NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

SC1008 C and C++ Programming

Assistant Professor WANG Yong

yong-wang@ntu.edu.sg

CCDS, Nanyang Technological University

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

Week 10
Class and Object

Outline

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- Introduction to Class and Object
- Member Access Specifiers
- Constructor and Destructors
- Dynamic memory allocation in a Class
- `this` Pointer in C++
- Operator Overloading
- Friend Function

The Need for New Types

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- A C++ "**Class**" is simply a very-slightly modified ***Structure***
- As with structures, we sometimes want new types:
  - A calendar program might want to store information about dates, but *C++ does not have a Date type*
  - A student registration system needs to store info about students, but *C++ has no Student type*
  - A music synthesizer app might want to store information about users' accounts, but *C++ has no Instrument type*

Introduction to Class

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- A **user-defined** data type
- Represents the set of properties or methods that are common to all objects of one type
- A blueprint for an object

  ***Example*: Car**

- Cars have different names and brands, but all share some common properties (4 wheels, Speed Limit, Mileage range, etc)

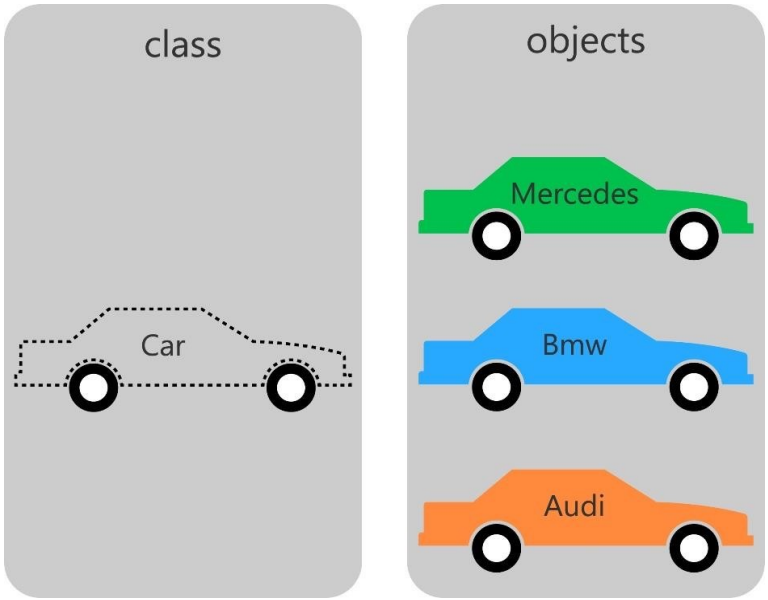
Classes are fundamental to implementing Object-Oriented Programming

Introduction to Object

- An **instance** of a class
- Represents the real-life entities
- Has an identity, state, and behavior. Each object contains data and code to manipulate the data.

Example: "My Car"

- Characteristics like color, manufactured year, price, drive, and break.



Introduction to Class

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

Elements of a Class

- **member variables**: State inside each object
  - Also called "instance variables" or "fields"
  - Declared as private
  - Each object created has a copy of each field.
- **member functions**: Behavior that executes inside each object
  - Also called "methods"
  - Each object created has a copy of each method
  - The method can interact with the data inside that object
  - Constructors and destructors are special type of member functions

Introduction to Class

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

The implementation of a Class

- When building a class, we provide an *interface* to the user of the class that details how the class works. The user often does not have access to the code itself, but the interface usually suffices to use the class properly
  - The interface is generally put into a header file, e.g., **ClassName.h**
  - The header file shows the class functions and variables (even though some may be private)
- The source code for the class member functions is held in a .cpp file, e.g., **ClassName.cpp**
  - The .cpp file is written by the implementer of the class, and they implement all of the class functions
  - The user generally does not have or need access to this code (though you can get it for open source code libraries)

Class Implementation

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- One toy example

  Best Practice: Separate the **class definition** and **member function implementations** into two files: *classname.h* and *classname.cpp*

```
// in MyClass.h
#ifndef MYCLASS_H
#define MYCLASS_H

class MyClass {
public:
    MyClass();
    MyClass(int a);
    ~MyClass(){
      std::cout<<"Bye!" <<data
<<std::endl;
}
    void doSomething();

private:
    int data;
};

#endif // MYCLASS_H
```

```
// in MyClass.cpp
#include "MyClass.h"
#include <iostream>

MyClass::MyClass(){ data = 0; }
MyClass::MyClass(int a){
        data = a;
}

void MyClass::doSomething() {
    std::cout << "Do something!"
      << std::endl;
}
```

Class Implementation

- One toy example

```
// in MyClass.h
#ifndef MYCLASS_H
#define MYCLASS_H

class MyClass {
public:
    MyClass();
    MyClass(int a);
    ~MyClass(){
      std::cout<<"Bye!" <<data
<<std::endl;
}
    void doSomething();
private:
    int data;
};

#endif // MYCLASS_H
```

Header Guards in C++:

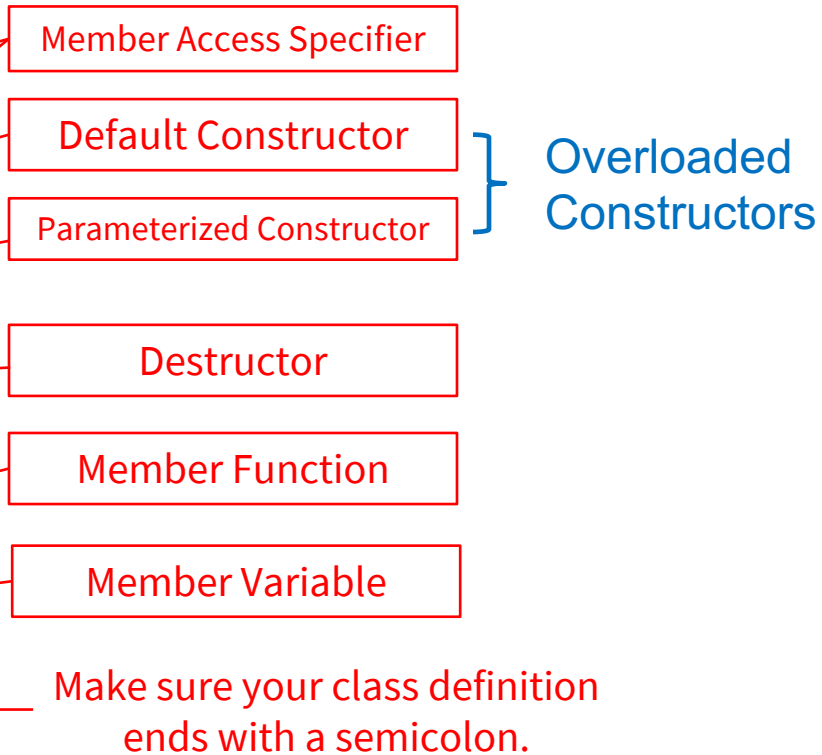
- Preprocessor directives that help to avoid errors that arise when the same function or variable or class is included/defined more than once

- Another choice:
  #pragma once

Class Implementation

- One toy example

```
// in MyClass.h
#ifndef MYCLASS_H
#define MYCLASS_H

class MyClass {
public:
    MyClass();
    MyClass(int a);
    ~MyClass(){
      std::cout<<"Bye!" <<data
<<std::endl;
}
    void doSomething();
private:
    int data;
};

#endif // MYCLASS_H
```



Member Access Specifiers

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- Classes can limit the access to their member functions and data
- The three types of access a class can grant are:
  - **`Public`** — Accessible wherever the program has access to an object of the class
  - **`private`** — Accessible only to member functions of the class, not accessible outside the class
  - **`Protected`** — Accessible to member functions of the class and its derived classes (i.e., child classes), not accessible outside the class

Member Access Specifiers

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- One toy example

```
// in MyClass.h
#ifndef MYCLASS_H
#define MYCLASS_H

class MyClass {
public:
    MyClass();
    MyClass(int a);
    ~MyClass(){
      std::cout<<"Bye!" <<data
<<std::endl;
}
    void doSomething();
private:
    int data;
};

#endif // MYCLASS_H
```

Member variables of a class are often defined as private. Why?

- We want to encapsulate them. We can provide controlled access to them by defining other public/protected member functions, e.g., user-defined functions like `getter()` and `setter()`.

Constructor

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- Special member function that initializes the data members of a class object
- Its name is exactly the same as the class name
- A constructor CANNOT return a value and has NO return type (NOT even void)
- There are 3 types of constructors:
  - Default constructors
  - Parametrized constructors
  - Copy constructors

Default Constructor

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- A constructor that can be invoked without any arguments, e.g. a constructor with an empty parameter list, or a constructor with default values for all of its arguments
- Never define more than one default constructor for a class.
- If you fail to write a constructor for a class, the compiler will automatically define a default constructor that leaves all data members un-initialized

- Whenever you defined one or more non-default constructors (with parameters) for a class, the compiler **will not automatically provide a default constructor**. If you still need a default constructor (without parameters), you must explicitly define it

Parametrized Constructor

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- Pass arguments to constructors. The initial values are passed as arguments to the constructor function
- The arguments passed to a constructor are typically used to initialise the object's member variables

Class Implementation

- One toy example

Detailed implementation of constructors and other member functions.

The scope resolution operator (i.e., **two colons ::**) is used to indicate which class the function belongs to.

Including the header file of the class

```
// in MyClass.cpp
#include "MyClass.h"
#include <iostream>

MyClass::MyClass(){ data = 0; }
MyClass::MyClass(int a){
        data = a;
}

void MyClass::doSomething() {
    std::cout << ”Do something!”
      << std::endl;
}
```

```
// in main.cpp
#include "MyClass.h"
#include <iostream>
using namespace std;

int main() {
    MyClass toyClass1;
    toyClass1.doSomething();
    cout<<
toyClass1.data<<endl;
    MyClass toyClass2(10);
}
```

Class Implementation

- One toy example

Detailed implementation of constructors and other member functions.

The scope resolution operator (i.e., **two colons ::**) is used to indicate which class the function belongs to.

Including the header file of the class

The default constructor will be executed, i.e., data = 0.

```
// in main.cpp
#include "MyClass.h"
#include <iostream>
using namespace std;

int main() {
    MyClass toyClass1;
    toyClass1.doSomething();
    cout<<
toyClass1.data<<endl;
    MyClass toyClass2(10);
}
```

```
// in MyClass.cpp
#include "MyClass.h"
#include <iostream>

MyClass::MyClass(){ data = 0; }
MyClass::MyClass(int a){
        data = a;
}

void MyClass::doSomething() {
    std::cout << "Do something!"
      << std::endl;
}
```

Class Implementation

- One toy example

Detailed implementation of constructors and other member functions.

The scope resolution operator (i.e., **two colons ::**) is used to indicate which class the function belongs to.

Including the header file of the class

```
// in MyClass.cpp
#include "MyClass.h"
#include <iostream>

MyClass::MyClass(){ data = 0; }
MyClass::MyClass(int a){
        data = a;
}

void MyClass::doSomething() {
    std::cout << "Do something!"
      << std::endl;
}
```

The default constructor will be executed, i.e., data = 0.

```
// in main.cpp
#include "MyClass.h"
#include <iostream>
using namespace std;

int main() {
    MyClass toyClass1;
    toyClass1.doSomething();
    cout<<
toyClass1.data<<endl;
    MyClass toyClass2(10);
}
```

Error!
"data" is private and cannot be accessed outside the class.

The parameterized constructor will be executed, i.e., data = 10.

Class Implementation

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- When the class and member functions are small, we can define member functions inside the class to improve readability --- but NOT recommended for large/complex member functions and classes

```
#include <iostream>

class MyClass {
public:
    MyClass() { data = 0; }
    MyClass(int a) { data = a; }
    ~MyClass() {
      std::cout << "Bye! " << data << std::endl;
    }

    void doSomething() {
     std::cout << "Do something!" << std::endl;
    }

private:
    int data;
};
```

Class Implementation

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- Constructors can initialize their members in two different ways.

- **Assignment way** (assignment statement in constructor's method body): a constructor can use the arguments passed to it to initialize member variables in the constructor definition:

```
MyClass(int a) { data = a; }
```

- **Initializer list way**: a constructor can have an initializer list within the definition but prior to the constructor body:

```
MyClass(int a): data(a) { }
```

```
//If there are more variables:

 MyClass(int a, double b): data1(a), data2(b) { }
```

https://www.ibm.com/docs/en/zos/2.4.0?topic=only-initialization-base-classes-members-c

Class Implementation

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- Update the class definition with the constructors using an initializer list

```
#include <iostream>

class MyClass {
public:
    MyClass(): data(0) {}
    MyClass(int a): data(a) {}
    ~MyClass() {
      std::cout << "Bye! " << data << std::endl;
    }

    void doSomething() {
     std::cout << "Do something!" << std::endl;
    }

private:
    int data;
};
```

Copy Constructor

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- A copy constructor is a special constructor in C++ that initializes a new object as a copy of an existing object
- There are two types of copy constructor
  - Default copy constructor
  - User-defined copy constructor

- If a copy constructor for a class is NOT defined explicitly (i.e., the user-defined copy constructor), then the compiler will automatically provide a copy constructor (i.e., the default copy constructor)
- If a user-defined copy constructor exists, the compiler will NOT automatically generate the default copy constructor

Default Copy Constructor

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

```
#include <iostream>

class MyClass {
public:
    MyClass();
    MyClass(int a);

  //No user-defined copy constructor here

    ~MyClass(){
    std::cout<<"Bye!"<<data <<std::endl;
}
    void doSomething();
private:
    int data;
};

MyClass::MyClass(){ data = 0; }
MyClass::MyClass(int a){
          data = a;
}

void MyClass::doSomething() {
    std::cout << "Do something!"
      << std::endl;
}

int main() {
    MyClass toyClass1(10);
    MyClass toyClass2 = toyClass1;
    MyClass toyClass3(toyClass1);
}
```

The compiler-generated default copy constructor is called when initializing an object from another or assigning one object to another.

toyClass2.data and toyClass3.data are also assigned as 10 like toyClass1.

Default Copy Constructor

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- What the default copy constructor actually does is as follows:

```
MyClass::MyClass(const MyClass& other){
        data = other.data;
}
```

- You can still explicitly define the default copy constructor in C++, but it is **NOT necessary and NOT recommended.**

User-defined Copy Constructor

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

```
#include <iostream>

class MyClass {
public:
    MyClass();
    MyClass(int a);
    // User-defined copy constructor
    MyClass(const MyClass &obj) {
        data = obj.data * 2;
        std::cout << "User-defined Copy constructor
is called!" << std::endl;
    }
    ~MyClass(){
    std::cout<<"Bye! "<<data <<std::endl;
}
    void doSomething();
private:
    int data;
};

MyClass::MyClass(){ data = 0; }
MyClass::MyClass(int a){
          data = a;
}

void MyClass::doSomething() {
    std::cout << "Do something!"
      << std::endl;
}

int main() {
    MyClass toyClass1(10);
    MyClass toyClass2 = toyClass1;
}
```

The user-defined copy constructor **MUST** take a **const reference of the same class** as the parameter.

The user-defined copy constructor is called.

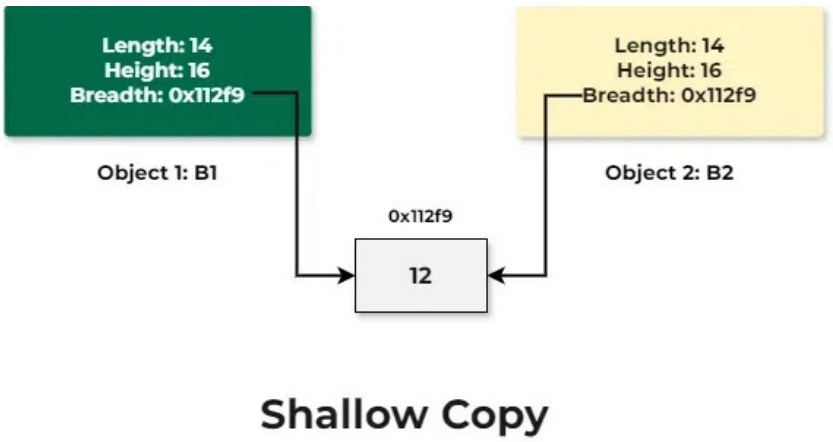
toyClass2.data is 20 now!

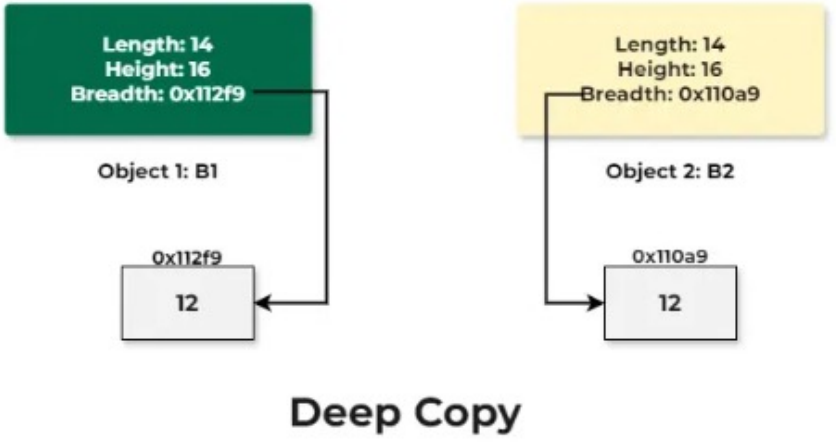
Shallow Copy vs. Deep Copy

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- **Shallow Copy:**
  - An object is created by simply copying the data of all variables of the original object
  - If a variable of the object are dynamically allocated memory (e.g., `new`), the copied object variable will also refer to the same memory location -- only the memory address is copied!

- **Deep Copy:**
  - It copies the data of all variables
  - For variables using dynamically allocated memory, it also allocates **new memory resources** to **store the same value**.





Shallow Copy vs. Deep Copy

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- Default copy constructor can perform only **shallow copy**

```
#include <iostream>
class MyClass {
public:
    int* data;  // Pointer to dynamically allocated memory

    // Constructor
    MyClass(int val) {
        data = new int(val);  // Allocating memory
        std::cout << "Constructor called, allocating memory at " << data <<
std::endl;
    }

    //No user-defined copy constructor here

    // Destructor
    ~MyClass() {
        std::cout << "Destructor called, freeing memory at " << data <<
std::endl;
        delete data;  // Deallocating memory
    }
};

int main() {
    MyClass obj1(10);   // Constructor called
    MyClass obj2 = obj1; // Default copy constructor called - Shallow Copy!!!

    // Modifying obj2 will also affect obj1 due to shared memory
    *obj2.data = 20;

    std::cout << "obj1.data: " << *obj1.data << std::endl;
    std::cout << "obj2.data: " << *obj2.data << std::endl;

    return 0; // Destructor called twice, leading to double deletion (ERROR!)
}
```

Shallow Copy vs. Deep Copy

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- **Default copy constructor** can perform only **shallow copy**

Output

```
Constructor called, allocating memory at 0x1fff2b0
obj1.data: 20
obj2.data: 20
Destructor called, freeing memory at 0x1fff2b0
Destructor called, freeing memory at 0x1fff2b0
free(): double free detected in tcache 2
Aborted
```

Destructor

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- A special member function that is automatically called whenever an object of its class ceases to exist. Its main purpose is to release any resources that the corresponding constructor allocated
- A destructor's name is the same as the name of the class it belongs to, except that the name is preceded by a tilde symbol (~)
- The destructor does not have arguments, and has no return type not even void
- A destructor should be declared in the public section of the class
- There cannot be more than one destructor in a class.
- When a destructor is NOT specified in a class, compiler generates a default destructor
- Generally, destructors are called in the reverse order of constructor calls

Destructor

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- **Destructor** is often used to **free dynamically allocated memory**, avoiding memory leakage!

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

class MyClass {
public:
    int* data;
    int size;

    MyClass(int val) {// Constructor
        data = new int(val);
        size = val;
        cout << "Constructed! The array size is " << size << endl;
    }

    ~MyClass() {// Destructor
        cout << "Destroyed! The array size is " << size << endl;
        delete data;
    }
};

int main() {

    MyClass obj1(10);
    MyClass obj2(20);
    MyClass obj3(30);

    return 0;
}
```

Destructor

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- Destructors are **called in the reverse order** of constructor calls

Output

```
Constructed! The array size is 10
Constructed! The array size is 20
Constructed! The array size is 30
Destroyed! The array size is 30
Destroyed! The array size is 20
Destroyed! The array size is 10
```

The `this` pointer in C++

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- The `this` pointer in C++ is an implicit pointer that stores the memory address of the calling object
- It is available in the member functions of a class and allows access to its own members

  - We will use arrow -> instead of dot . to access its members, as `this` is a pointer

- It is often used to 1) avoid naming conflicts and 2) return *this for method chaining

The `this` pointer in C++

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

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

class MyClass {
private:
    int number;

public:
    // Constructor
    MyClass(int number) {
        this->number = number; // Using `this` to resolve naming conflict
    }

    MyClass& setValue(int x) {
        this->number = x;
        return *this;
    }   // this will allow method chaining like: obj.setValue(10).display();

    // Display function using `this` pointer
    void display() {
        cout << "Object Number: " << this->number << endl;
        cout << "`this` Pointer Address: " << this << endl;
    }
};

int main() {
    MyClass obj1(100), obj2(200);
    obj1.display();
    obj2.display();
    obj2.setValue(10).display();
    return 0;
}
```

Output

```
Object Number: 100
`this` Pointer Address: 0x7fff8c4abd6c
Object Number: 200
`this` Pointer Address: 0x7fff8c4abd68
Object Number: 10
`this` Pointer Address: 0x7fff8c4abd68
```

Operator Overloading

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- Operator overloading in C++ allows us to define the behavior of operators for user-defined types, i.e., classes or structures, improving code readability and usability

- To overload an operator, we use a special function form called an **operator function**, i.e., operator*op* (*argument-list*). For example,
  - operator**+** (*argument-list*) // overload operator +
  - operator**<<** (*argument-list*) // overload operator <<
- Suppose you define an operator**+**() member function to overload operator **+** for a class named `Salesperson`, and `district2`, `sid` and `sara` are objects of `Salesperson` class, then

```
district2 = sid + sara;
district2 = sid.operator+(sara);
```

will be equivalent.

Operator Overloading

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

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

class MyClass {
private:
    int value;

public:
    MyClass(int v) {value = v; } // Constructor

    // Overloading the + operator
    MyClass operator+(const MyClass& obj) const {
          return MyClass(value + obj.value);
    }

    // Function sum() that does the same as operator+
    MyClass sum(const MyClass& obj) const {
        return MyClass(value + obj.value);
    }

    void display() { cout<< "Value: " <<value<<endl; }
};

int main() {
    MyClass obj1(10), obj2(20);

    MyClass obj3 = obj1 + obj2;  // Using overloaded + operator
    obj3.display();
    MyClass obj4 = obj1.sum(obj2); // Using sum() function
    obj4.display();

    return 0;
}
```

The overloaded operator+ is doing the same job as member function `sum()`!

You can also use the operator function:

```
MyClass obj3 =
obj1.operator+(obj2);
```

Output

```
Value: 30
Value: 30
```

Operator Overloading

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

class MyClass {
private:
    int value;

public:
    MyClass(int v) {value = v; } // Constructor

    // Overloading the + operator
    MyClass operator+(const MyClass& obj) const {
          return MyClass(value + obj.value);
    }

    // Function sum() that does the same as operator+
    MyClass sum(const MyClass& obj) const {
        return MyClass(value + obj.value);
    }

    void display() { cout<< "Value: " <<value<<endl; }
};

int main() {
    MyClass obj1(10), obj2(20);

    MyClass obj3 = obj1 + obj2;  // Using overloaded + operator
    obj3.display();
    MyClass obj4 = obj1.sum(obj2); // Using sum() function
    obj4.display();

    return 0;
}
```

`const` after the member function specifies that it is a "read-only" function that will not alter the object of the class.

Can we overload << to print the object content?

Output

```
Value: 30
Value: 30
```

Overloading <<

- Can we define the operator<<() function as a member function here?

  - Not appropriate!

- The statement like

  `cout << obj3;`

  uses two objects, and is equivalent to

  `operator<<(cout, obj3);` with the `ostream` class object `(cout)` first.

- If we define overload the operator << as a member function, the class object itself `obj3` will be the first parameter, which yields:

  `obj3 << cout;`

  which is quite confusing!

Friend and Friend Functions

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- In C++, private members of a class cannot be accessed directly outside the class. A `friend` in C++ provides access to private and protected members of another class. `Friends` come in three varieties:

  - Friend functions

  - Friend classes

  - Friend member functions

- By making a function a friend to a class, you allow the function the same access privileges that a member function of the class has.

Friend Functions

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

- Place a prototype in **the class declaration** and prefix the declaration with the keyword friend:

```
friend ostream& operator<<(ostream& out, const MyClass& obj);
```

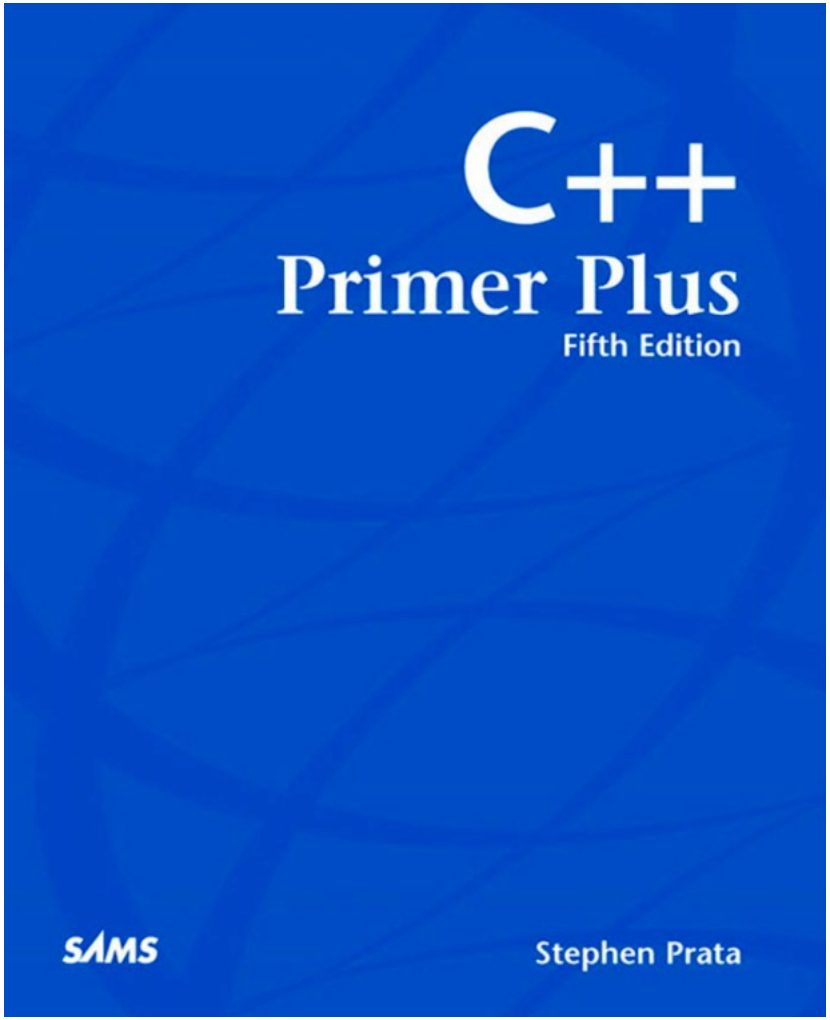
- Write the function definition without `MyClass::` qualifier

```
friend ostream& operator<<(ostream& out, const MyClass& obj)
{
    out << obj.value;
    return out;
}
```

Operator Overloading with Friend Function

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

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

class MyClass {
private:
    int value;
public:
    MyClass(int v) {value = v; }

    // Overloading the + operator
    MyClass operator+(const MyClass& obj) const {
        int sum = value + obj.value;
        return MyClass(sum);
    }

    // Declaring friend function to overload << operator
    friend ostream& operator<<(ostream& out, const MyClass& obj);
};

    // Overloading the << operator for output
ostream& operator<<(ostream& out, const MyClass& obj) {
        out << "value: " << obj.value;
        return out;
}

int main() {
    MyClass obj1(10), obj2(20);
    MyClass obj3 = obj1 + obj2;  // Using overloaded + operator
    cout << obj3 << endl;

    return 0;
}
```



References:

[1] Prata, Stephen. C++ primer plus. Sams Publishing, 2002, 5th edition.
Chapters 10, 11, 12.

Questions?

Thank You!