Fundamentals of C++ Programming

Class
Lecturer: Duc Dung Nguyen
Credits: 4
Outcomes

❖ Understand the concept of Class.
❖ Understand advantages of Object Oriented Programming (OOP).
❖ Be able to program using OOP technique.
Outline

- Class:
  - Concept and definition
  - Encapsulation
  - Static, const members
- Constructor/Destructor
- Overloading operators
- Pointer and dynamic allocation
- Friendship and inheritance
Class

❖ What is class?
❖ Structured Programming
  ❖ Define data structures
  ❖ Tasks and sub-tasks: functions
❖ Pros and Cons
❖ Object Oriented Programming (OOP)
Class

- C++: not a pure object-oriented language
- OOP: a programming paradigm based on the concept of “objects”
  - Programs are designed by making them out of objects that interact with one another.
- E.g.: C++, Python, Smalltalk, Delphi, Java, C#, Perl, Ruby, PHP, etc.
Class

❖ OOP’s features:

❖ **Encapsulation**: grouping related data and functions together as objects and defining an interface to those objects.

❖ **Inheritance**: allowing code to be reused between related types.

❖ **Polymorphism**: allowing a value to be one of several types, and determining at runtime which functions to call on based on its type.

❖ Etc.
Class

- OOP’s terminologies
  - **Object** is an instant of a particular **class**
  - **Procedures** are known as **methods**
  - **Variables** are known as **fields**, **members**, **attributes**, or **properties**
- Class: a user defined datatype which groups together related pieces of information
  - **Data + Methods**
Class

❖ Class definition:
❖ class keyword
❖ class name
❖ access specifiers
❖ members: variables, functions

```cpp
class Student {
    int    ID;
    char   name[50];

public:
    void  setName(char*);
    char* getName();
    void  doHomework(HW_t &personalHW);
    void  takeExam(float* grade);
};
```
Class

- class <CName> [: <access specifier> <other classes>] {
  <access specifier>: 
  <members>; 
  <access specifier>: 
  <members>; 
  ...
} [<objects>];

- Default access specifier: private
Class

- **Access specifier:**
  - **private:** members are accessible only from within other members of the same class (or from their friends).
  - **protected:** members are accessible from other members of the same class (or their friends), but also from their derived classes.
  - **public:** members are accessible from anywhere where the object is visible.
Class

❖ Encapsulation

❖ Packaging related stuff together

❖ User need to know only public methods/data of the object: interface

❖ Interfaces abstract away the details of how all the operations are performed

❖ “Data hiding”, “black box”.

CuuDuongThanCong.com
Define a method of class

```
<return type> <CName>::<function>(<parameters>) {
    <statements>;
}
```

```
class <CName> [[: <access specifier> <CName> ]{
    <access specifier>:
        <return type> <function>(<parameters>) {
            <statements>;
        }
    ...
} [<objects>]];
```
## Class

### Example:

```cpp
class Student {
    int ID;
    char name[50];
public:
    void setName(char* pStr) {
        if (pStr && strlen(pStr) > 0)
            strcpy(name, pStr);
    }
    char* getName() { return name; }
    void doHomework(HW_t &);
    void takeExam(Exam_t, float*);
};
```

```cpp
#include <math.h>
#include "Student.h"

void Student::doHomework(HW_t &personalHW) {
    if (hasTime()) playLoL();
    else if (deadlineTomorrow()) overnight();
    else goToSleep();
}

void Student::takeExam(Exam_t test, float* grade) {
    if (isEasy(test)) *grade = 7 + (random() % 30) * 0.1f;
    else if (!isHard(test))
        *grade = 5 + (random() % 30) * 0.1f;
    else *grade = (random() % 50) * 0.1f;
}
```
Class

❖ Scope operator ::
  ❖ Is used in the definition of member function outside the class
❖ Inline function vs. normal function
  ❖ Member functions defined in the class definition is considered as inline function.
Class

Example

Student.h

```cpp
class Student {
    int    ID;
    char   name[50];
public:
    void  setName(char* pStr) {
        if (pStr && strlen(pStr) > 0)
            strcpy(name, pStr);
    }
    void  setID(int i) { ID = i; }
    char* getName() { return name; }
    void  doHomework(HW_t &);
    void  takeExam(Exam_t, float*);
};
```

test.cpp

```cpp
#include <iostream>
#include "Student.h"
#define MAX_MEMBERS    100

int main(int narg, char **argv) {
    Student    sList[MAX_MEMBERS];
    for (int i = 0; i < MAX_MEMBERS; i++) {
        sList[i].setName("unknown");
        sList[i].setID(i);
    }
    return 0;
}
```
Class

- Pointer to class
  - Objects can also be pointed by pointers. Class is a valid type.
  - Class pointers is similar to struct pointers.
- E.g.:
  ```cpp
  Student *pHuy = new Student;
  (*pHuy).setName("Huy");
pHuy->doHomework(hw);
pHuy[0].takeExam(ex, &grad);
  ```
Class

- Pointer to class
- **this**: a pointer to the object itself

```cpp
class Student {
    int    ID;
    char   name[50];

public:
    void  setName(char* pStr) {
        if (pStr && strlen(pStr) > 0)
            strcpy(this->name, pStr);
    }
    char* getName() { return name; }
    void  doHomework(HW_t &);
    void  takeExam(Exam_t, float*);
};
```
Class

- **Static members:**
  - **Data:** static data members are considered as “class” variables since they are common variables for all objects of the same class.
  - **Example:** object counter
  - **Need to be initialized somewhere outside the class**
  - **Can be accessed through object or class**
  - **Function:** can only access static members of the class.
Class

 ❖ **Static members:**

```cpp
class Student {
    int        ID;
    char       name[50];
public:
    static int nStudents;
    void      addStudent() { ID = nStudents++; }
    void      delStudent() { cleanup(); nStudents--; }
    int       getNStudents() { return nStudents; }
    char*     getName() { return name; }
    void      doHomework(HW_t &);
    void      takeExam(Exam_t, float*);
}
```

```cpp
#include "student.h"
#include <iostreams>
Student::nStudents = 0;

int main(int narg, char** argv) {
    Student  A, B;
    A.addStudent();
    B.addStudent();
    cout<< "Number of students " << A.getNStudents() << "\n";
    B.delStudent();
    cout<< "Number of students after remove B: " << Student::nStudents() << "\n";
}
```

Class

- **Const** member functions:
  - `const Student A;`
  - `char* getName() const { return name; }`
- Const functions cannot modify non-static data members nor call non-const member functions.
Constructor/Destructor
Constructor/Destructor

- **Constructors**: a special function that is automatically called whenever a new object is created.
- Allow the class to initialise member variables or allocate storage.

```cpp
class <CName> {
    ...
public:
    <CName>();
    <CName>(<parameters>);
};
```
Constructor/Destructor

❖ Constructor:

❖ Can not be called explicitly as member functions

❖ Default constructor: take no parameters

❖ Is called when an object is declared but is not initialized with any arguments.

❖ E.g.:

Student A("Tam"), B("Thu", 1.7), C;
CSE_Course KTLT(A, B, C);
Constructor/Destructor

❖ Constructor:

❖ Can be overloaded

❖ The compiler automatically call the one whose parameters match the arguments.

❖ Cannot call the default constructor explicitly using empty parentheses

❖ Student E(); // default constructor is not called
   Student F{}; // uniform initialization
   Student G{“No Name”};
Constructor/Destructor

❖ Constructor:

❖ Member initialization

❖ E.g.:

```
Student() : ID(-1) { . . . }
Student() : ID{-2} { . . . }
```

❖ This initialization technique is required when a class contains other objects which has no default constructor (has to be initialized with parameters)
Constructor/Destructor

- **Destructor**: responsible for the necessary cleanup of a class when lifetime of an object ends.

```cpp
class <CName> {
    ...
    public:
    ~<CName>();
}
```
Constructor/Destructor

❖ Copy constructor

❖ <CName>(([const] <CName>&);

❖ Student::Student(const Student& ref) : ID(ref.ID), name(ref.name) {}

❖ Shallow copy vs. deep copy

❖ E.g.: Student b(a), c = a;
 Constructor/Destructor

❖ Copy assignment:

❖ \texttt{<CName>& \ operator = ([const] <CName>&);}

❖ \texttt{Student& \ operator = (const Student& ref) \{}
   \hspace{1em} \texttt{ID = ref.ID;}
   \hspace{1em} \texttt{strcpy(name, ref.name);}
   \hspace{1em} \texttt{return *this;}
\texttt{\}}
Constructor/Destructor

- **Move constructor**
  - `<CName>(([const] <CName>**)&);
  - `Student b = Student(“NV”);` // move-construction

- **Move assignment**
  - `<CName>& operator = ([const] <CName>&&);
  - `Vector a(1, 2), b(0, -1);
    a = a + b;` // move-assignment
Overloading operators
Overloading operators

❖ Like functions, operators can be overloaded

❖ Vector a, b, c;
  c.x = a.x + b.x;
  c.y = a.y + b.y;
  c /= 2;

❖ Allow the definition of specific operations on complex data structures
Overloading operators

❖ Overloadable operators:

❖ +  -  *  /  =  <  >  +=  -=  *=  /=  <<=  >>=
❖ <<=  >>=  ==  !=  <=  >=  ++  --  %  &  ^  !  |  
❖ ~  &=  ^=  |=  &&  ||  %=  []  ()  ,  ->*  ->
❖ new  delete  new[]  delete[]

❖ E.g.:
Vector Vector::operator + (const Vector &a);
Overloading operators

❖ Example:

**Vector.h**

```cpp
class Vector {
  float x, y;
public:
  Vector() { x = y = 0; }
  Vector(float x, float y);
  Vector operator + (const Vector &a);
  float getX() { return x; }
  float getY() { return y; }
};
```

**Vector.cpp**

```cpp
#include <math.h>
#include "Vector.h"
Vector::Vector(float x, float y) {
  this->x = x;
  this->y = y;
}
Vector Vector::operator + (const Vector &a) {
  Vector c;
  c.x = this->x + a.x;
  c.y = this->y + a.y;
  return c;
}
```

**test.cpp**

```cpp
#include <iostream.h>
#include "Vector.h"
int main(int narg, char** argv) {
  Vector a0(1.0f, 0), a1(0.5f, 1.0f);
  a0 = a0 + a1;
  cout << "a0 + a1: (" << a0.getX()
       << ", " << a0.getY() << ")\n";
  return 0;
}
```

CuuDuongThanCong.com
https://fb.com/tailieudientucntt
Pointer and dynamic allocation
Like other structures, objects can be allocated at runtime

Using `new` and `delete` operators

```cpp
cout << "Please input number of students in the class: " ;
cin >> N;
Student *sList = new Student[N];
if (sList != NULL) {
    for (int i = 0; i < N; i++) sList[i].setID(i + 1);
delete [] sList;
}
```
Pointer and dynamic allocation

- **delete vs. delete []**
  - delete: release single element
  - delete []: release array of elements

- **Memory leaks issues**
  - Forget release memory after using
  - Release with inappropriate operator
Pointer and dynamic allocation

- C++ vs. Java
- Be a responsible person!
- Practice memory management methods
Friendship and inheritance
Friendship and inheritance

❖ Friendship

❖ Friends are functions or classes declared with the `friend` keyword

❖ A non-member function can access private and protected members of class if it is declared as a friend of class.

❖ E.g.:
```cpp
class Student { 
    public:
        friend Student duplicateStudent(Student& a);
};
```

CuuDuongThanCong.com
Friendship:

- Friend class: is a class whose members can access to private and protected members of other classes.

```
class Lecturer;
class Student {  
    friend class Lecturer; // lecturer is a friend  
  ...
};
```
Friendship and inheritance

- Inheritance
  - Allow classes extend features from the base classes.
  - The *derived class* inherits the members of the *base class*, on top of which it can add its own members.
Friendship and inheritance

❖ Inheritance

❖ class <CName> [: <access specifier> <BaseCName>] {
   . . .
};

❖ E.g.:
class Polygon : public Shape { . . . };
class Rectangle : public Polygon { . . . };
class Square : public Rectangle, public Rhombus { . . . };
class Ellipse : public Shape { . . . };

Friendship and inheritance

❖ Example:

Shape.h

class Shape {
    int id;
public:
    Shape() { id = 0; }
    ~Shape();
    void draw();
};

Polygon.h

#include "Shape.h"

class Polygon : public Shape {
    int nVertex;
    Vector2D* pVertex;
public:
    Polygon(int n) : Shape(), nVertex(n) {}  
    ~Polygon();
    void draw();
};

Ellipse.h

#include "Shape.h"

class Ellipse : public Shape {
    float theta;
    Vector2D center, len;
public:
    Ellipse();
    ~Ellipse();
    void draw();
};
Friendship and inheritance

❖ Inheritance

❖ Access types

<table>
<thead>
<tr>
<th>Access</th>
<th>public</th>
<th>protected</th>
<th>private</th>
</tr>
</thead>
<tbody>
<tr>
<td>Members of the same class</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Derived class</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>outside</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>
Friendship and inheritance

❖ Inheritance

❖ **protected**: all public members of the base class are inherited as protected in the derived class.

❖ **private**: all base class members are inherited as private in the derived class.
Friendship and inheritance

- Inheritance
  - A **publicly** derived class inherits access to all members of base class except:
    - constructor, destructor
    - assignment operators
    - friends
    - private members
Discussion

- C vs C++:
  - Class vs. struct, union
    - C++ has blurred the concept of struct in C.
  - Structured programming vs OOP
- Polymorphism
- Templates
Summarise

- Understand Class: concept and definition, encapsulation
- Member functions, static and const members
- Constructor/Destructor and overloaded operators
- Pointer and dynamic allocation
- Friendship and inheritance
Quiz & homework