a class as private
  - That means, only the codes inside its class can use it (data hiding)



- In other word, it is a mechanism for restricting access to some of the object's data (characteristics)
  - Data is only accessible through the functions present inside the class

- Encapsulation **helps to make the** module independent of all other modules and therefore reusable.

Introduction

❑ Why Encapsulation?

- Secure and consistent results
  - That means, giving the user access to a limited data and keeps our valuable data

- Example: Suppose that we made a Rectangle class that contained four variables (characteristics) such as ***length***, ***width***, ***area*** **and** ***perimeter***.
  - Note that *area* and *perimeter* are obtained from *length* and *width*, so changing *length* would change *area* and *perimeter*.
  - If not use info hiding (encapsulation), then another program using that Rectangle class could change the *length* without changing the *area*, and you would have an inconsistent Rectangle

Introduction

❑ Main advantages of Encapsulation

- Makes maintenance of application easier
- Improve the understandability of the application
- Enhanced security

Introduction

❑ Implementation

- To implement an encapsulation
  - We declare the variables (characteristics) that should be restrict access as a private
  - Then, we create a public function to return those variables
- Example
  - Person

Examples

```
#include<iostream>
using namespace std;
class Person{
    public:
        int idNum;
        string lastName;
        string firstName;
    private: string password;
    public:
        string getPassword(int);
        void setFields(int, string,
string);
};
void Person::setFields(int num, string
last, string first){
    idNum = num;
    lastName = last;
    firstName = first;

}
string Person::getPassword(int id){
    if(id==idNum){
        return password;
    }
}
```

Example 1: Create a class

Encapsulation

```
class Customer: public Person{
    //other statements go here
};

int main(){
    Customer c1;
    c1.setFields(1, "Sok", "Dara");
    cout<<c1.lastName<<endl;
    cout<<c1.idNum<<endl;
    cout<<c1.password<<endl;//error
    cout<<c1.getPassword(c1.idNum)<<endl;
}
```

1

2

Example 2: Create an inheritance class

1. Create a new class with inheritance to the existing class
2. Using the child class with encapsulation

Q & A

Practice exercise

❑ Practice

1) Create a class that has two properties as private and one public method.

Create an object in main. Then try to access an property of its object.

Overloading Vs. Overidding function

Introduction

❑ Overloading functions

- The processing of providing more than one functions with the same name is called method overloading (overloading function).
  - We can say: These functions have been overloaded
- Overloading helps us to provide a consistent and clear interface to our methods regardless of the parameter types.
  - Ex: We don't need to create two functions with different names addTwoInt and addTwoFloat. We can just create two functions having the same names *but* different returning type and/or parameter types.
    - int addTwo(int, int)
    - double addTwo(double, double)

Introduction

❑ Overloading functions

- **Overloading** occurs when the same function name is used with different *signatures*
  - *Signature* refers ordered list of its parameter types
- Examples:
  - void add(int, double) and void add(int, int) : they are overloading functions because they have the same names but their second parameter types are different
  - void add(int, int, int) and void add(int, int) : they are overloading functions because they have the same names but the number of parameters are not the same
  - int add(int, int) and double add(int, int) : they are overloading functions because they have the same names but they have different returning types

Overloading function

❑ Code example

```
#include<iostream>
using namespace std;

int addTwoNumbers(int n1, int n2){
    int result;
    result = n1+n2;
    return result;
}
double addTwoNumbers(double n1, double n2){
    double result;
    result = n1+n2;
    return result;
}
main(){
    cout<<addTwoNumbers(1, 1)<<endl;
    cout<<addTwoNumbers(1.3, 2.6)<<endl;
}
```

Example 1: Overloading function

Introduction

❑ Overriding functions

- **Overriding functions** occurs in inheritance when the child class has exactly *the same function* as function in the parent class
  - Same function here means they have the same returning type, the same name, the same number of parameters and the same parameter types
- Example:
  - Parent class has this function void add(int, int)
  - Child class also has its own function called void add(int, int)
  - *That means the function in child class overrides the function in its parent class*

Overriding function

```
#include<iostream>
using namespace std;
class A{
    public:
        int addTwoNumbers(int n1, int n2);
        void display();
};
int A::addTwoNumbers(int n1, int n2){
    int result;
    result = n1+n2;
    return result;
}
void A::display(){
    cout<<"Good bye! (called from class A)"<<endl;
}
class B: public A{
    public:
        void display();
};
void B::display(){
    cout<<"Good bye! (called from class B)"<<endl;
}
```

```
main(){
    A obj1;
    B obj2;
    int a=1, b=1;
    int r;

    r = obj1.addTwoNumbers(a, b);
    cout<<"Sum of a and b is: "<<r<<endl;
    obj1.display();
    cout<<"---------"<<endl;
    r = obj2.addTwoNumbers(a, b);
    cout<<"Sum of a and b is: "<<r<<endl;
    obj2.display();
}
```

```
Sum of a and b is: 2
Good bye! (called from class A)
---------
Sum of a and b is: 2
Good bye! (called from class B)
```

Output

Example 2: Overriding function

Overloading Vs. Overriding functions

❑ Comparison

- Overloading deals with multiple functions in the same class with the same name but different signatures
- Overriding deals with two functions, one in a parent class and one in a child class that have the same signature

Q&A

What we have learnt about OOP so far …

- Class and Object
- Inheritance
- Encapsulation
- Other
  - Overriding method, Access modifier

Continue ...

- Constructor
- Polymorphism
- Data Abstraction
- Interface

Constructor

- A constructor in C++ is a special **method/function that is automatically called** when an object of a class is created.
- A constructor is basically **used to initialize variables** when create object.
- To create a constructor, use the **same name as the class**, followed by parentheses ( )

```
class Car {         // The class
  public:           // Access specifier
    string brand;   // Attribute
    string model;   // Attribute
    int year;       // Attribute
    Car(string x, string y, int z) { // Constructor with parameters
      brand = x;
      model = y;
      year = z;
    }
};
```

```
int main() {
  // Create Car objects and call the constructor with different values
  Car carObj1("BMW", "X5", 1999);
  Car carObj2("Ford", "Mustang", 1969);

  // Print values
  cout << carObj1.brand << " " << carObj1.model << " " << carObj1.year << "\n";
  cout << carObj2.brand << " " << carObj2.model << " " << carObj2.year << "\n";
  return 0;
}
```

```
class MyClass {     // The class
  public:           // Access specifier
    MyClass() {     // Constructor
      cout << "Hello World!";
    }
};

int main() {
  MyClass myObj;    // Create an object
  return 0;
}
```

Remark: We can more than one constructor in a class. Just make each constructor has different parameter.

Polymorphism

- Polymorphism means "**many forms**" and it happens in inheritance.
- It occurs when we have many classes that are related to each other by inheritance.
- Inheritance lets us inherit attributes and methods from another class.
  - Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways.

```
// Base class
class Animal {
  public:
    void animalSound() {
    cout << "The animal makes a sound \n" ;
  }
};

// Derived class
class Pig : public Animal {
  public:
    void animalSound() {
    cout << "The pig says: wee wee \n" ;
   }
};

// Derived class
class Dog : public Animal {
  public:
    void animalSound() {
    cout << "The dog says: bow wow \n" ;
  }
};
```

```
int main() {
  Animal myAnimal;
  Pig myPig;
  Dog myDog;

  myAnimal.animalSound();
  myPig.animalSound();
  myDog.animalSound();
  return 0;
}
```

Remark: Polymorphism happens in inheritance in the forms of overriding methods.

Data Abstraction

- Data abstraction refers to providing only **essential information** to the outside world and **hiding their background details**,
  - i.e., to represent the needed information in program without presenting the details.
- Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

Consider a class below. It add numbers together and returns the sum.

```cpp
#include <iostream>
using namespace std;

class Adder {
   public:
      // constructor
      Adder(int i = 0) {
         total = i;
      }

      // interface to outside world
      void addNum(int number) {
         total += number;
      }

      // interface to outside world
      int getTotal() {
         return total;
      };

   private:
      // hidden data from outside world
      int total;
};
```

```cpp
int main() {
   Adder a;

   a.addNum(10);
   a.addNum(20);
   a.addNum(30);

   cout << "Total " << a.getTotal() <<endl;
   return 0;
}
```

- The public members ***addNum*** and ***getTotal*** are the interfaces to the outside world and a user needs to know them to use the class.
- The private member ***total*** is something that the user doesn't need to know about, but is needed for the class to operate properly.

Interface

- An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class.
- The C++ interfaces are implemented using **abstract classes**
  - Note: Don't confuse abstract class with data abstraction. Data abstraction is a concept of keeping implementation details separate from associated data.

- A class is made abstract by declaring at least one of its functions as **pure virtual function**.
- A pure virtual function is specified by placing "= 0" in its declaration

```
class Box {
   public:
      // pure virtual function
      virtual double getVolume() = 0;

   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};
```

- ✓ The purpose of an **abstract class** is to provide an appropriate base class from which other classes can inherit.
- ✓ Abstract classes cannot be used to instantiate objects. It serves only as an **interface**.
- ✓ We can not create an object of an abstract class. It causes a compilation error.
- ✓ A child class that inherits this abstract class will also be called an abstract class if this class does not override the pure virtual functions.

More about virtual function: https://www.geeksforgeeks.org/virtual-function-cpp/

Q&A