# Object-Oriented Programming: Class and Object

Lectured by Ming-Te Chi 紀明德

Computer Science Department
National Chengchi University

First Semester, 2022

Slides credited from 李蔡彥 and 廖峻鋒

# Class and Object - Object-based Programming

- Encapsulation - combining data with functions
  - Data members
  - Member functions
- Access specifiers
- Inline member functions and constant functions
- Accessor and mutator functions
- Constructors and destructors

# Is OOP Possible in C?

- Define the following data structure and functions

```
struct PointT {
    int x, y;
};
void SetValues(struct PointT *object, int inX, int inY) {
    object->x = inX;
    object->y = inY;
}
int Addxy(struct PointT inObject) {
    return inObject.x + inObject.y;
}
int main() {
    struct PointT object;
    SetValues(&object, 2, 3);
    printf("%d\n", Addxy(object));
}
```

- Good programming practice in C can achieve ***encapsulation*** (data-hiding) - the ability to bundle data together with the functions that act on the data.

# Is OOP Possible in C++ without class?

- Define the following data structure and functions

```
struct PointT {
    int x, y;
};
void SetValues(PointT &object, int inX, int inY) {
    object.x = inX;
    object.y = inY;
}
int Add(const PointT &inObject) {
    return inObject.x + inObject.y;
}
void main() {
    PointT object;
    SetValues(object, 2, 3);
    cout << Add(object);
}
```

- Good programming practice in C can achieve ***encapsulation*** (data-hiding) - the ability to bundle data together with the functions that act on the data.

# Syntax and Terminology of Encapsulated Types in C++

**C++ provide a more natural way to achieve encapsulation for ADT.**

```
class PointT {
    int x;
    int y;
    void setValues(int inX, int inY);
    int add();
};




void main() {
    PointT object;
    object.setValues(2,3);
    cout << object.add();
}
```

data member

required

member function

calling the member functions
Dot notation for a struct

# Syntax and Terminology of Encapsulated Types in C++

**C++ provide a more natural way to achieve encapsulation for ADT.**

```
class PointT {
    int x;
    int y;
    void setValues(int inX, int inY);
    int add();
};

void PointT::setValues(int inX, int inY) {
    x = inX;
    y = inY;
}

int PointT::add() {
    return x + y;
}

void main() {
    PointT object;
    object.setValues(2,3);
    cout << object.add();
}
```

**member function name**

**Data member(object is implicit)**

# Encapsulation in C++: Classes

- Class: a new data type that contains data and functions.
- Object: instance of a class.

**Built-in Type**

```
int
float
```

**Variable**

```
int year;
float length
```

**User-Defined Type**

```
class PointT {
      int x;
      int y;
      void setValues(int, int);
      int add();
};
```

**Objects**

```
PointT object;
PointT start_point;
PointT end_point;
PointT light_source;
```

# Encapsulation in C++: Classes

- Class: a new data type that contains data and functions.
- Object: instance of a class.
- What you can not do:
  - `add();`

    Error: `undefined identifier ’add'`
  - `PointT::add();`

    Error: `illegal use of non-static member`
  - `cout << x;`

    Error: `undefined identifier 'x'`

```
class PointT {
        int x;
        int y;
        void setValues(int, int);
        int add();
};
```

- What you can do:
  - `object.setValues(2,3);`
  - `cout << object.add();`
  - `object.x = 4; //only when x in public`

```
class PointT {
public:
        int x;
};
```

# Classes and Access Specifiers

- `public` and `private` specifiers

```
class PointT {
  public:
    void setValues(int inX, int inY);
    int add();
  private:
    int x;
    int y;
};
void main() {
    PointT object;
    object.setValues(2,3);
    object.x = 4;
    cout << object.add();
}
```

**Error:** illegal access to private member

- Private members can only be accessed in *member* functions.
- `public` and `private` specifiers can appear more than once.
- `public` is preferably to appear before `private`.

# Class Member Access Control

- Rules: encapsulation is to keep implementation details from abstraction; therefore,
  - It is a good practice to have data members always be private.
  - Member functions should be private unless they must be public.
- How if data member is public?
  - The "`object.X = 5;`" statement will break if the variable name x or its type are changed.
  - There could be some internal checking to do before the assignment.
- Why do we need private functions?
  - Suppose that you are implementing a calendar class that prints up months, keeps track of appointments, etc.
  - Does the following function need to be `public`?

```
bool CalenderT::isLeapYear(int year) {
    return (((year%4==0)&&(year%100!=0))||(year%400==0));
}
```

# Scope of Class

- A class is valid within its file or within another file which includes the class declaration.
  - Classes are typically declared within the .h files.
  - Member functions are defined within the .cc files.

```
// PointT.h
class PointT {
  public:
    setValues(int inX,
              int inY);
    int add();
  private:
    int x;
    int y;
};
```

```
// PointT.cc
#include <PointT.h>
void PointT::setValues(int inX,
                       int inY) {
    x = inX;
    y = inY;
}
int PointT::add() {
    return x + y;
}
```

- Two classes can have member functions of the same name.

```
mathObject.setValues(3, 4);
graphicsObject.setValues(4, 5);
```

# Inline Member Functions

- Member functions can be *inline*, but it cannot be called before its definition due to its internal linkage. Therefore, inline member functions are usually defined right after class definition in the .h file.

```
class PointT {
...
    inline void setValues(int inX, int inY);
    int add() { return x + y; }
...
};
inline void PointT::setValues(int inX, int inY) {
    x = inX;
    y = inY;
}
```

- Member functions that are defined in a class definition are automatically *inline*. But, avoid this unless the function is really short.

# Constant Member Functions

- A member function can be declared as *constant*.

```
class PointT {
...
    int add() const;
...
};
int PointT::add() const {
    return x + y;
}
```

```
void PointT::print() const {
    cout << add() << "\n";
}
// assuming add() is not const
```

**Error:** cannot pass const data object non-const member function

- Assume that you don't bother to add the `const` keyword, then other `const` functions cannot call this `add` function.

# Accessor and Mutator Functions

- Accessor functions:
  - Definition: a function that reads one or more data members but does not change them.
  - Example: `get(), add(), print(), etc.`
- Mutator functions:
  - Definition: a function that alters one or more data members.
  - Example: `setValues();`
- Simple accessor and mutator functions are often *inline*.
- Should you give every data member an accessor and a mutator function?
  - Never give the client more than absolutely necessary.
  - Combine several related mutator functions into one.

```
calendarObject.setDay(31);
calendarObject.setMonth(2);
calendarObject.setYear(1996);
```

```
calendarObject.setDate(31,2,1996)
```

# Constructions and Destructors

- Motivation - why do we need constructors and destructors?
- The idea and syntax of constructors
- The idea and syntax of destructors
- **When** are constructors and destructors are called?
- *Default constructors* for an array of objects
- Constructors with *default arguments*
- Constructors and *initialization lists*
- Objects within objects

# Motivation for Constructor

- Initialization is one of the most frequent activities in programming, but it is also easy to forget.

```
class ArrayT {
  public:
    void initArray(int arraySize);
    void insertElement(int element, int idx);
    int getEelement(int idx) const;
  private:
    int size;
    int *array;
};
void ArrayT::initArray(int arraySize) {
    size = arraySize;
    array = new int[arraySize];
}
void main() {
    ArrayT a;
    a.insertElement(10, 1); // segmentation fault!!!
}
```

# Basic Constructors: Idea and Syntax

- *Constructors* are functions that allow foolproof initialization. They are called when an object is created.
- Syntax: same name as the class; no return type.

```
class ArrayT {
  public:
    ArrayT(int arraySize); // constructor; no return
    void insertElement(int element, int idx);
    int getEelement(int idx) const;
  private:
    int size;
    int *arrayElements;
};
ArrayT::ArrayT(int arraySize) { // no void
    size = arraySize;
    arrayElements = new int[arraySize];
}
int main() {
    ArrayT array1(20); // create array of 20 elements
    array1.insertElement(10, 1);
    ArrayT array2; // Error: no match constructors
}
```

# Destructors

- Definition: A function called whenever a class goes out of scope.
- Motivation: to free any memory allocated by the class.
- Syntax: same name as the class name preceded by tilde '~'.

```
class ArrayT {
  public:
   ArrayT(int arraySize);
   ~ArrayT();
   void insertElement(int element, int idx);
   int getEelement(int idx) const;
  private:
   int size;
   int *arrayElements;
};
ArrayT::~ArrayT() { // no arguments
   delete [] arrayElements;
}
```

# When Are Constructors and Destructors Called?

- Static variables:
  - Constructors are called when they are *declared*.
  - Destructors are called when they go *out of scope*. (Local or global)
- Dynamic variables:
  - Constructors are called when the objects are *allocated* by `new`.
  - Destructors are called when the objects are *freed* by `delete`.

```
// Example: constructors and destructors
void Foo() {
   ArrayT array1(10); // constructor called
   ArrayT *array2;    // no initialization value
   array2 = new ArrayT(20);    // constructor called
   array1.InsertElement(5,1);  // dot notation
   array2->InsertElement(10,1);// pointer notation
   delete array2; // destructor for array2 called
} // destructor for array1 called
```

# Multiple Constructors

- A class can have more than one constructor (overloading)

```
class ArrayT {
  public:
   ArrayT();
   ArrayT(int arraySize);
   ~ArrayT();
   void setSize(int arraySize);
   void insertElement(int element, int idx);
   int getEelement(int idx) const;
  private:
   int size;
   int *arrayElements;
};
ArrayT::ArrayT() { // another constructor
   arrayElements = NULL;
}
void ArrayT::setSize(int arraySize) {
   size = arraysize;
   arrayElements = new int[arraySize];
}
```

# Default Constructors for Array of Objects

- Suppose that you are trying to allocate an array of objects which do not have a *default constructor* (a constructor having no parameters).

```
class ArrayT {
  public:
   ArrayT(int arraySize); // only constructor
   ~ArrayT();
   ... // other member functions
  private:
   int size;
   int *arrayElements;
};
void main() {
   ...
   ArrayT arrays[10];
   ...
}
```

**Error: no matched constructors**

- Reason: the compiler calls the default constructor of every object in the array.

# Solutions for the Above Problem

- In this case, you need to supply a default constructor with no arguments in addition to other constructor.
- Or, you need to eliminate all other constructors so that the compiler can automatically provide a default one for you.
- Another solution would be to require the user to create an array of objects dynamically.

```
const int kNumOfArray = 10, kDefaultArraySize = 20;
void main() {
    ArrayT *arrays[kNumOfArray];
    for(int i=0; i<kNumOfArray; i++) {
        arrays[i] = new ArrayT(kDefaultArraySize);
    }
}
```

- Which way is preferred?
  - The last solution changes the structure of the program.
  - The second solution doesn't really solve the problem.
  - The first solution can be easily achieved by using *default arguments*.

# Constructor with Default Arguments

- Consider the following example:

```
class ClientT {
  public:
    ClientT(double startingBalance = 0);
    void changeBalance(double amount);
    void showBalance() const;
  private:
    double balance;
};
void main() {
    ClientT newClient(100);
    ClientT clients[100]; //default constructor is called
    clients[0].changeBalance(100);
    clients[0].showBalance();
}
```

- Never write a default constructor that leaves your object in a uncertain or an incomplete state.

```
ArrayT::ArrayT() { // default constructor
    arrayElements = NULL;
}
```

**At least we know the pointer is either allocated or NULL.**

# Constructors with Initialization Lists

```
class PersonT {
  public:
    PersonT(char *inName, int inAge, char inType);
    ~PersonT();
    void print();
  private:
    char *name;
    int age;
    char bloodType;
};
PersonT::PersonT(char *inName, int inAge, char inType)
    :age(inAge), bloodType(inType) {
    name = new char[strlen(inName)+1];
    strcpy(name, inName);
    // age = inAge;          // originally here
    // bloodType = inType;   // originally here
}
void main() {
    PersonT myFriend("Bill Clinton", 50, 'B');
    myFriend.print();
}
```

# Objects within Objects (char *name version)

```
class PersonT {
  public:
    PersonT(char *name);
    ~PersonT();
    char *getName() const;
  private:
    char *name;
};
class DormroomT {
  public:
    DormroomT(char *myName, char *roommateName);
    void ListPeople() const;
  private:
    PersonT me;
    PersonT roommate;
};
DormroomT::DormroomT(char *myName, char *roommateName) {
    me(myName);
    roommate(roommateName);
}
void main() {
    DormroomT myRoom("John", "Mary");
}
```

**Error:** **error: no matching function for call to ‘PersonT::PersonT()’**

# Try Another Approach? (char *name version)

- Take another approach:

```
class DormroomT {
  public:
    DormroomT();   // new constructor
    void setPeople(PersonT me, PersonT roommate); // new function
  private:
    PersonT me;
    PersonT roommate;
};
void main() {
    PersonT me("Jamie"), roommate("Paul");
    DormroomT myRoom;
    myRoom.setPeople(me, roommate);
}
```

Despite the effort, you still get the same error.

**Error:** ***cannot construct direct member me and roommate.***

# A Correct Solution - Initialization Lists (char *name version)

- Use the initialization list for the DormroomT constructor.

```
DormroomT::DormroomT(char *myName, char *roommateName)
    : me(myName), roommate(roommateName) {

}
```

- Similarly, how if the data members are *constant* or *reference* variables?

```
class DormroomT {
  public:
    DormroomT(PersonT &me, const PersonT *roommate);
    ...
  private:
    PersonT& me;
    const PersonT *roommate;
};
DormroomT::DormroomT(PersonT &inMe, const PersonT *inRoommate)
    : me(inMe), roommate(inRoommate) {
}
```

# Objects within Objects (string version)

```
class PersonT {
public:
    //PersonT() {};
    PersonT(const string &name) { this->name = name;};
    //~PersonT();
    const string getName() const {return name;};
private:
    string name;
};
```

**Error:** **error: no matching function for call to 'PersonT::PersonT()'**

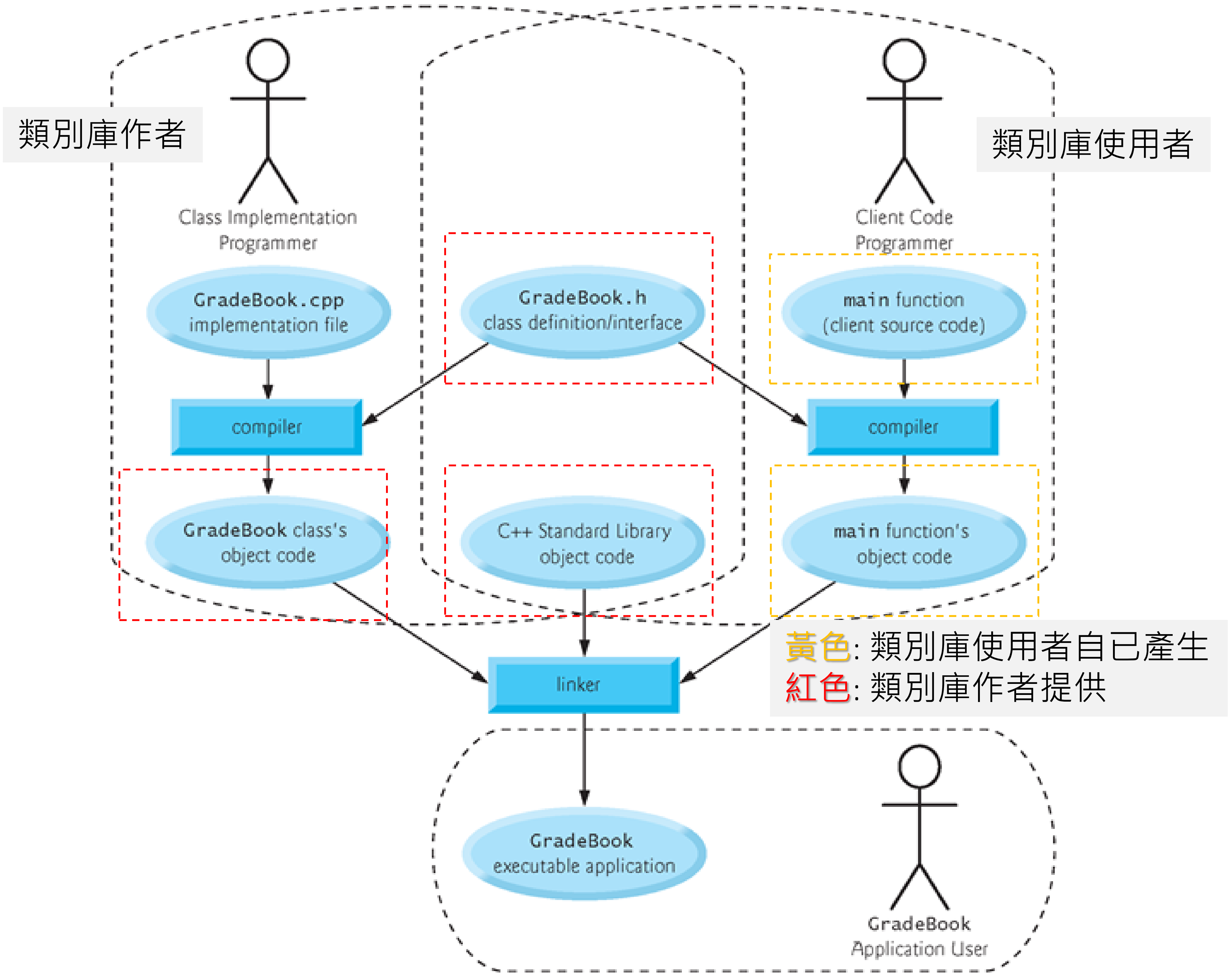
# Naming Style Convention

- Class naming style: Pascal
  - GradeBook
  - ApplicationController
- Method naming style: Camel
  - displayMessage
  - showResult

```
class GradeBook {
  public:
   void displayMessage() const {
      cout << "Welcome to the Grade Book!" << endl;
   }
};
```

# 區分類別定義與實作

- 如果要將類別/函式庫給別人使用，但又不想給他人原始碼怎麼辦?
  - 將類別宣告在.h檔
  - 將實作宣告在.cpp檔
  - 如果一來，只要給他人.h檔與cpp編譯後的二進位檔，他人使用你的類別要編譯時，只要引入.h並link起來就可以了
- 區分類別定義與實作在軟體工程上的意義
  - 修改實作時，不用影響到使用方的原始碼


類別庫作者
類別庫使用者
Class Implementation Programmer
Client Code Programmer
GradeBook.cpp implementation file
GradeBook.h class definition/interface
main function (client source code)
compiler
compiler
GradeBook class's object code
C++ Standard Library object code
main function's object code
黃色: 類別庫使用者自已產生
紅色: 類別庫作者提供
linker
GradeBook executable application
GradeBook Application User

# An Example: GradeBook.h

```
#include <string>
//using namespace std;
using std::string;

class GradeBook {
        private:
                string courseName;
        public:
                GradeBook(string name);
                void displayMessage() const;
                string getCourseName() const;
                void setCourseName(string name);
};
```

Only contain declaration.

Implementation is in .cpp.

# An Example: GradeBook.cpp

```
#include "GradeBook.h"

GradeBook::GradeBook(string name) {
        courseName = name;
}
void GradeBook::displayMessage() const {
        cout << "Welcome to the grade book for\n" <<
getCourseName() << "!" << endl;
}
void GradeBook::setCourseName(string name) {
        courseName = name;
}
string GradeBook::getCourseName() const {
        return courseName;
}
```

# An Example: main.cpp

```
#include <string>
#include <iostream>
#include "GradeBook.h"

using namespace std;

int main() {
        string nameOfCourse;
        cout << "Please enter the course name:"
                << endl;
        getline( cin, nameOfCourse );
        GradeBook myGradeBook(nameOfCourse);
        myGradeBook.displayMessage();
}
```

Note:
Only need to include the header file: GradeBook.h in order to use the class GradeBook.