

CS2810: Object Oriented Algorithms Implementation and Analysis Lab

Instructor: Kartik Nagar

TAs: TBD

Course Overview

- Learn the fundamentals of object-oriented concepts and programming.
- Implement some of the data structures and algorithms from CS2800 using object-oriented concepts.
- Programming language will be C++.
 - Note that OO concepts are general, applicable across many languages.

Learning Outcomes

- Analyze a given problem and model it using objects.
- Use existing data structures and algorithms, and develop methods to solve the problem.
- Note that learning the entirety of C++ is not an outcome of this course. We will only focus on a relevant subset of C++.

Learning Mode

- Lecture on OO Concepts, followed by in-person lab sessions in DCF+Systems Lab
 - Q Slot, Tuesday 2-4:45 PM.
- Total 11 lab sessions (excluding midsem and endsem exams).
 - Mandatory Attendance, as per institute norms (minimum 85%). **You must attend at least 9 out of 11 sessions.**
 - No Lab session today.
- Programming and assignment/exam submissions will be through Hackerrank.

Grading Policy

Lab Assignments (total 11, each 7%): 77%

Midsem Exam (March 7): 11%

Endsem Exam (April 25): 12%

Grading Policy

Lab Assignments (total 11, each 7%): 77%

Midsem Exam (March 7): 11%

Endsem Exam (April 25): 12%

- Lab assignments will have an “in-lab” component which must be submitted within the lab slot.
 - There will also be a “take-home” component, for which the deadline will be Sunday 11:45 PM.
- The exams will be fully “in-lab”.
- There will be no graded assignment in the first week.

Late Submission Policy

- To help students cope with unexpected emergencies, there will be a total of **5 grace days** during which late submissions will not be penalised.
- 5 grace days must be utilised across the entire semester, so use them judiciously.
- Grace days are only applicable for assignments, not exams.
- Once grace days are exhausted, late submissions will not be evaluated.

Academic Honesty

- Any malpractice including plagiarism will be directly referred to IITM Discipline and Welfare of Students Committee.
- Penalties for violation
 - First violation: 0 marks in the corresponding component, drop of one penalty in overall course grade
 - Second violation: 'U' grade in the course.
 - DWC may impose more penalties.
- To avoid plagiarism, please discuss only high-level programming ideas with each other. **You should totally avoid looking at each others actual programs.**

Lab Assignment Details

- Use your **email address** to create an account on Hackerrank.
 - Make all submissions using this account only.
 - Multiple submissions are allowed. We will consider your last submission for grading purposes.
- Hackerrank links for lab assignments will be released on Moodle.
- Each student will also be assigned a TA, whom they can contact for any questions/guidance related to lab topics.
 - Will be done by the end of the week.

Reference Books

- **[Lafore]** Robert Lafore. Object-oriented programming in C++. Pearson Education, 1997.
- **[Weiss]** Data Structures and Algorithm Analysis in C++, by Mark Allen Weiss (Pearson 2007).
- **[Goodrich]** Michael T. Goodrich, Roberto Tamassia, and David M. Mount. Data structures and algorithms in C++. John Wiley & Sons, 2011.
- **[CLRS]** Introduction to Algorithms, by Cormen, Leiserson, Rivest, and Stein, MIT Press, Third Edition, 2009

Acknowledgments

- The lecture slides are based on course material prepared by the previous instructors of the course (Prof. LA Prashanth, Prof. Rupesh Nasre), as well as the reference books.

Let's START!

Object-Oriented Programming

- In procedural style (such as usual C programs), we solve problems using algorithmic steps.
- In OO style, we solve problems by casting them into objects and interactions among them.

```
#include <iostream>

int main()
{
    std::cout << "Hello World!" << endl;
    return 0;
}
```

Procedural

```
#include <iostream>
...
int main()
{
    Message msg("Hello World\n");
    msg.print();
    return 0;
}
```

Object-Oriented

Procedural vs. OO

- OO allows us to build a program in the application's **vocabulary**.
 - e.g.: student, teacher, lecture, exam, question, ...
 - e.g.: car, brake accelerator, wheel, seat, key, ...
- Procedural is often **top-down** (from programs to functions); OO is more of a **bottom-up design** (from classes to programs).
- OOP was devised to primarily cope with the complexity of large programs.
 - A major advantage is that conceptual errors in an OO program can be flagged by the compiler.

Additional Capabilities

- Helps in enforcing data hiding.
 - `public`, `private`, `protected`.
- Allows reuse of functionality.
 - Inheritance
- Enables change of behaviour under different contexts.
 - Polymorphism
- Allows creation of generic functionality
 - Templates

Interface

- Interface is defined to be the **public methods** of a class.
 - Behavior visible to the outside world: What clients need to know.
- Implementation details are hidden.
 - Allows changing implementation without changing behaviour.
- E.g. `int value` can be replaced by `int * value`, `int value[50]`, `Stack value`, etc.
 - From clients point-of-view, the behaviour of `read` and `write` does not change.

```
class IntCell
{
    public:
        int read();
        void write();
    private:
        int value;
}
```


Abstraction

- Abstraction simplifies complexity.
 - Consider an electric switch: an user only needs to know of switch-on and switch-off, not about ground, live and neutral wires, circuit, etc.
 - Consider a two-wheeler: a user only needs to know lever, brake, on-off switch, etc. not about engine, cylinders, valve control.
- Interface defines an abstraction.

Class and Object

- Class can be viewed as a Type.
- Object can be viewed as a variable/instance.
 - E.g. Card myCard; Student CS21B001;
 - Card, Student are classes; myCard, CS21B001 are objects.
- Each object has all the properties defined by its class.
 - It has all the member fields.
 - It has all the member function.

```
class Card {  
public:  
    std::string    getNumber();  
    std::string    getName();  
    std::string    getExpDate();  
    int            getCVV();  
    bool           chargeIt(double  
price);  
  
private:  
    std::string    number;  
    std::string    name;  
    std::string    expdate;  
    int            cvv;  
};
```

Encapsulation

- Putting data and associated functions together is called [encapsulation](#).
- Encapsulation improves programmer productivity, software design as well as software efficiency.

Access Permissions

- Unlike `struct` in C, members of C++ classes have access permissions.
 - `public`, `private`, `protected`.
 - We can say that all `struct` fields are `public`.
- Language enforces access checks.
 - Such checks help programmers to avoid inadvertent or unintentional access.
 - This improves the overall software design.
 - Note that this has nothing to do with security, secrets, etc.
 - The goal is to hide one part of the program from other parts.

Access Permissions

- A class has two types of members: fields and methods.
- We divide the world into three parts:
 - Class, immediate children (inheritance), rest of the world.

| | public | protected | private |
|----------|--------|-----------|---------|
| class | ✓ | ✓ | ✓ |
| children | ✓ | ✓ | ✗ |
| rest | ✓ | ✗ | ✗ |

Constructor

- A constructor is called (automatically) when an object is created/instantiated.
 - It is a special method with the same name as the class name, without any return type.
- It typically assigns initial values to fields and allocates resources.

Constructor-Example

Constructor

```
class IntCell
{
    public:
        IntCell(int initialValue) {storedValue = initialValue;}
        int read() {return storedValue;}
        void write(int x) {storedValue = x;}
    private:
        int storedValue;
};
```

Constructor Overloading


```
class IntCell
{
    public:
        IntCell() {storedValue = 0;}
        IntCell(int initialValue) {storedValue = initialValue;}
        int read() {return storedValue;}
        void write(int x) {storedValue = x;}
    private:
        int storedValue;
};
```


Alternate constructor

Default Value

Initialization list

```
class IntCell
{
    public:
        IntCell(int initialValue=0) : storedValue(initialValue) {}
        int read() {return storedValue;}
        void write(int x) {storedValue = x;}
    private:
        int storedValue;
};
```



Creating Objects

```
class IntCell
{
    public:
        IntCell(int initialValue=0) : storedValue(initialValue) {}
        int read() {return storedValue;}
        void write(int x) {storedValue = x;}
    private:
        int storedValue;
};

int main()
{
    IntCell i1;
    IntCell i2(50);
    IntCell * i3 = new IntCell(100);
    IntCell i4 = 500;
}
```

 This works because the constructor is implicitly called

Creating Objects

`class IntCell` keyword 'explicit' forces the constructor to be called explicitly

```
{  
    public:  
        explicit IntCell(int initialValue=0) : storedValue(initialValue) {}  
        int read() {return storedValue;}  
        void write(int x) {storedValue = x;}  
    private:  
        int storedValue;  
};
```

Recommended Practice: Use explicit in all constructors

```
int main()  
{  
    IntCell i1;  
    IntCell i2(50);  
    IntCell * i3 = new IntCell(100);  
    IntCell i4 = 500;  
}
```

This will now generate a compilation error

Separating the interface and implementation

```
class IntCell
{
    public:
        IntCell(int initialValue=0);
        int read();
        void write(int x);
    private:
        int storedValue;
}
```

IntCell.h

```
#include "IntCell.h"
#include <iostream>

IntCell::IntCell(int initialValue=0):
    storedValue(initialValue){ }

IntCell::read() {return storedValue;}

IntCell::write(int x) {}

int main()
{
    IntCell i1;
    IntCell i2(50);
    IntCell * i3 = new IntCell(100);
    IntCell i4 = 500;
}
```

Default Constructor

- If we do not define any constructor, then C++ provides a default (zero-parameter) constructor, which will initialise all the fields to their default values.
- If we define any constructor, then C++ will not provide the default constructor.

Accessors and Mutators

- A member function that only reads the state of its object, but does not change it is called an **accessor**.
- A member function that changes the state is called a **mutator**.
- Accessors can be marked using the keyword **const**.

```
class IntCell
{
    public:
        IntCell(int initialValue=0)
            :storedValue(initialValue) {}
        int read() const
            {return storedValue;}
        void write(int x)
            {storedValue = x;}
    private:
        int storedValue;
};
```

Recommended Practice: Declare all accessors using const