

Program Design

-- Constructors & Destructors

Junjie Cao @ DLUT

Summer 2022

<https://github.com/jjcao-school/c>

Constructors

构造函数

Constructors

```
class Foo{  
public:  
    int m_x;  
    int m_y;  
};
```

```
int main(){  
    Foo foo1 = { 4, 5 }; // initialization list  
    Foo foo2 { 6, 7 }; // uniform initialization (C++11)  
    return 0;  
}
```

However, as soon as we make any member variables private, we're no longer able to initialize classes in this way.

It does make sense: if you can't directly access a variable (because it's private), you shouldn't be able to directly initialize it. => constructor

Constructors 构造函数

- A **constructor** is a special kind of class member function that is automatically called when an object of that class is instantiated.
 - Constructors should always have the **same name** as the class (with the same capitalization)
 - Constructors have **no return** type (not even void)

Default constructors

- A constructor that takes no parameters (or has parameters that all have default values)

```
class Fraction{
private:
    int m_numerator;  int m_denominator;
public:
    Fraction(){ // default constructor
        m_numerator = 0;
        m_denominator = 1;
    }
    int getNumerator() { return m_numerator; }
};

Fraction frac; // Since no arguments, calls Fraction() default constructor
std::cout << frac.getNumerator() << "/" << frac.getDenominator() << '\n';
```

Direct and uniform initialization using constructors with parameters

public:

```
Fraction(){ // default constructor
```

```
    m_numerator = 0;
```

```
    m_denominator = 1;
```

```
}
```

// Constructor with two parameters, one parameter having a default value

```
Fraction(int numerator, int denominator=1){
```

```
    assert(denominator != 0);
```

```
    m_numerator = numerator;
```

```
    m_denominator = denominator;
```

```
}
```

```
int x(5);
```

```
Fraction fiveThirds(5, 3);
```

```
Fraction six(6);
```

Reducing your constructors

```
Fraction(){ // default constructor
```

```
    m_numerator = 0;  m_denominator = 1;
```

```
}
```

```
Fraction(int numerator, int denominator=1){
```

```
    assert(denominator != 0);
```

```
    m_numerator = numerator;
```

```
    m_denominator = denominator;
```

```
}
```



```
Fraction(int numerator=0, int denominator=1){
```

```
    assert(denominator != 0);
```

```
    m_numerator = numerator;
```

```
    m_denominator = denominator;
```

```
}
```

Reducing your constructors

```
Fraction(int numerator=0, int denominator=1){  
    assert(denominator != 0);  
    m_numerator = numerator;  
    m_denominator = denominator;  
}
```

Fraction default; // will call Fraction(0, 1)

Fraction six(6); // will call Fraction(6, 1)

Fraction fiveThirds(5,3); // will call Fraction(5, 3)

Classes without default constructors

```
class Date{  
private:  
    int m_year;    int m_month;    int m_day;  
    // No default constructor provided, so C++ creates an empty one for us  
    // Because no other constructors exist, this provided constructor will be public  
};
```

```
Date date; // calls default constructor that does nothing  
// date's member variables are uninitialized  
// Who knows what date we'll get?
```

- if you do **have other non-default constructors** in your class, but no default constructor, C++ will **not create an empty default constructor for you**

```
class Date{
```

```
private:  int m_year;  int m_month;  int m_day;
```

```
public:
```

```
    Date(int year, int month, int day){ // not a default constructor
```

```
        m_year = year;  m_month = month;  m_day = day; }
```

```
    // No default constructor provided
```

```
};
```

```
Date date; // error: Can't instantiate object because default constructor doesn't exist
```

```
Date today(2020, 10, 14); // today is initialized to Oct 14th, 2020
```

Constructors in Python

```
class Cat
```

```
    def getHumanAge(self):  
        return self._age
```

```
...
```

```
class Cat:
```

```
    def __init__(self, age=0):  
        self._age = age
```

```
    def getHumanAge(self):  
        return self._age
```

Quiz time - Write a class named Ball.

- Ball should have two private member variables with default values: m_color ("Black") and m_radius (10.0).
- Ball should provide constructors to set only m_color, set only m_radius, set both, or set neither value.
- //do not use default parameters for your constructors.
- Also write a function to print out the color and radius of the ball.
- The following sample program should compile:

```
Ball def; def.print();
```

```
Ball blue("blue"); blue.print();
```

```
Ball twenty(20.0); twenty.print();
```

```
Ball blueTwenty("blue", 20.0); blueTwenty.print();
```

color: black, radius: 10

color: blue, radius: 10

color: black, radius: 20

color: blue, radius: 20

Quiz 2

- Update your answer to the previous question to use constructors with default parameters. Use as few constructors as possible.

Quiz 3

- What happens if you don't declare a default constructor?
 - If you haven't defined any other constructors, the compiler will create an empty public default constructor for you.
 - This means your objects will be instantiable with no parameters.
 - If you have defined other constructors (default or otherwise), the compiler will not create a default constructor for you.
 - Assuming you haven't provided a default constructor yourself, your objects will not be instantiable with no parameters.

Constructor member initializer lists

```
class Something{
```

```
private:
```

```
    int m_value1;    char m_value3;
```

```
public:
```

```
    Something()
```

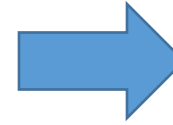
```
{
```

```
    // These are all assignments, not initializations
```

```
    m_value1 = 1.0;    m_value3 = 'c';
```

```
}
```

```
};
```



```
int m_value1;
```

```
double m_value2;
```

```
char m_value3;
```

```
m_value1 = 1.0;
```

```
m_value2 = 2.2;
```

```
m_value3 = 'c';
```

Constructor member initializer lists

```
class Something{  
private:  
    const int m_value;  
public:  
    Something(){  
        m_value = 1; // error: const vars can not be assigned to  
    }  
};
```

```
const int m_value; // error: const vars must be initialized with a value  
m_value = 5; // error: const vars can not be assigned to
```


Member initializer lists

```
class Something{  
private:  
    int m_value1;  
    double m_value2;  
    char m_value3;  
public:  
    Something() : m_value1(1), m_value2(2.2), m_value3('c')  
        // directly initialize our member variables  
    {  
        // No need for assignment here  
    }  
}
```

Overlapping and delegating constructors

class Foo

{

public:

Foo(){

// code to do A

}

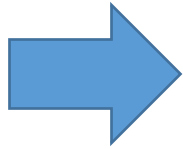
Foo(int value){

// code to do A

// code to do B

}

};



Using a separate function

class Foo{

private:

void DoA(){ // code to do A }

public:

Foo(){ DoA(); }

Foo(int nValue){

DoA();

// code to do B

}

};

code duplication is kept to a minimum.

you may find yourself in the situation where you want to write a member function to re-initialize a class back to default values.

```
class Foo{  
public:  
    Foo(){ Init(); }  
  
    Foo(int value){ Init();  
        // do something with value  
    }  
  
    void Init() { // code to init Foo }  
};
```

Delegating constructors in C++11

```
class Employee{
```

```
private:
```

```
    int m_id;    std::string m_name;
```

```
public:
```

```
    Employee(int id, std::string name):
```

```
        m_id(id), m_name(name) { }
```

```
// All three of the following constructors use delegating constructors to minimize redundant code
```

```
    Employee() : Employee(0, "") { }
```

```
    Employee(int id) : Employee(id, "") { }
```

```
    Employee(std::string name) : Employee(0, name) { }
```

```
};
```

Destructors

析构函数

Destructors

- A **destructor** is another special kind of class member function that is executed when an object of that class is destroyed.

- **Destructor naming**

- 1) The destructor must have the same name as the class, preceded by a tilde (~).
- 2) The destructor can not take arguments.
- 3) The destructor has no return type.

```
class MyString
{
    ~MyString() { // destructor
        delete[] m_string;
    }
}
```

- only one destructor may exist per class
- like constructors, destructors should not be called explicitly
- destructors may safely call other member functions since the object isn't destroyed until after the destructor executes.

Constructor and destructor timing

```
class Simple{  
private:  int m_nID;  
public:  
    Simple(int nID) {  
        std::cout << "Constructing Simple " << nID << '\n';  
        m_nID = nID;  
    }  
  
    ~Simple(){std::cout << "Destructing Simple" << m_nID << '\n';}  
    int getID() { return m_nID; }  
};
```

Constructor and destructor timing

```
int main(){  
    // Allocate a Simple on the stack  
    Simple simple(1);  
    std::cout << simple.getID() << '\n';  
  
    // Allocate a Simple dynamically  
    Simple *pSimple = new Simple(2);  
    std::cout << pSimple->getID() << '\n';  
    delete pSimple;  
  
    return 0;  
} // simple goes out of scope here
```

Constructing Simple 1
1
Constructing Simple 2
2
Destructing Simple 2
Destructing Simple 1

A warning about the `exit()` function

- if you use the `exit()` function, your program will terminate and no destructors will be called.
- Be wary if you're relying on your destructors to do necessary cleanup work (e.g. write something to a log file or database before exiting)

Destructors in Python

```
class Cat:
    def __init__(self, age=0):
        self._age = age
    def __del__(self):
        # body of a destructor
        pass
```

- The `__del__()` method will be implicitly invoked when all references to the object have been deleted,
 - i.e., is when an object is eligible for the garbage collector.

this vs self

a hidden pointer named “this”

- “When a member function is called, how does C++ keep track of which object it was called on?”

- `simple.setID(2);`



- `setID(&simple, 2);` // note that `simple` has been changed from an object prefix to a function argument!

- `void setID(int id) { m_id = id; }`



- `void setID(Simple* const this, int id) { this->m_id = id; }`

self in Python Class

```
class Cat(object):  
    def __init__(self, age=0):  
        self._age = age
```

```
    def getHumanAge(self):  
        return self._age
```

```
    def setHumanAge(self, value):  
        self._age = value
```

```
    def getAge(self):  
        return self._age * 7
```

```
    def setAge(self, value):  
        self._age = value / 7
```

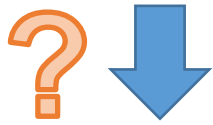
```
if __name__ == '__main__':  
    c = Cat(age=5)  
    print(c.getHumanAge())  
    print(c.getAge())
```

Chaining objects

```
class Calc{  
private: int m_value;  
  
public:  
    Calc() { m_value = 0; }  
  
    void add(int value) { m_value += value; }  
    void sub(int value) { m_value -= value; }  
    void mult(int value) { m_value *= value; }  
  
    int getValue() { return m_value; }  
};
```

Chaining objects

- `Calc calc;`
- `calc.add(5); // returns void`
- `calc.sub(3); // returns void`
- `calc.mult(4); // returns void`
- `std::cout << calc.getValue() << '\n';`



- `calc.add(5).sub(3).mult(4);`

```
Calc& add(int value) { m_value += value; return *this; }  
Calc& sub(int value) { m_value -= value; return *this; }  
Calc& mult(int value) { m_value *= value; return *this; }
```

Head file & source file

.h & .cpp

Class code and header files

- Defining member functions outside the class definition

```
#ifndef DATE_H
```

```
#define DATE_H
```

```
class Date{
```

```
private:  int m_year;  int m_month; ...
```

```
public:
```

```
    Date(int year, int month, int day);
```

```
    void SetDate(int year, int month, int day);
```

```
    int getYear() { return m_year; }; ...
```

```
#endif
```

Class code and header files

- Date.cpp:

```
#include "Date.h"
```

```
Date::Date(int year, int month, int day){  
    SetDate(year, month, day);  
}
```

```
void Date::SetDate(int year, int month, int day){  
    m_month = month;  
    m_day = day;  
    m_year = year;  
}
```

a couple of downsides to expose implementation

- First, your class implementation code will be **copied into every file** that `#includes` it, and get recompiled there.
 - This can **be slow**, and will cause bloated file sizes.
- Second, if you **change anything about the code in the header**, then you'll need to **recompile every file that includes that header**.
 - This can have a ripple effect, where one minor change causes the entire program to need to recompile (which can be slow).
 - If you change the code in a `.cpp` file, only that `.cpp` file needs to be recompiled!
- **Default parameters**
 - Default parameters for member functions should be declared in the class declaration (in the **header** file), where they can be **seen** by whomever `#includes` the header.

Libraries

- Separating the class declarations and class implementation is very common for libraries that you can use to extend your program.
- `#included` `iostream`, `string`, ...
- No need to add `iostream.cpp`, `string.cpp` into your projects.
- the implementations for the classes that belong to the C++ standard library is contained in a precompiled file that is linked in at the link stage.

Libraries

- most 3rd party libraries provide only header files, along with a precompiled library file.
- reasons for this
 - 1) It's faster to link a precompiled library than to recompile it every time
 - 2) a precompiled library can be distributed once, whereas compiled code gets compiled into every executable that uses it (inflating file sizes)
 - 3) intellectual property reasons (you don't want people stealing your code).

Class - summary

- Encapsulation: properties and functions
- Constructors: default, non-default, system generated
 - member initializer lists
- Destructor
- This

Quiz time

1a) Write a class named Point2d. Point2d should contain two member variables of type double: m_x, and m_y, both defaulted to 0.0. Provide a constructor and a print function.

- The following program should run:

```
int main()
{
    Point2d first;
    Point2d second(3.0, 4.0);
    first.print();
    second.print();

    return 0;
}
```

1b) Now add a member function named distanceTo.

Given two points (x_1, y_1) and (x_2, y_2) , the distance between them can be calculated as $\text{sqrt}((x_1 - x_2)^2 + (y_1 - y_2)^2)$.

The sqrt function lives in header cmath.

The following program should run:

```
Point2d first;
```

```
Point2d second(3.0, 4.0);
```

```
first.print();
```

```
second.print();
```

```
std::cout << "Distance between two points: " << first.distanceTo(second) << '\n';
```


- 1c) Change function distanceTo from a member function to a non-member friend function that takes two Points as parameters. Also rename it “distanceFrom”.
- The following program should run:

```
int main(){
    Point2d first;
    Point2d second(3.0, 4.0);
    first.print();
    second.print();
    std::cout << "Distance between two points: " << distanceFrom(first, second)
    << '\n';

    return 0;
}
```

<http://www.learncpp.com/cpp-tutorial/8-15-chapter-8-comprehensive-quiz/>

- 3) Let's create a random monster generator
- 4) rewrite the Blackjack games using classes!

Object-Based vs Object-Oriented programming

- Object-Based: **Encapsulation** (define composite datatypes using classes: fields + methods)
- Object-Oriented:
 - Encapsulation
 - **Inheritance**: reusing code between related types
 - **Polymorphism**: determining at runtime which functions to call on it based on its type

