Class, Function Overloading, and Constructor/Destructor

Jinsoo Jang

Classes

Classes

- Class is user-defined type
- Specifies followings
  - How objects of class are represented
  - Operations that can be performed on objects of class

Class Members

- Class consists of zero or more members
- Three basic kinds of members (excluding enumerators):
  - Data member
    - Define representation of class object
  - Function member
    - Also called member functions, provide operations on such objects
  - Type member
    - Specify any types associated with class

Access Specifiers

- Control level of access to class members
  - Public
    - Member can be accessed by any code
  - Private
    - Member can only be accessed by other members of class and friends of class
  - Protected
    - Related to inheritance

Example of Class

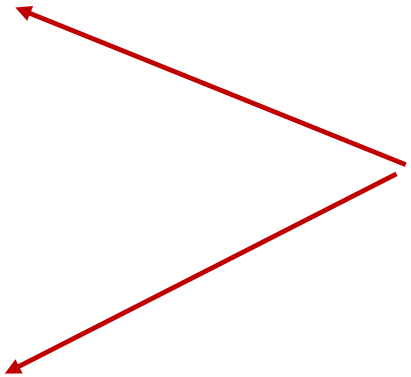
- Typical form of class

```
class Widget // The class is named Widget.
{
public:
// public members
// (i.e., the interface to users)
// usually functions and types (but not data)
private:
// private members
// (i.e., the implementation details only
// accessible by members of class)
// usually functions, types, and data
};
```

struct Keyword

- ***struct*** is also class
  - members of struct are public by default
- Example

```
struct Employee {
    //… members …
};

class Employee {
    public:
    // …
};
```



Data Members

- Data members x and y
- Members accessed by "." operator

```
class Vector_2 { // Two-dimensional vector class.
public:
    double x; // The x component of the vector.
    double y; // The y component of the vector.
};
void func() {
    Vector_2 v;
    v.x = 1.0; // Set data member x to 1.0
    v.y = 2.0;  // Set data member y to 2.0
}
```

Function Members

- Example code
  - Member function: initialize
  - scope-selection operator (::)
    - E.g., Vector_2::initialize

```
class Vector_2 { // Two-dimensional vector class.
public:
    void initialize(double newX, double newY);
    double x; // The x component of the vector.
    double y; // The y component of the vector.
};
void Vector_2::initialize(double newX, double newY) {
    x = newX; // "x" means "this->x"
    y = newY; // "y" means "this->y"
}
void func() {
    Vector_2 v; // Create Vector_2 called v.
    v.initialize(1.0, 2.0); // Initialize v to (1.0, 2.0).
}
```

Declaration of function

Definition of function

const Member Functions

❖ member function indicated with ***const*** cannot change value of object

```
1   #include <iostream>
2   class Counter {
3   public:
4     int getCount() const
5     {    return count;  } // count means this->count
6     void setCount(int newCount)
7     {     count = newCount;   } // count means this-> count
8     void incrementCount () const
9     {     ++count;   } // count means this->count
10  private:
11    int count; // counter value
12  };
13
14  int main() {
15    Counter ctr;
16    ctr.setCount(0);
17    int count = ctr.getCount();
18    const Counter& ctr2 = ctr;
19    count = ctr2.getCount(); // get Count better be const!
20    return 0;
21  }
```

```
clang++-7 -pthread -std=c++17 -o main main.cpp
main.cpp:9:9: error: cannot assign to non-static
      data member within const member function
      'incrementCount'
  {     ++count;   } // count means this->count
        ^ ~~~~~
main.cpp:8:8: note: member function
      'Counter::incrementCount' is declared const
      here
  void incrementCount () const
  ~~~~~^~~~~~~~~~~~~~~~~~~~~~~
1 error generated.
compiler exit status 1
```

Type Members

❖ Equivalent expressions

- using Coordinate = double
- typedef double Coordinate;

```
#include <iostream>

class Point_2 { // Two-dimensional point class.
public:
    using Coordinate = double; // Coordinate type.
    Coordinate x; // The x coordinate of the point.
    Coordinate y; // The y coordinate of the point.

    Point_2( ) :x(1), y(2) {};
};
void main( ) {
    Point_2 p;
    // ...
    Point_2::Coordinate x = p.x;
    // Point_2::Coordinate same as double
    std::cout <<p.x << std::endl;
    getchar( );
}
```

friend Keyword

- Only class has access to its private members
- ***friend*** of class is function/class that is allowed to access private members of class
- Example

```
class Gadget; // forward declaration of Gadget

class Widget {
// ...
    friend void myFunc();
    // function myFunc is friend of Widget
    friend class Gadget;
    // class Gadget is friend of Widget
// ...
};
```

- Generally, use of friends should be avoided except when absolutely necessary

Example with *friend*

```
1   #include <iostream>
2   class Information {
3   public:
4     int updateValue(int newValue) { //member function
5       int oldValue = value;
6       value = newValue;
7       return oldValue;
8     }
9   private:
10    friend void friendAccess();
11    void tempFunction(){};
12    int value;
13  };
14  void friendAccess() {
15    Information info;
16    info.tempFunction(); // OK with friend keyword
17    info.value = 14; // Ok with friend keyword
18  }
19  void notFriendAccess() {
20    Information info;
21    info.updateValue(14); // OK
22    info.tempFunction(); //Error: tempFunction is private
23    info.value = 14;  //Error: value is private
24  }
25  int main(){
26  //do nothing
27  return 0;
28  }
```

Using ***friend*** keyword enables to access the every member of Information class

Note: The effect of friend keyword is not bidirectional!

Only friendAccess() → Information class is allowed

Function Overloading

Function Overloading

- Multiple functions can have same name as long as they differ in number/type of their arguments

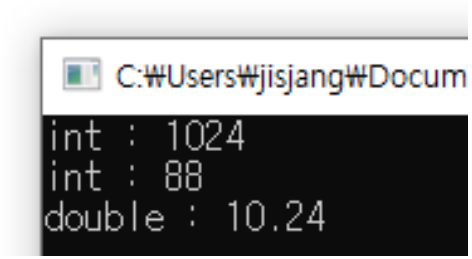
```
#include <iostream>

void print(int x) { std::cout << "int : " << x << std::endl; }
void print(char x) { std::cout << "char : " << x << std::endl; }
void print(double x) { std::cout << "double : " << x << std::endl; }

int main() {
    int a = 1024;
    char b = 'X';
    double c = 10.24f;

    print(a);
    print(b);
    print(c);

    getchar();
    return 0;

}
```

```
C:₩Users₩jisjang₩Documents₩Visual Studio 2015₩Pr
int : 1024
char : X
double : 10.24
```

Function Overloading cont'd

- Print(b) calls print (int x)

```
#include <iostream>

void print(int x) { std::cout << "int : " << x << std::endl; }
//void print(char x) { std::cout << "char : " << x << std::endl; }
void print(double x) { std::cout << "double : " << x << std::endl; }

int main() {
    int a = 1024;
    char b = 'X';
    double c = 10.24f;

    print(a);
    print(b);
    print(c);

    getchar();
    return 0;


}
```



Function Overloading cont'd

- Compilation error due to ambiguous call to print function with c

```
#include <iostream>

void print(int x) { std::cout << "int : " << x << std::endl; }
void print(char x) { std::cout << "char : " << x << std::endl; }
//void print(double x) { std::cout << "double : " << x << std::endl; }

int main() {
    int a = 1024;
    char b = 'X';
    double c = 10.24f;

    print(a);
    print(b);
    print(c);

    getchar();
    return 0;


}
```

```
1>  Hello.cpp
1>c:₩users₩jisjang₩documents₩visual studio 2015₩projects₩helloworld₩helloworld₩hello.cpp(14): error C2668: 'print': 오버로드된 함수에 대한 호출이 모호합니다.
1>  c:₩users₩jisjang₩documents₩visual studio 2015₩projects₩helloworld₩helloworld₩hello.cpp(4): note: 'void print(char)'일 수 있습니다.
1>  c:₩users₩jisjang₩documents₩visual studio 2015₩projects₩helloworld₩helloworld₩hello.cpp(3): note: 또는       'void print(int)'
1>  c:₩users₩jisjang₩documents₩visual studio 2015₩projects₩helloworld₩helloworld₩hello.cpp(14): note: 인수 목록 '(double)'을(를) 일치시키는 동안
========== 빌드: 성공 0, 실패 1, 최신 0, 생략 0 ==========
```

Argument Matching Rule

1. C++ tries to find an exact match

2. If no exact match is found, C++ tries to find a match through promotion
   - Char, unsigned char, and short is promoted to an int.
   - Unsigned short can be promoted to int or unsigned int, depending on the size of an int
   - Float is promoted to double
   - Enum is promoted to int

Reference: https://www.learncpp.com/cpp-tutorial/76-function-overloading/

(… continued)

Argument Matching Rule Cont'd

3. If no promotion is possible, C++ tries to find a match through standard conversion
   - Any numeric type will match any other numeric type, including unsigned (e.g. int to float)
   - Enum will match the formal type of a numeric type (e.g. enum to float)
   - Zero will match a pointer type and numeric type (e.g. 0 to char*, or 0 to float)
   - A pointer will match a void pointer
4. C++ tries to find a match through user-defined conversion

5. Ambiguous matches

   more than one matches → compile-time error

Reference: https://www.learncpp.com/cpp-tutorial/76-function-overloading/

Constructors and Destructors

Constructors

- Constructor is member function that is called automatically when the object created in order to initialize its value
- Has same name as class
- Has no return type
- Can be overloaded
- Cannot be called directly

Default Constructor

- Called with no arguments
  - Automatically provided as public member if no user-declared constructors

```
class Animal {
public:
    Animal() {
        numberOfLeg = 4;
    }
    Animal(int numberOfLeg) {
        this->numberOfLeg = numberOfLeg;
    }
private:
    int numberOfLeg;
};

Animal objAnimal;
Animal funcAnimal();
```

Indicates current class

What's the difference?

Parameterized Constructor

❖ Constructor can be overloaded with parameters

```
class Animal {
public:
    Animal() {
        numberOfLeg = 4;
    }
    Animal(int numberOfLeg) {
        this->numberOfLeg = numberOfLeg;
    }
private:
    int numberOfLeg;
};

Animal objAnimal;
Animal funcAnimal();
```

Constructor overloading

Explicit and Implicit Call

```
1   #include <iostream>
2
3   class Point {
4   private:
5       int x, y;
6   public:
7       // Parameterized Constructor
8       Point(int x1, int y1)
9       {
10          x = x1;
11          y = y1;
12      }
13      int getX()
14      {
15          return x;
16      }
17      int getY()
18      {
19          return y;
20      }
21  };
22  int main()
23  {
24      // Constructor called
25      Point p1(10, 15);
26      Point p2 = Point(10,15);
27      // Access values assigned by constructor
28      std::cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
29
30      return 0;
31  }
```

Implicit call of constructor

Explicit call of constructor

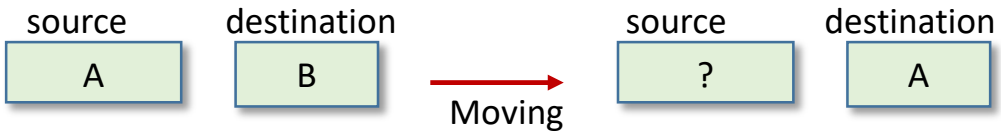
Copying and Moving

❖ Copying propagates the value of the source object to the destination object

- Without modifying the source object

❖ Moving propagates the value of the source object to the destination object

- Permitted to modify the source object



❖ In general, moving is more efficient than copying

Copy Constructor

- Used to create object by copying from already-existing object
- Typical form: T (const T&) /*T is class*/
- Example

```
class Animal {
public:
    Animal() { //Default constructor
        numberOfLeg = 4;
    }
    Animal(int numberOfLeg) {
        this->numberOfLeg = numberOfLeg;
    }
    Animal(const Animal& a) //Copy constructor
    {
        numberOfLeg = a.numberOfLeg;
    }
private:
    int numberOfLeg;
};

Animal a;
Animal b(a); //invokes Animal(const Animal&)
Animal c = a; //invokes Animal(const Animal&)
```

Look at how it is invoked!

Copy Constructor cont'd

❖ Copy constructor with shallow copy

```
class Animal{
public:
    char * name;
    int age;

    Animal(int age_, char* name_) {
        age = age_;
        name = new char[strlen(name_) + 1];
        strcpy(name, name_);
    }
    Animal(Animal & a) {//Copy constructor with shallow copy
        age = a.age;
        name = a.name;
    }

    void changeName(char *newName) {
        strcpy(name, newName);
    }
    void printAnimal() {
        std::cout << "Name: " << name << " Age: "
            << age << std::endl;
    }
};
```

Shallow copy

Invoke copy constructor

```
void main() {
    Animal A(10, "Jenny");
    Animal B = A;
    A.age = 22;
    A.changeName("Brown");

    A.printAnimal();
    B.printAnimal();
    getchar();
}
```

Output

```
Name: Brown Age: 22
Name: Brown Age: 10
```

Why the names of both A and B are changed to Brown?? ☹

Copy Constructor cont'd

❖ Copy constructor with deep copy

```
class Animal {
public:
    char * name;
    int age;

    Animal(int age_, char* name_) {
        age = age_;
        name = new char[strlen(name_) + 1];
        strcpy(name, name_);
    }
    Animal(Animal & a) {//Copy constructor with deep copy
        age = a.age;
        name = new char[strlen(a.name) + 1];
        strcpy(name, a.name);
    }

    void changeName(char *newName) {
        strcpy(name, newName);
    }
    void printAnimal() {
        std::cout << "Name: " << name << " Age: "
            << age << std::endl;
    }
};
```

Deep copy

```
void main() {
    Animal A(10, "Jenny");
    Animal B = A;
    A.age = 22;
    A.changeName("Brown");

    A.printAnimal();
    B.printAnimal();
    getchar();
}
```

Output

```
Name: Brown Age: 22
Name: Jenny Age: 10
```

Name of B is preserved. Why?

Copy Constructor cont'd

❖ Shallow copy vs. deep copy (cont'd)

Consideration:

Default copy constructor created by compiler do not support deep copy.

If we need to use deep copy, the copy constructor must be manually created by a programmer!

Move Constructor

- Constructor that takes rvalue
- Used to create object by moving from already-existing object
- Form: T (T&&)

```
class Animal {
public:
    Animal() { //Default constructor
        numberOfLeg = 4;
    }
    Animal(int numberOfLeg) {
        this->numberOfLeg = numberOfLeg;
    }
    Animal(const Animal&& a) //Move constructor
    {
        numberOfLeg = a.numberOfLeg;
    }
private:
    int numberOfLeg;
};

Animal a;
Animal b(std::move(a)); //invokes Animal::Animal(const Animal&&)
Animal c = std::move(a); //invokes Animal::Animal(const Animal&&)
```

Constructor Initializer Lists

❖ Data members always initialized in order of declaration, regardless of order in initializer list

```
#include <string.h>
#include <iostream>

class My_cat {
    int age;
    char *name;

public:
    My_cat();
    My_cat(int x, const char *name);
    My_cat(const My_cat &cat);
    ~My_cat();

    void show_status();
};
My_cat::My_cat():age(20), name(NULL) { }
/*
My_cat::My_cat()  {
    age = 20;
    name = NULL;
}*/
My_cat::My_cat(const My_cat &cat) {
    std::cout << "Copy constructor invocation ! " << std::endl;
    age = cat.age;
    name = new char[strlen(cat.name) + 1];
    strcpy(name, cat.name);
}
```

Order of declaration:
age → name

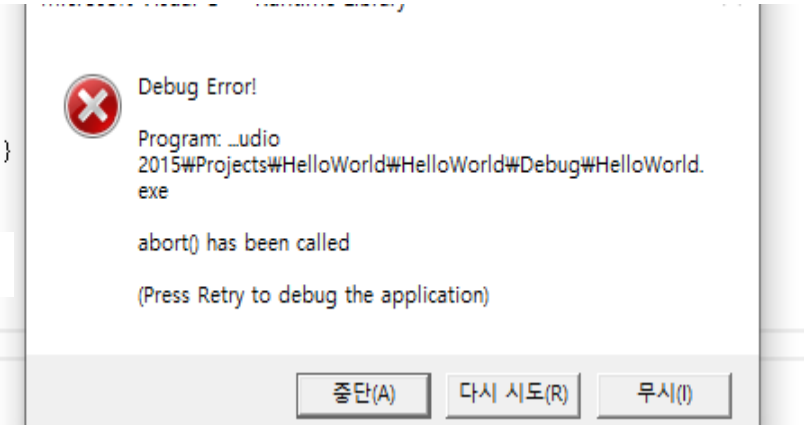
Initializer list

Constructor Initializer Lists cont'd

- Data members always initialized in order of declaration, regardless of order in initializer list

```
#include <cassert>

class Machine {
public:
    Machine( ) : width(55), height(width + 30) { assert(height == 85); }
    int height;
    int width;
};

int main( ) {
    Machine m;
}
```

Initialization order: height → width



Error (undefined behavior)!
height is initialized before ***width***!

Why Use Initializer List?

❖ Initializer List creates and initializes member variables at the same time

❖ ***const*** values and references must be created and initialized at the same time

- Therefore, ***const*** values and ***reference*** should be initialized using initializer list

```
const int i;
i = 3;          //Error

int y = 4;
int& ref;
ref = y;        //Error
```

Destructor

❖ Destructor is a member function which destructs or deletes an object.

- Called automatically when the object goes out of scope
  - Function ends
  - Program ends
  - A block that containing local variables ends
  - A delete operator is called

- Syntax

  ~ classname() // no return and no argument

Destructor Example

```
class String
{
private:
    char *s;
    int size;
public:
    String(char *); // constructor
    ~String();      // destructor
};

String::String(char *c)
{
    size = strlen(c);
    s = new char[size+1];
    strcpy(s,c);
}

String::~String()
{
    delete []s;
}
```

IMPORTANT: default destructor does not invoke "delete". Hence, if we have "new" in object, the destructor needs to be written by a programmer!

No return and no argument

Why the "delete" is invoked?

Because of the "new" operation, "delete" was invoked.

Figure from: https://www.geeksforgeeks.org/destructors-c/

Copy and Move Constructors

Example use case

Copy Constructor Only

```cpp
#include <iostream>
#include <string.h>
#include <vector>

class Animal {

  char * name;
  int age;
  public:
    Animal(int age_, const char* name_) {
    age = age_;
    name = new char[strlen(name_) + 1];
    strcpy(name, name_);
   }
  Animal(const Animal & a) {//Copy constructor with deep copy
   age = a.age;
   name = new char[strlen(a.name) + 1];
   strcpy(name, a.name);
   std::cout << "Copy constructor is invoked!!\n";
 }
 ~Animal(){
  std::cout <<"Destructor!!" << std::endl;
  delete [] name;
  }
  void changeName(const char *newName) {
   strcpy(name, newName);
  }
  void printAnimal() {
   std::cout << "Name: " << name << " Age: "
   << age << std::endl;
  }
};
```

```cpp
int main() {
 Animal A(10, "Jenny");
 A.printAnimal();
 std::vector<Animal> vec; //Vector for Animal type
 std::cout << "-----1st push-----\n";
 vec.push_back(A);   //Insert an Animal object to vec
 std::cout << "-----2nd push-----\n";
 vec.push_back(A);
 std::cout << "-----3rd push-----\n";
 vec.push_back(A);
 std::cout << "-----4th push-----\n";
 vec.push_back(A);
 std::cout << "-----5th push-----\n";
 vec.push_back(A);

 A.printAnimal();
 vec[0].printAnimal();
 vec[1].printAnimal();
 vec[2].printAnimal();
 vec[3].printAnimal();
 vec[4].printAnimal();

 return 0;
}
```

Copy Constructor Only

```
> clang++-7 -pthread -std=c++17 -o main main.cpp
> ./main
Name: Jenny Age: 10
-----1st push-----
Copy constructor is invoked!!
-----2nd push-----
Copy constructor is invoked!!
Copy constructor is invoked!!
Destructor!!
-----3rd push-----
Copy constructor is invoked!!
Copy constructor is invoked!!
Copy constructor is invoked!!
Destructor!!
Destructor!!
-----4th push-----
Copy constructor is invoked!!
-----5th push-----
Copy constructor is invoked!!
Copy constructor is invoked!!
Copy constructor is invoked!!
Copy constructor is invoked!!
Copy constructor is invoked!!
Destructor!!
Destructor!!
Destructor!!
Destructor!!
Name: Jenny Age: 10
Name: Jenny Age: 10
Name: Jenny Age: 10
Name: Jenny Age: 10
Name: Jenny Age: 10
Name: Jenny Age: 10
Destructor!!
Destructor!!
Destructor!!
Destructor!!
Destructor!!
Destructor!!
```

With Move Constructor

```cpp
#include <iostream>
#include <string.h>
#include <vector>

class Animal {

  char * name;
  int age;
public:
  Animal(int age_, const char* name_) {
    age = age_;
    name = new char[strlen(name_) + 1];
    strcpy(name, name_);
  }
  Animal(const Animal & a) {//Copy constructor with deep copy
    age = a.age;
    name = new char[strlen(a.name) + 1];
    strcpy(name, a.name);
    std::cout << "Copy constructor is invoked!!\n";
  }
  Animal(Animal && a) noexcept {//Move constructor with shallow copy
    age = a.age;
    name = a.name;
    std::cout << "Move constructor is invoked!!\n";
    a.name = nullptr;
  }
  ~Animal() {
    std::cout << "Destructor!!" << std::endl;
    if (name) delete[] name;
  }
  void changeName(const char *newName) {
    strcpy(name, newName);
  }
  void printAnimal() {
    std::cout << "Name: " << name << " Age: "
      << age << std::endl;
  }
};
```

```cpp
int main() {
  Animal A(10, "Jenny");

  A.printAnimal();
  std::vector<Animal> vec; //Vector for Animal type
  std::cout << "-----1st push-----\n";
  vec.push_back(A);   //Insert an Animal object to vec
  std::cout << "-----2nd push-----\n";
  vec.push_back(A);
  std::cout << "-----3rd push-----\n";
  vec.push_back(A);
  std::cout << "-----4th push-----\n";
  vec.push_back(A);
  std::cout << "-----5th push-----\n";
  vec.push_back(A);
  A.printAnimal();
  vec[0].printAnimal();
  vec[1].printAnimal();
  vec[2].printAnimal();
  vec[3].printAnimal();
  vec[4].printAnimal();

  return 0;
}
```

With Move Constructor

```
clang++-7 -pthread -std=c++17 -o main main.cpp
./main
Name: Jenny Age: 10
-----1st push-----
Copy constructor is invoked!!
-----2nd push-----
Copy constructor is invoked!!
Move constructor is invoked!!
Destructor!!
-----3rd push-----
Copy constructor is invoked!!
Move constructor is invoked!!
Move constructor is invoked!!
Destructor!!
Destructor!!
-----4th push-----
Copy constructor is invoked!!
-----5th push-----
Copy constructor is invoked!!
Move constructor is invoked!!
Move constructor is invoked!!
Move constructor is invoked!!
Move constructor is invoked!!
Destructor!!
Destructor!!
Destructor!!
Destructor!!
Name: Jenny Age: 10
Name: Jenny Age: 10
Name: Jenny Age: 10
Name: Jenny Age: 10
Name: Jenny Age: 10
Name: Jenny Age: 10
Destructor!!
Destructor!!
Destructor!!
Destructor!!
Destructor!!
Destructor!!
```

NOTE: move constructors are invoked !!

Let’s Run the Code!

https://docs.google.com/document/d/1R_GIK8oYn30M82AKi-3DgJorRmzPt5H2S-ZMR23gmOM/edit?usp=sharing

References

- *geeksforgeeks.org*
- *modoocode.com*
- *www.learncpp.com*
- *sourcemaking.com*
- www.infobrother.com
- *Lecture Slides for Programming in c++, google books*