C++ Programming

Week 6: C++ Classes

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

2023 Summer

Week 6: Agenda

- Review Week 5 - Functions
- Review Homework 5)
- New Topic: C++ Classes

Function Review

A **function** (**procedure** or **routine**) is a piece of code that performs a *specific task.* Function is a block of code which only runs when it is called.

Purpose:

- **Avoiding code duplication**: less code for the same functionality → less bugs
- **Readability**: better express what the code does
- **Organization**: break the code in separate modules

Function Review

- Function is a block of code with a name.
- Declare a function with its name, parameters and return type
- Define a function with details
- Execute a function by calling the function
- A function takes zero or more arguments and usually returns a result.
- Functions can be overloaded, meaning that the same name may have different arguments and different return values

Function Declaration/Definition

Declaration/Prototype

A **declaration** (or *prototype*) of an entity is an identifier describing its type

A declaration is what the compiler and the linker needs to accept references (usage) to that identifier

C++ entities (class, functions, etc.) can be declared multiple times (with the same signature)

Definition/Implementation

An entity **definition** is the implementation of a declaration

For each entity, only a single *definition* is allowed

Homework 5

1) Indicate which of the following functions are in error and why. Suggest how you might correct the problems.

```
(a) int f() {
    string s;
    // . . .
    return s;
    }
(b) f2(int i) { /* . . . */ }
(c) int calc(int v1, int v1) /* . . . */ }
(d) double square(double x) return x * x;
```

Homework 5

1) Indicate which of the following functions are in error and why. Suggest how you might correct the problems.

```
(a) int f() {.
    string s;
    // . . .
    return s;
    }

(b) f2(int i) { /* . . . */ }
(c) int calc(int v1, int v1) /* . . . */ }
(d) double square(double x) return x * x;
```

Solutions to 1):

(a) Return type and the actual return value mismatch.
(b) No return type. Add a void keyword.
(c) Two input arguments are both named v1
(d) Missing {}

Homework 5

2) Assuming T is the name of a type, explain the difference between a function declared as

```
    void f(T)
```

and

```
    void f(T&)
```

Homework 5

```
2) Assuming T is the name of a type, explain the difference
between a function declared as
    void f(T)
and
    void f(T&)
```

Solutions to 2):
f(T) is a function that passes its argument by value.
f(T&) is a function that passes its argument by reference.

Homework 5

3) Explain the behavior of the following function. If there are problems in the code, explain what they are and how you might fix them.

```
void print(const int ia[10])
{
    for (size_t i = 0; i != 10; ++i)
        cout << ia[i] << endl;
}
```

Homework 5

Solutions to 3):

The function **print** takes as an argument an array of 10 constant integers. The function then iterates through the array using a for loop and prints each element of the array to the standard output stream, followed by a newline character.

However, there is a problem with the function's parameter declaration. The parameter const int ia[10] is misleading because it suggests that the function will only accept arrays of size 10. In reality, the parameter is equivalent to const int* ia, which means that the function will accept a pointer to a constant integer. This can lead to problems if an array of a different size is passed to the function, as the function will still try to access 10 elements, potentially causing undefined behavior.

One way to fix this issue is to change ia to a pointer and add a size of the ia array:

```
void print(const int ia[], int size)
{
    for (size_t i = 0; i < size; ++i)
        cout << ia[i] << endl;
}
```

Homework 5

4) Given the following declarations, determine which calls are legal and which are illegal. For those that are illegal, explain why.

```
double calc(double);
int count(const string &, char);
int sum(vector<int>::iterator, vector<int>::iterator, int);
vector<int> vec(10);
(a) calc(23.4, 55.1);
(b) count("abcda", 'a');
(c) calc(66);
(d) sum(vec.begin(), vec.end(), 3.8);
```

Homework 4

4) Given the following declarations, determine which calls are legal and which are illegal. For those that are illegal, explain why.

```
double calc(double);
int count(const string &, char);
int sum(vector<int>::iterator, vector<int>::iterator, int);
vector<int> vec(10);
(a) calc(23.4, 55.1);
(b) count("abcda", ’a’);
(c) calc(66);
(d) sum(vec.begin(), vec.end(), 3.8);
```

Solutions to 4)

a) **Illegal. Pass two values to a function has a single parameter.**
b) **Legal. Pass a string literal to a string reference is fine.**
c) **Legal. Convert 66 to a double as a parameter value in calc()**
d) **Legal. 3.8 is converted to integer. However, the outcome is random as vec is not initialized.**

Homework 5

```
5) Write a function that will calculate all factors of an integer

    function name: factor()
    input: int n
    output: a vector of integers with each element being a unique factor of n
```

Homework 5

6) Write a function that will calculate the GCD (greatest common divisor) of two integers. Use factor() function defined in 5)

```
function name: gcd()
input: int a, int b
output: gcd of (a, b)
```

7) Write a function that determines whether an input integer is a prime or not.

```
function name: isPrime()
input: int
output: bool
```

8) Write a program that will produce a list of prime numbers that are less than a given input integer. Use isPrime() function in 7) if needed.

```
function name: prime_list()
input: int n
output: a list of prime numbers
```

Main program: prompt user to enter a number and store it as integer n.
Call prime_list() and print the list of primes on screen.

C++ Classes

C++ Classes and Objects

- C++ is an object-oriented programming language.
- Everything in C++ is associated with classes and objects, along with its attributes and methods.
- For example: a car is an **object**.
  - **attributes**, such as weight and color
  - **methods**, such as drive and brake.
- Attributes and methods are basically **variables** and **functions** that belongs to the class. These are often referred to as "**class members**".

C/C++ Structure

Before C++, C has a **structure** (`struct`) is a collection of variables of the same or different data types under a single name

Structures

A `struct` in **C** is a type consisting of a sequence of data members.

Some functions/statements are needed to operate the data members of an object of a `struct` type.

```
struct Student
{
    char name[20];
    int born;
    bool male;
};

struct Student stu;
strcpy(stu.name, “John");
stu.born = 2000;
stu.male = true;
```

C++ Classes

- In C++, we define our own data structures by defining a **class**.
- A class defines a type along with a collection of operations that are related to that type.
- The class mechanism is one of the most important features in C++.

C++ Classes

- Classes are an expanded concept of data structures: like data structures, they can contain data members, but they can also contain functions as members.

- An object is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

- Classes are defined using either keyword class or keyword struct.

C++ Classes

To use a class, we need to know three things:

- What is its name?
- Where is it defined?
- What operations does it support?

Struct vs Class

C/C++ Structure

A **structure** (`struct`) is a collection of variables of the same or different data types under a single name

C++ Class

A **class** (`class`) extends the concept of structure to hold functions as members

`struct` vs. `class`

Structures and classes are *semantically* equivalent. In general, `struct` represents *passive* objects, while `class` *active* objects

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**

struct Declaration and Definition

struct declaration

```
struct A;     // struct declaration
```

struct definition

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

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: firstclass.cpp

```
class Student
{
  public:                                        //everything is public
    string name;
    int birthyear;
    char gender;
    void setName(const string s)
    {   name = s;    }
    void setBirthyear(int b)
    {   birthyear = b; }
    void setGender(char c)
    {  gender = c;  }
    void printInfo()
    {
        cout << "Name: " << name << endl;
        cout << "Born in " << birthyear << endl;
        cout << "Gender: " << gender << endl;
    }
};
```

C++ class Example: student.h

```
class Student
{
  private:
    string name;                          // variables are privates
    int birthyear;
    char gender;
  public:                                 // functions are public
    void setName(string s)
    {
      name = s;
    }
    void setBirthyear(int b)
    {
      birthyear = b;
    }
    // the declarations, the definitions are out of the class
    void setGender(char gender);
    void printInfo();
};
```

C++ class Example: student.cpp

```
//Class Definitions

#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;
}
```

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;
}
```

Source Code Management

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;
}
```

Makefile

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
```

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 <u>no argument</u>

Every class has <u>always</u> 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"
```

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;
};
```

Deleted Default Constructor

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

```cpp
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

```cpp
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

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)
```

Deleted Copy Constructor

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

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

```
class NonDefault { int& x; }; // deleted copy constructor
```

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

```
class B { // deleted copy constructor
    NonDefault a;
};
class B : NonDefault {}; // delete copy constructor
```

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

- The class has the move constructor (next lectures)

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: $\sim\mathbf{T()}$ 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();
```

```cpp
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

keyword   user-defined name

```
class ClassName

{  Access specifier:             //can be private,public or protected

   Data members;                 // Variables to be used

   Member Functions() { }        //Methods to access data members

};                               // Class name ends with a semicolon
```

Access specifiers

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 {
    int var1; // public
protected:
    int var2; // protected
};

  class B : protected A {
    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

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 <u>never</u> inherited

A *Derived* class <u>must</u> call *implicitly* or *explicitly* a *Base* constructor <u>before</u> the current class constructor

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

```cpp
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 <u>never</u> 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;
    }
};
```

const Keyword - const Overloading

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
```

mutable Keyword

mutable

`mutable` members of *const* class instances are modifiable

Constant references or pointers to objects cannot modify objects in any way, except for data members marked `mutable`

- It is particularly useful if most of the members should be constant but a few need to be modified
- *Conceptually,* `mutable` *members should not change anything that can be retrieved from the class interface*

```
class A {
    int         x = 3;
    mutable int y = 5;
};
const A a;
// a.x = 3; // compiler error const
a.y = 5;    // ok
```

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.