C++ Programming

Week 7: C++ Classes – Part II

Dr. Owen Chen
Cary Chinese School
Director of Math and Computer Science

2023 Summer

Week 7: Agenda

- **Review Week 6 – Classes – Part I**
  - Class declaration
  - Class definition
  - Public/Private access patterns
  - Makefile
- **Review Homework 6)**
- **New Topic: C++ Classes – Part II**
  - Class Constructor
  - Class Destructor
  - Class Hierarchy/Class Inheritance
  - Class Keywords

C++ Class Members - Data and Function Members

Data Member

Data within a class are called **data members** or **class fields**

Function Member

Functions within a class are called **function members** or **methods**

class Declaration and Definition

class declaration

```
class A;        // class declaration
```

class definition

```
class A {       // class definition
   int x;       // data member
   string name;
  void f(); // function member
};
```

Class Function Declaration and Definition

```
class A {
    void g();          // function member declaration

    void f() {         // function member declaration
        cout << "f"; // inline definition
    }
};

void A::g() {          // function member definition
    cout << "g";      // out side definition
}
```

class Members

```
class B {
    void g() { cout << "g"; } // function member
};

 class A {
    int  x;                   // data member
    B    b;                   // data member b is a class of B
    void f() { cout << "f"; } // function member
};

A a;
a.x;
a.f();
a.b.g();
```

C++ class Example: student_main.cpp

```
//Main Program – instantiate a Class and call member functions

#include "student.h"

int main(){
    Student st1;
    st1.setName(“John");
    st1.setBirthyear(2008);
    st1.setGender(‘m’);
    st1.printInfo();
    return 0;
}
```

C++ class Example: Student class

The source code can be saved into multiple files. Create a makefile to link them.

student.h

```
class Student
{
  private:
    string name;
    int birthyear;
    char gender;
  public:
    void setName(string s) // inline definition
    {
       name = s;
    }
    void setBirthyear(int y) // inline definition
    {
       birthyear = y;
    }
    void setGender(char g);
    void printInfo();
};
```

student.cpp

```
#include <iostream>
#include <cctype>
#include "student.h"

void Student::setGender(char g)
{
   gender = tolower(g);
}

void Student::printInfo()
{
   cout << "Name: " << name << endl;
   cout << "Born in year " << born << endl;
   cout << "Gender: " << (gender=='m'?
"Male":gender=='f'? "Female":"Other") << endl;
}
```

Student_main.cpp

```
#include "student.h"
int main(){
   Student st1;
   ...
   st1.printInfo();
   return 0;
}
```

Compile multiple dependent source code files

- When there are multiple C++ source code files, compile each cpp file into an object first with this syntax:
  - **g++ -c program1.cpp -o program1.o**
  - **g++ -c program2.cpp -o program2.o**

- Then link objects together:
  - **g++ program1.o program2.o -o program.exe**

Compile multiple dependent source code files

- In our student example:

```
$ g++ -c student.cpp –o student.o

$ g++ -c student_main.cpp –o student_main.o

$ g++ student.o student_main.o –o student.exe
```

Makefile is another method to compile multiple files

Create a makefile to compile multiple files

makefile

```
# executable files for this directory
OBJECTS = student.exe

# tells make to use the file "../makefile_template", which
# defines general rules for making .o and .exe files
include ../makefile_template

student.exe: student_main.o student.o
    $(CPP) $(CPPFLAGS) student_main.o student.o -o student.exe
```

Make file template for g++

Makefile_template

```
CPP = g++
CPPFLAGS = -std=c++20 -I..
LOCFLAGS =

all: $(OBJECTS)

%.o: %.cpp
    $(CPP) $(CPPFLAGS) $(LOCFLAGS) -c $< -o $@

%.exe: %.o
    $(CPP) $(CPPFLAGS) $(LOCFLAGS) $< -o $@

clean:
    rm -rf *.o *.obj core *.stackdump

clobber: clean
    rm -rf *.exe
```

Make file commands

The following commands can be used with this makefile:

```
$ make
$ make all
$ make clean
$ make clobber

$ make student.exe
```

Homework 6

1) What is the output of following C++ program? sizeof() is a standard C++ function to check the memory size (in bytes) of a variable.

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

class Empty {};

int main()
{
cout << sizeof(Empty);
return 0;
}
```

A) non-zero value
B) 0
C) Compiler error
D) Run time error

Homework 6

1) What is the output of following C++ program? sizeof() is a standard C++ function to check the memory size (in bytes) of a variable.

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

class Empty {};

int main()
{
cout << sizeof(Empty);
return 0;
}
```

A) non-zero value – Class Empty is a valid class and has a non-zero size
B) 0
C) Compiler error
D) Run time error

Homework 6

2) What is the output of following C++ program?

```
class Test {
    int x;
};
int main(){
    Test t;
    cout << t.x;
    return 0;
}
```

A) 0
B) A garbage value
C) Compiler error
D) Run time error

Homework 6

2) What is the output of following C++ program?

```
class Test {
    int x;
};
int main(){
    Test t;
    cout << t.x;
    return 0;
}
```

A) 0
B) A garbage value
C) Compiler error – by default, a data member is private in C++ Class
D) Run time error

Homework 6

```
3) Which of the following is true?

A) All objects of a class share all data members of class
B) Objects of a class do not share non-static members. Every object has its own
copy.
C) Objects of a class do not share codes of non-static methods - they have
their own copy.
D) None of the above
```

Homework 6

3) Which of the following is true?

A) All objects of a class share all data members of class
B) Objects of a class do not share non-static members. Every object has its own copy.
C) Objects of a class do not share codes of non-static methods - they have their own copy.
D) None of the above

Homework 6

4) Which of the following is true about the following program?

```
#include <iostream>
using namespace std;
class Test
{
public:
    int i;
    void get();
};
void Test::get()
{
  cout << "Enter the value of i: ";
  cin >> i;
}
Test t;  // Global object
int main()
{
  Test t;  // local object
  t.get();
  cout << "value of i in local t: "<<t.i<<'\n';
  ::t.get();
  cout << "value of i in global t: "<<::t.i<<'\n';
  return 0;
}
```

A) Compiler Error: Cannot have two objects with same class name.
B) Compiler error in Line "::t.get();"
C) Run time error.
D) Compiles and runs fine.

Homework 6

4) Which of the following is true about the following program?

```
#include <iostream>
using namespace std;
class Test
{
public:
    int i;
    void get();
};
void Test::get()
{
  cout << "Enter the value of i: ";
  cin >> i;
}
Test t;  // Global object
int main()
{
  Test t;  // local object
  t.get();
  cout << "value of i in local t: "<<t.i<<'\n';
  ::t.get();
  cout << "value of i in global t: "<<::t.i<<'\n';
  return 0;
}
```

A) Compiler Error: Cannot have two objects with same class name.
B) Compiler error in Line "::t.get();"
C) Run time error.
D) Compiles and runs fine.

Homework 6

5) A member function can always access the data in _____ (in C++).

A) the class of which it is member
B) the object of which it is a member
C) the public part of its class
D) the private part of its class

Homework 6

5) A member function can always access the data in _____ (in C++).

A) the class of which it is member – a member function can access both private and public members of the class
B) the object of which it is a member
C) the public part of its class
D) the private part of its class

Homework 6

6) Write a C++ class called "Factor" which includes the following attributes and methods:

- private attributes:
  - int n
  - vector<int> factors
- public methods:
  - read() get an integer from a user input and save to n
  - getFactors() get all factors of n and store them in factors.
  - printFactors() print all elements of vector factors. If factors is empty, call getFactors() to populate vector factors first.

Declare and define this class.

Test your class in a main program.

Class Constructor

Class Constructor

Constructor [ctor]

A **constructor** is a *special* member function of a class that is executed when a new instance of that class is created

Goals: *initialization* and *resource acquisition*

Syntax: `T(...)` same named of the class and no return type

- A *constructor* is supposed to initialize *all* data members
- We can define *multiple constructors* with different signatures

Default Constructor

The **default constructor** `T()` is a constructor with no argument

Every class has always either an *implicit* or *explicit* default constructor

```
class A {
    A()     {} // explicit default constructor
    A(int) {} // user-defined (non-default) constructor
};
```

```
class A {
    int x = 3; // implicit default constructor
};
A a{}; // ok
```

- An *implicit* default constructor is `constexpr`

Default Constructor Examples

```
class A {
    A() { cout << "A"; }  // default constructor
};

A a1;              // call the default constructor
// A a2();         // interpreted as a function declaration!!
A a3{};            // ok, call the default constructor
                   // direct-list initialization (C++11)

A array[3];        // print "AAA"

A* ptr = new A[4]; // print "AAAA"
```

Deleted Default Constructor

The *implicit* default constructor of a class is marked as **deleted** if (simplified):

- It has any user-defined constructor

```
class A {
    A(int x) {}
};
// A a; // compile error
```

- It has a non-static member/base class of reference/const type

```
class NoDefault { // deleted default constructor
    int&       x;
    const int  y;
};
```

- It has a non-static member/base class which has a deleted (or inaccessible) default constructor

```
class A {
    NoDefault var;        // deleted default constructor
};
class B : NoDefault {}; // deleted default constructor
```

- It has a non-static member/base class with a deleted or inaccessible destructor

```
class A {
private:
    ~A() {}
};
```

Initializer List

The **Initializer list** is used for *initializing the data members* of a class or explicitly call the base class constructor before entering the constructor body

(Not to be confused with `std::initializer_list`)

```
class A {
    int x, y;

    A(int x1) : x(x1) {}  // ": x(x1)" is the Initializer list
                          // direct initialization syntax

    A(int x1, int y1) :   // ": x{x1}, y{y1}"
        x{x1},            // is the Initializer list
        y{y1} {}          // direct-list initialization syntax
};                        // (C++11)
```

In-Class Member Initializer

C++11 **In-class non-static data members** can be initialized where they are declared (NSDMI). A constructor can be used when run-time initialization is needed

```
class A {
    int         x   = 0;       // in-class member initializer
    const char* str = nullptr; // in-class member initializer

    A() {} // "x" and "str" are well-defined if
           // the default constructor is called

    A(const char* str1) : str{str1} {}
};
```

Data Member Initialization

const and **reference** data members must be initialized by using the *initialization list* or by using in-class *brace-or-equal-initializer* syntax (C++11)

```
class A {
    int         x;
    const char  y;      // must be initialized
    int&        z;      // must be initialized

    int&        v = x; // equal-initializer (C++11)
    const int   w{4};  // brace initializer (C++11)

    A() : x(3), y('a'), z(x) {}
};
```

Initialization Order ⋆

Class members initialization follows the order of declarations and *not* the order in the initialization list

```
class ArrayWrapper {
    int* array;
    int  size;

    A(int user_size) :
        size{user_size},
        array{new int[size]} {}
        // wrong!!: "size" is still undefined
};

ArrayWrapper a(10);
cout << a.array[4]; // segmentation fault
```

C++ class Example with Constructors

```
class Student
{
  private:
    string name;
    int birthyear;
    char gender;
  public:
    Student()                    //default constructor
    {
      name = "";
      birthyear = 0;
      gender = 'u';
    }
    Student(string n, int y, char g)  //constructor
    {
      name = n;
      birthyear = y;
      gender = tolower(g);
    }
...
```

C++ class Example with Constructors

```
int main()
{
    Student st1; ;      // calling the default constructor
    cout << "Student 1 (default values):" << endl;
    st1.printInfo();
    st1.setName("John");
    st1.setBirthyear(2008);
    st1.setGender('M');
    cout << "Student 1:" << endl;
    st1.printInfo();

    Student st2("Tom", 2009, 'm'); // calling the non-default constructor
    cout << "Student 2:" << endl;
    st2.printInfo();

    Student st3("Emma", 2010, 'F'); // calling the non-default constructor
    cout << "Student 3:" << endl;
    st3.printInfo();
    return 0;
}
```

Copy Constructor

Copy Constructor

A **copy constructor** `T(const T&)` creates a new object as a *deep copy* of an existing object

```
class A {
    A()           {} // default constructor
    A(int)        {} // non-default constructor
    A(const A&) {} // copy constructor
}
```

- Every class always defines an *implicit* or *explicit* copy constructor
- Even the copy constructor implicitly calls the *default* Base class constructor
- Even the copy constructor is considered a non-default constructor

Copy Constructor Example

```
class Array {
    int  size;
    int* array;

    Array(int size1) : size{size1} {
        array = new int[size];
    }
    // copy constructor, ": size{obj.size}" initializer list
    Array(const Array& obj) : size{obj.size} {
        array = new int[size];
        for (int i = 0; i < size; i++)
            array[i] = obj.array[i];
    }
};
Array x{100}; // do something with x.array ...
Array y{x};   // call "Array::Array(const Array&)"
```

Copy Constructor Usage

The copy constructor is used to:

- Initialize one object from another one having the same type
  - Direct constructor
  - Assignment operator

```
A a1;
A a2(a1);     // Direct copy initialization
A a3{a1};     // Direct copy initialization
A a4 = a1;    // Copy initialization
A a5 = {a1}; // Copy list initialization
```

- Copy an object which is *passed by-value* as input parameter of a function

```
void f(A a);
```

- Copy an object which is returned as result from a function*

```
A f() { return A(3); } // * see RVO optimization
```

Copy Constructor Usage Examples

```
class A {
    A() {}
    A(const A& obj) { cout << "copy"; }
};

void f(A a) {} // pass by-value

A g() { return A(); };

A a;
A b = a;      // copy constructor (assignment)     "copy"
A c(b);       // copy constructor (direct)         "copy"
f(b);         // copy constructor (argument)       "copy"
g();          // copy constructor (return value)   "copy"
A d = g();    // * see RVO optimization           (depends)
```

C++ class Example with a Copy Constructor

```
class Student
{
  private:
    string name;
    int birthyear;
    char gender;
  public:
    Student()                    //default constructor
    {
      name = “unknown";
      birthyear = 0;
      gender = 'u';
    }
    Student(const Student &St)  //copy constructor
    {
      name = St.name;
      birthyear = St.birthyear;
      gender = St.gender;
    }
...
```

C++ class Example with a Copy Constructor

```
class Student
{
  private:
    string name;
    int birthyear;
    char gender;
  public:
    Student()                    //default constructor
    {
      name = “unknown";
      birthyear = 0;
      gender = 'u';
    }
    Student(const Student &St)  //copy constructor
    {
      name = St.name;
      birthyear = St.birthyear;
      gender = St.gender;
    }
...
```

C++ class Example with Constructors

```
int main()
{
    Student st1; ;      // calling the default constructor
    cout << "Student 1 (default values):" << endl;
    st1.printInfo();
    st1.setName("John");
    st1.setBirthyear(2008);
    st1.setGender('M');
    cout << "Student 1:" << endl;
    st1.printInfo();

    Student st2("Tom", 2009, 'm'); // calling the non-default constructor
    cout << "Student 2:" << endl;
    st2.printInfo();

    Student st4(st2);   // calling the copy constructor
    cout << "Student 4 (copied from Student 2):" << endl;
    st4.printInfo();
    return 0;
}
```

Class Destructor

Destructor

A **destructor** is a special member function that is invoked automatically whenever an object is going to be destroyed. Meaning, a destructor is the last function that is going to be called before an object is destroyed. Destructor release memory space occupied by the objects created by the constructor.

Goals: *resources releasing*

Syntax: `~T()` same name of the class and no return type

- Any object has exactly one *destructor*, which is always *implictly* or *explicitly* Declared
- If a destructor is not defined for a class, compiler will automatically create a default one.

```
class Array {
    int* array;

    Array() {    // constructor
        array = new int[10];
    }

    ~Array() {   // destructor
        delete[] array;
    }
};
int main() {
   Array a;      // call the constructor
   for (int i = 0; i < 5; i++)
       Array b; // call 5 times the constructor + destructor
}  // call the destructor of "a"
```

Class Hierarchy

Child/Derived Class or Subclass

A new class that inheriting variables and functions from another class is called a **derived** or **child** class

Parent/Base Class

The *closest* class providing variables and functions of a derived class is called **parent** or **base** class

Extend a *base class* refers to creating a new class which retains characteristics of the base class and *on top it can add* (and never remove) its own members

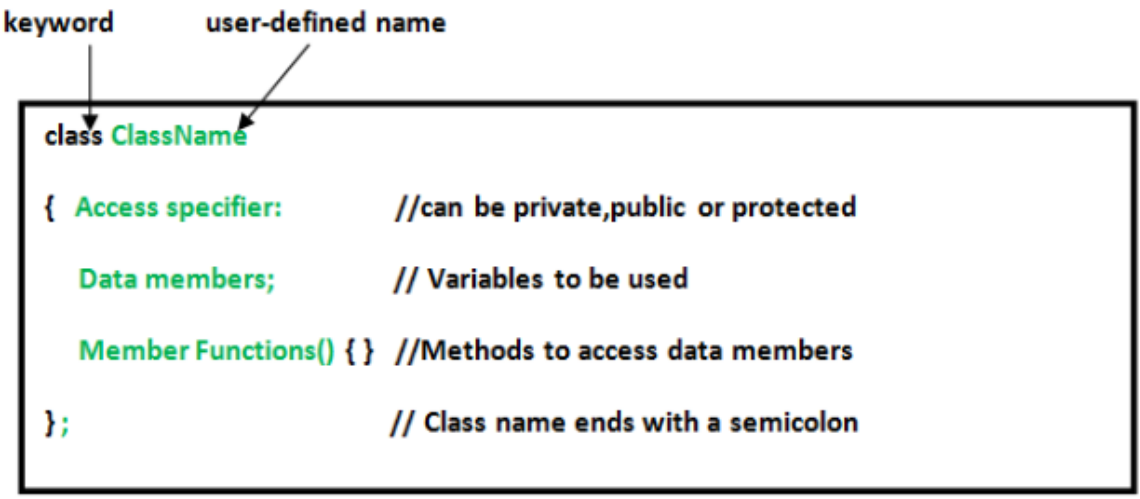
Syntax:

```
class DerivedClass : [<inheritance attribute>] BaseClass {
```

```
class A {           // base class
    int value = 3;
    void g() {}
};

class B : A {       // B is a derived class of A (B extends A)
    int data = 4;   // B inherits from A
    int f() { return data; }
};

A a;
B b;
a.value;
b.g();
```

```
class A {};
class B : A {};

void f(A a) {}        // copy
void g(B b) {}        // copy

void f_ref(A& a) {} // the same for A*
void g_ref(B& b) {} // the same for B*

A a;
B b;
f(a); // ok, also f(b), f_ref(a), g_ref(b)
g(b); // ok, also g_ref(b), but not g(a), g_ref(a)

A a1 = b;     // ok, also  A& a2 = b
// B b1 = a; // compile error
```

C++ class definition with access specifier



The **access specifiers** define the visibility of inherited members of the subsequent base class. The keywords `public`, `private`, and `protected` specify the sections of visibility

The goal of the *access specifiers* is to prevent a direct access to the internal representation of the class for avoiding wrong usage and potential inconsistency (access control)

- **public:** No restriction (*function members*, *derived classes*, *outside the class*)
- **protected:** *Function members* and *derived classes* access
- **private:** *Function members* only access (internal)

`struct` has default `public` members
`class` has default `private` members

```
class A1 {
public:
    int value; // public
protected:
    void f1() {} // protected
private:
    void f2() {} // private
};
class A2 {
    int data;    // private (by default)
};
class B : A1 {
   void h1() { f1(); } // ok, "f1" is visible in B
   //void h2() { f2(); } // compile error "f2" is private in A1
};

A1 a;
a.value; // ok
// a.f1() // compile error protected
// a.f2() // compile error private
```

The **access specifiers** are also used for defining how the visibility is propagated from the *base class* to a *specific derived class* in the inheritance

Member declaration		Inheritance		Derived classes
public protected private	→	**public**	→	public protected \
public protected private	→	**protected**	→	protected protected \
public protected private	→	**private**	→	private private \

```
class A {
public:
    int var1; // public
protected:
    int var2; // protected
};

class B : protected A {
public:
    int var3; // public
};

B b;
// b.var1; // compile error, var1 is protected in B
// b.var2; // compile error, var2 is protected in B
b.var3;    // ok, var3 is public in B
```

```
class A {
public:
    int var1;
protected:
    int var2;
};

class B1 : A {};          // private inheritance - default

class B2 : public A {}; // public inheritance B1 b1;
// b1.var1; // compile error, var1 is private in B1
// b1.var2; // compile error, var2 is private in B1



B2 b2;
b2.var1;     // ok, var1 is public in B2
```

Constructors and Inheritance

Class constructors are never inherited

A *Derived* class must call *implicitly* or *explicitly* a *Base* constructor before the current class constructor

Class constructors are called in order from the top Base class to the most Derived class (C++ objects are constructed like onions)

```
class A {
    A() { cout << "A" };
};
class B1 : A { // call "A()" implicitly
    int y = 3;  // then, "y = 3"
};
class B2 : A { // call "A()" explicitly
    B2() : A() { cout << "B"; }
};
B1 b1; // print "A"
B2 b2; // print "A", then print "B"
```

Class destructor is never inherited. *Base* class destructor is invoked *after* the current class destructor

Class destructors are called in reverse order. From the most Derived to the top Base class

```
class A {
    ~A() { cout << "A"; }
};
class B {
    ~B() { cout << "B"; }
};
class C : A {
    B b;                // call ~B()
    ~C() { cout << "C"; }
};
int main() {
    C b;  // print "C", then "B", then "A"
}
```

Class Keywords

this Keyword

this

Every object has access to its own address through the const pointer `this`

Explicit usage is not mandatory (and not suggested)

`this` is necessary when:

- The name of a local variable is equal to some member name
- Return reference to the calling object

```
class A {
    int x;
    void f(int x) {
        this->x = x; // without "this" has no effect
    }
    const A& g() {
        return *this;
    }
};
```

`this` Pointer Example: this.cpp

```
Student(const string name, int birthyear, char gender)
{
    this->name = name;
    this->birthyear = birthyear;
    this->setGender(gender);
    cout << "Constructor: Student(const string,int,char)" << endl;
}

void setName(const string name)
{
    this->name = name;
}
void setBirthyear(int birthyear)
{
    this->birthyear = birthyear;
}
```

Const member functions

Const member functions (**inspectors** or **observer**) are functions marked with `const` that are not allowed to change the object state

Member functions without a `const` suffix are called *non-const member functions* or **mutators**. The compiler prevents from inadvertently mutating/changing the data members of *observer* functions

```
class A {
    int x = 3;

    int get() const {
     // x = 2;     // compile error class variables cannot be modified
        return x;
    }
};
```

The `const` keyword is part of the functions signature. Therefore a class can implement two similar methods, one which is called when the object is `const`, and one that is not

```
class A {
    int x = 3;
public:
    int& get1()       { return x; } // read and write
    int  get1() const { return x; } // read only
    int& get2()       { return x; } // read and write
};

A a1;
cout << a1.get1();     // ok
cout << a1.get2();     // ok
a1.get1() = 4;         // ok
const A a2;
cout << a2.get1();     // ok
// cout << a2.get2(); // compile error "a2" is const
//a2.get1() = 5;      // compile error only "get1() const" is available
```

friend Class

A `friend` class can access the `private` and `protected` members of the class in which it is declared as a `friend`

Friendship properties:

- **Not Symmetric**: if class **A** is a friend of class **B**, class **B** is not automatically a friend of class **A**
- **Not Transitive**: if class **A** is a friend of class **B**, and class **B** is a friend of class **C**, class **A** is not automatically a friend of class **C**
- **Not Inherited**: if class **Base** is a friend of class **X**, subclass **Derived** is not automatically a friend of class **X**; and if class **X** is a friend of class **Base**, class **X** is not automatically a friend of subclass **Derived**

```
class B;   // class declaration

class A {
    friend class B;
    int x;       // private
};

class B {
    int f(A a) { return a.x; } // ok, B is friend of A
};

class C : B {
//  int f(A a) { return a.x; } // compile error not inherited
};
```

friend Method

A *non-member function* can access the `private` and `protected` members of a class if it is declared a `friend` of that class

```
class A {
    int x = 3;  // private

    friend int f(A a); // friendship declaration, no implementation
};

// 'f' is not a member function of any class
int f(A a) {
    return a.x;  // A is friend of f(A)
}
```

`friend` methods are commonly used for implementing the stream operator `operator<<`

Summary

- **Classes** are the most fundamental feature in C++. Classes let us define new types for our applications, making our programs shorter and easier to modify.
- **Data abstraction**—the ability to define both **data** and **function** members.
- **Encapsulate** a class by defining its implementation members as **private**.
- Classes may grant access to their nonpublic member by designating another class or function as a **friend**.
- Classes may define **constructors**, which are special member functions that control how objects are initialized. Constructors may be **overloaded**.
- Classes may define a single **destructor**, which is a special member function that releases memory when an object is destroyed.