EEET2482 Software Engineering Design
COSC2082 Advanced Programming Techniques

Week 4 – Classes and Objects – Part 2

Lecturer: Ling Huo Chong

Course notes acknowledgement: Linh Tran

RMIT UNIVERSITY

Review of Data Encapsulation

- Encapsulation is used to hide "sensitive" data from users.
- To encapsulate, declare attributes as **private** (*cannot be accessed directly from outside the class*).
- Provide read/write access to private attributes (if needed) through public **get()** and **set()** methods.

```
#include <iostream>

class Employee{
private:
    int salary; //private attribute

public:
    int getSalary() {
        return salary;
    }
    void setSalary(int val) {
        salary = val;
    }
};

int main() {
    Employee emp1;
    emp1.setSalary(50000);
    std::cout << "Salary = " << emp1.getSalary();

    return 0;
}
```

Output:

```
Salary = 50000
```

Assigning Objects

- If two objects have the same type, then **one object can be assigned to the other** (*values will be copied*).

```
string str1("Hello World");          // creating Object

string str2;                         // creating Object
str2 = str1;          // assign new value for an Object by another Object

string str3 = str1;   // initialize an Object by another Object
```

- Execution of Assigning Objects :
  - Define and initialize a new object from an existing object: *Copy constructor* is called
  - Assign new value for an existing object from another existing object: *Copy Assignment operator* is called
  - *C++ compiler implicitly provides the copy constructor and/or assignment operator if we don't define them. We will learn to define them by ourselves later.*

See: https://www.geeksforgeeks.org/copy-constructor-vs-assignment-operator-in-c/

Passing and Returning Objects from/to Functions

We can ***use class type as a normal data type to pass to and return from a function*** with all scenarios as below:

- Pass by value
- Pass by pointer/ reference
- Return by value
- Return by pointer/ reference

Overloading (Operators and Functions)

- We can have **multiple definitions *with* *different parameters*** for a function (*function overloading*) or an operator (*operator overloading*) in the same scope:
  - Different number of parameters
  - Same number of parameters, but different data types
- When you call an **overloaded function** or **operator**, *the compiler determines the most appropriate definition to use* (by comparing the argument list).

Function Overloading

- We can **overload constructor/ method/ any normal functions**.

```
#include <iostream>
using std::string;

class BankAcc{
public:
    string name = "";
    int amount = 0;

    /* Function overloading: 2 versions for the constructor */
    BankAcc(string name_val = "", int amount_val = 0) //v1
        : name(name_val), amount(amount_val){}

    BankAcc(int amount_val = 0) //v2
        : amount(amount_val){}

    void showInfo() {
        std::cout << "Name = " << name
                  << ", Amount = " << amount << "\n";
    }
};


int main(){
    BankAcc myacc1("Techcombank");  //v1
    myacc1.showInfo();

    BankAcc myacc2(1000000);  //v2
    myacc2.showInfo();

    return 0;
}
```

“this” keyword

- A **pointer** refers **address of the current object.**
- Implicit parameter (automatically passed) to all member functions of a class.
  - ➡ In a member function, **`this`** can be used to refer to the current object.
  - ➡ Access members of the current objects via **ARROW operator `this->` or by dereferencing `(*this).`**

```
#include <iostream>
using std::string;

class BankAcc{
public:
    string name = "";
    int amount = 0;

    /* Function overloading: 2 versions for the constructor */
    BankAcc(string name = "", int amount = 0) { //v1
        this->name = name;
        this->amount = amount;
    }

    BankAcc(int amount = 0) { //v2
        this->amount = amount;
    }

    void showInfo() {
        std::cout << "Name = " << name
                  << ", Amount = " << amount << "\n";
    }
};

int main(){
    BankAcc myacc1("Techcombank");  //v1
    myacc1.showInfo();

    BankAcc myacc2(1000000);  //v2
    myacc2.showInfo();
    return 0;
}
```

Operator Overloading

- In C++, *the meaning of a standard operator* (eg: `=`, `+`, `-` , `++`, `--`, `*`, `<<`, `>>`, etc.) *on user-defined class/ data-type can be overloaded*.
- Almost all operators can be overloaded except a few: `.` (member selection), `::` (scope resolution), `?:` (conditional operator), `sizeof`
- Define inside the class that we want the overloaded operator to work with its objects/ variables.

```
return_type operator Symbol (Arguments) {
    ... .. ...
}
```

Overloading Unary Operators ++, --

- **No Parameters** for unary operator overloading function. By default, we can only use it as **prefix**.
- To use as **postfix**, create an extra parameter of type **int** (as default data type for ++, -- operators. F*or another unary operator, we may use another appropriate data type*)

See: https://www.programiz.com/cpp-programming/operator-overloading

```cpp
#include <iostream>

class BankAcc {
public:
    std::string name = "";
    int amount = 0;

    BankAcc(std::string name_val = "", int amount_val = 0)
        : name(name_val), amount(amount_val){}

    std::string toString() {
        return "Name = " + name + ", Amount = "
                + std::to_string(amount);
    }

    // Overload ++ when used as prefix (++ object)
    BankAcc operator ++ () {
        amount++;    return *this;
    }

    // Overload ++ when used as postfix (object ++)
    BankAcc operator ++ (int) {
        amount++;    return *this;
    }
};
```

```cpp
int main() {
    BankAcc acc1("TCB", 1000), acc2("ACB", 2000);
    BankAcc result;

    // Use the overloaded operator ++ as prefix
    result = ++acc1;
    std::cout << result.toString() << "\n";

    // Use the overloaded operator ++ as suffix
    result = acc2++;
    std::cout << result.toString() << "\n";;

    return 0;
}
```

Output:

```
Name = TCB, Amount = 1001
Name = ACB, Amount = 2001
```

EXAMPLE

Overloading Binary Operators (+, -, /, *, etc.)

- Pass the second operand as a *parameter for operator overloading function*.
- Can pass by value or pass by pointer/ reference

```
#include <iostream>

class BankAcc {
public:
    std::string name = "";
    int amount = 0;
    //…… Constructors as in previous Example ……
    // Overload + operator
    BankAcc operator + (BankAcc &acc2) {
        BankAcc temp;
        //Same name: add ammounts
        if (this->name == acc2.name) {
            temp.name = this->name;
            temp.amount = this->amount + acc2.amount;
            return temp;
        }
        //Name is different: return an object with empty attributes
        std::cerr << "Cannot add accounts: names are different !\n";
        return temp;
    }
};
```

```
int main() {
    BankAcc acc1("TCB", 1000),
            acc2("TCB", 2000),
            acc3("ACB", 5000),
            result;

    // Use the overloaded operator +
    result = acc1 + acc2;
    std::cout << result.toString() << "\n\n";

    // Use the overloaded operator +
    result = acc2 + acc3;
    std::cout << result.toString() << "\n\n";

    return 0;
}
```

Output:

```
Name = TCB, Amount = 3000

Cannot add accounts: names are different !
Name = , Amount = 0
```

Friendship

- By default, private and protected members of a class cannot be accessed from outside of the class. However, *this rule does not apply to "friends".*
- **Friends of a class** are non-member **functions** or other **classes** *declared with the **friend** keyword (which will have access to all members of the class).*

See: https://www.programiz.com/cpp-programming/friend-function-class

Friend Function

```cpp
#include <iostream>

class Distance {
private:
    int meter;

public:
    Distance() : meter(0) {}

    //Declare a non-member function is a Friend
    friend int addFive(Distance d);
};


//Definition of the function
int addFive(Distance d) {
    //Access private members of the friend class
    d.meter += 5;
    return d.meter;
}
```

```cpp
int main() {
    Distance dist;
    std::cout << "Distance: " << addFive(dist);
    return 0;
}
```

Output:

```
Distance: 5
```

Overloading Operator as a Friend Function

- Allow us to *use the* ***overloaded operator*** with first operand could be another data type.

- Approach:
  - **Declare** the **operator overloading function** is a **friend** inside the class.
  - **Define** the operator overloading function **outside** the class (*must take at least one parameter of the class type)*.

```cpp
#include <iostream>

class Point2D {
    int x, y; // 2D coordinates

public:
    Point2D(int xVal = 0, int yVal = 0) //constructor
        : x(xVal), y (yVal){};

    //Declare operator overloading functions as Friends
    friend Point2D operator +(Point2D point, int num);  //allows Point2D + int
    friend Point2D operator +(int num, Point2D point);  //allows int + Point2D

    std::string toString() {
        return "x = " + std::to_string(x) + ", y = " + std::to_string(y);
    }
};

//Define the operator overloading function for Point2D + int
Point2D operator +(Point2D point, int num) {
   Point2D temp;
   temp.x = point.x + num;
   temp.y = point.y + num;
   return temp;
}

Point2D operator +(int num, Point2D point) {
   return point + num;
}
```

```cpp
int main(){
    Point2D pointA(100, 200), pointB;

    pointB = 20 + pointA;
    std::cout << "\nValue of pointB: "
              << pointB.toString();

    pointB = pointA + 30;
    std::cout << "\nValue of pointB: "
              << pointB.toString();

    return 0;
}
```

Output:

```
Value of pointB: x = 120, y = 220
Value of pointB: x = 130, y = 230
```

EXAMPLE

Useful Functions to input/ convert String

Function	Meaning
stoi ()	Convert a string object to integer
stod ()	Convert a string object to double
to_string()	Convert numerical value to string object
= (const char* s)	Assign values of a C-type string to a string object
string (const char* s);	Initialize a string object with values of a C-type string
.**c_str**()	Get a C-type equivalent string from a string object
atoi()	Convert a C-type string to integer
atof()	Convert a C-type string to double
istream& **getline** (char* s, streamsize n);	Read a line from input stream (cin/ file) and store into a C-type string
istream& **getline** (istream& is, string& str);	Read a line from input stream (cin/ file) and store into a string object

Note: **getline** function may get an empty string if a newline '\n' character is entered before
→ should use cin.ignore() function or check size of resulting string and keep reading

Friend Class

When a class is declared a friend class, ***all the member functions of the friend class*** *are friend functions (of the friend granted class).*

```
#include <iostream>

class ClassA {
private:
    int numA = 10;

public:
    //Declare another class is a Friend
    friend class ClassB;
};


class ClassB {
private:
    int numB = 20;

public:
    //Access private members of classA from the friend class
    int add(ClassA obj_a) {
        return obj_a.numA + this->numB;
    }
};
```

```
int main() {
    ClassB objB;
    ClassA objA;

    std::cout << "Sum: " << objB.add(objA);
    return 0;
}
```

Output:

```
Sum: 30
```

Note: Declare friend class/function with private or public access specifier make no difference (but recommend to use public)

Static Member of a Class

- A static member is a member that is declared using **`static`** keyword.
- It is created for the entire class (***access directly with scope resolution :: operator***), and its **lifetime is the entire program**
- Similar with global variables, static members *should be used ONLY if necessary (due to its lifetime and scope).*

  See: https://www.tutorialspoint.com/cplusplus/cpp_static_members.htm

Example

```
#include <iostream>

using namespace std;

class Car {
public:
    std::string name;
    int price;

    static int objCount;    //static attribute
    static int getCount() { //static member function
        return objCount;
    }

    Car(std::string name, int price) {
        this->name = name;
        this->price = price;
        objCount++;
    }
};
```

```
// Initialize static attribute
int Car::objCount = 0;

int main(void) {
   cout << "Number of Car objects: "
        << Car::objCount << endl;
   Car car1("BWM", 10000), car2("Ferrari", 50000);
   cout << "Number of Car objects: "
        << Car::getCount() << endl;

   return 0;
}
```

Output:

```
Number of Car objects: 0
Number of Car objects: 2
```

Preprocessor Directives

- Preprocessor Directives executes *before* a program is compiled. Some actions it performs are:

  1. The **inclusion of other files** into the file being compiled.
  2. Definition of **symbolic constants and macros** *(work as text replacement)*.
  3. **Conditional compilation** of program code.

- Preprocessor directives begin with #

Preprocessor Directives

- `#include <filename>` //include standard library headers
- `#include "filename"` //include user-defined library headers
- `#define identifier replacement-text` //identifier will be replaced by replacement-text

  Example: `#define PI 3.14159`

  then in your code, you can use **PI** as a constant value (not variable), e.g. x = x * PI

- `#if` //Evaluates the expression/condition
- `#else #elseif` //Indicate alternative statements of an if directive
- `#endif` //Used to indicate the end of an if directive