Topic 7
Classes – Constructors and Destructor

Constructor

Constructor overloading

Default constructor

Member initialization list

Copy constructor

Move constructor

Destructor

Object Constructions

- Uninitialized instance can cause an unpredictable behavior
- Construction consists of two steps
  - allocation of appropriate memory space for the object
  - initialization function is called automatically to initialize the memory space
- Ensure when an object is created, it is always placed in a predictable state.

Good Design :

constructors, assignments, and destructors

- These functions control the lifecycle of objects: creation, copy, move, and destruction
- Define constructors to guarantee and simplify initialization of classes
- These are default operations:
  - default constructor: X()
  - copy constructor: X(const X&)
  - copy assignment: operator=(const X&)
  - move constructor: X(X&&)
  - move assignment: operator=(X&&)
  - destructor: ~X()
- The default operations are a set of related operations that together implement the lifecycle semantics of an object.

Good Design :
avoid defining default operations (C.20)

- 기본 연산을 정의하지 않아도 되면 그렇게하라!
- 가장 단순하고, 명료한 의미를 준다.
- This is known as “the rule of zero”.

```
struct Named_map {
public:
    // ... no default operations declared ...
private:
    string name;
    map<int, int> rep;
};

Named_map nm;            // default construct
Named_map nm2 {nm};  // copy construct
```

https://github.com/isocpp/CppCoreGuidelines

Good Design :
the rule of six (C.21)

- 복사(copy), 이동(move), 소멸자(destructor)의 의미는 서로 밀접하게 연관되어 있어, 만약 이들 중 하나가 선언되면, 다른 함수들도 고려할 필요가 있다.
- 복사/이동/소멸자 함수를 = default 또는 = delete로 선언하면, (컴파일러는) 이동 생성자 및 이동 할당 연산자의 묵시적 선언이 안됩니다.
- 이동 생성자 또는 이동 할당 연산자를 = default 또는 = delete로 선언하면 묵시적으로 생성된 복사 생성자 또는 복사 할당 연산자가 삭제된 것으로 정의됩니다.
- 이들 중 하나가 선언되자마자 나머지는 모두 선언해야 합니다. 모든 잠재적인 이동을 더 비싼 복사본으로 바꾸거나 클래스를 이동 전용으로 만드는 것과 같은 원치않는 효과를 피하기 위해서 입니다.

```
struct M2 {
  public:
    // ... no copy or move operations ...
    ~M2() { delete[] rep; }  //destructor
  private:
    std::pair<int, int>* rep;  // zero-terminated set of pairs
};
```

```
void use() {
        M2 x;
        M2 y;
        // the default assignment
        x = y;
} //정상 종료 안됨
```

Reference vs Value(Copy) Semantics

- C++ gives you the choice: use the assignment operator to copy the value (copy/value semantics), or use a pointer-copy to copy a pointer (reference semantics)

	Reference Semantics	Value Semantics
assignment	pointer-copy (i.e., a reference)	copies the value, not just the pointer **(default and most common)**
Pros	flexibility and dynamic binding (pass by pointer or pass by reference)	speed?*
Cons	memory management issues reference aliasing problems referential transparency	performance?

https://isocpp.org/wiki/faq/value-vs-ref-semantics
*http://cpp-next.com/archive/2009/08/want-speed-pass-by-value/

Good Design : 정규 타입(regular type) (C.11)

- One ideal for a class is to be a **regular type**. That means roughly "behaves like an ***int***."
- A value of regular type can be copied and the result of a copy is an independent object with the same value as the original.
- **If a concrete type has both = and ==, a = b should result in a == b being true** (assignment and equality)
- The C++ built-in types are regular, and so are standard-library classes, such as string, vector, and map
- Regular types are easier to understand and reason about than types that are not regular

```
struct Bundle {
    string name;
    vector<Record> vr;
};
bool operator==(const Bundle& a, const Bundle& b){
    return a.name == b.name && a.vr == b.vr;
}
```

```
Bundle b1 { "my bundle", {r1, r2, r3}};
Bundle b2 = b1;
if (!(b1 == b2)) error("impossible!");
b2.name = "the other bundle";
if (b1 == b2) error("No!");
```

Object Construction in C++

- Constructor is a special non-static member function of a class that is used to initialize objects of its class type
- A constructor is a specialized member function
  - Only used for initializing object
  - Automatically invoked whenever an object is created
  - has the same name as the class itself
  - has no return type

```
class Rectangle {
  int leftTopX, leftTopY ;
  int rightBottomX, rightBottomY ;
public:
  Rectangle(int x1, int y1, int x2, int y2) {
     leftTopX = x1 ; leftTopY = y1 ;
     rightBottomX = x2 ; rightBottomY = y2 ;
  }
  …
```

Constructor

```
class Rectangle {
  int leftTopX, leftTopY ;
  int rightBottomX, rightBottomY ;
public:
  Rectangle(int x1, int y1, int x2, int y2) {
    leftTopX = x1 ; leftTopY = y1 ;
    rightBottomX = x2 ; rightBottomY = y2 ;
    // 대신에set()을호출하는것도가능함
  }
  void set(int x1, int y1, int x2, int y2) {
    leftTopX = x1 ; leftTopY = y1 ;
    rightBottomX = x2 ; rightBottomY = y2 ;
  }
  void getLeftTop(int& x, int& y) const {
    x = leftTopX ; y = leftTopY ;
  }
  void getRightBottom(int& x, int& y) const {
    x = rightBottomX ; y = rightBottomY ;
  }
  int getArea() const {
    return (rightBottomX - leftTopX)
    * (rightBottomY - leftTopX) ;
  }
} ;
```

```
int main() {
    int x1, y1, x2, y2 ;
    cin >> x1 >> y1 >> x2 >> y2 ;

    Rectangle r1(x1, y1, x2, y2) ;
    // r1.set(...)을 하지 않음

    int x3, y3, x4, y4 ;
    r1.getLeftTop(x3, y3) ;
    r1.getRightBottom(x4, y4) ;

    Rectangle r2(x3, y3, x4, y4) ;
    // r2.set(...)을 하지 않음

    cout << endl << r1.getArea()
        << '₩t' << r2.getArea() << endl ;
}
```

Constructor Invocation

```
int main() {
    int x1, y1, x2, y2 ;
    cin >> x1 >> y1 >> x2 >> y2 ;

    Rectangle r1(x1, y1, x2, y2) ;

    int x3, y3, x4, y4 ;
    r1.getLeftTop(x3, y3) ;
    r1.getRightBottom(x4, y4) ;

    Rectangle* const pR = new Rectangle(x3, y3, x4, y4) ;

    cout << endl << r1.getArea() << '₩t' << pR->getArea() << endl ;

    delete pR ;
}
```

Constructor Overloading

```
class Point {
public:
    int x, y ;
    Point(int x, int y) { this->x = x ; this->y = y ; }
} ;
class Rectangle {
    int leftTopX, leftTopY, rightBottomX, rightBottomY ;
public:
    // 1) 번생성자
    Rectangle(int x1, int y1, int x2, int y2) { set(x1, y1, x2, y2) ; }
    // 2) 번생성자
    Rectangle(int x, int y) { set(x, y, 0, 0) ; }
    // 3) 번생성자
    Rectangle(const Point& leftTop, const Point& rightBottom) {
        set(leftTop.x, leftTop.y, rightBottom.x, rightBottom.y) ;
    }
    // 4번생성자
    Rectangle(const Point& leftTop) { set(leftTop.x, leftTop.y, 0, 0) ; }
    void set(int x1, int y1, int x2, int y2) {
        leftTopX = x1 ; leftTopY = y1 ;
        rightBottomX = x2 ; rightBottomY = y2 ;
    }
```

Constructor Overloading

```
int main() {
   int x1, y1, x2, y2 ;
   cin >> x1 >> y1 >> x2 >> y2 ;

   // 1) 번 생성자 호출
   Rectangle r1(x1, y1, x2, y2) ;

   // 2) 번 생성자 호출
   Rectangle* const pR2 = new Rectangle(x1+10, y1+10) ;

   Point p1(10, 10), p2(20, 20) ;
   // 3) 번 생성자 호출
   Rectangle r3(p1, p2) ;

   // 4) 번 생성자 호출
   Rectangle* const pR4 = new Rectangle(Point(30, 30)) ;

   delete pR2 ;
   delete pR4 ;
}
```

Constructor Overloading

- Overloading can be simplified by default arguements

```
class Point {
public:
    int x, y ;
    Point(int x, int y) { this->x = x ; this->y = y ; }
} ;
class Rectangle {
    int leftTopX, leftTopY ;
    int rightBottomX, rightBottomY ;
public:
    // 1) 번 생성자
    Rectangle(int x1, int y1, int x2=0, int y2=0) { set(x1, y1, x2, y2) ; }
    // 2) 번 생성자: 불필요
    // Rectangle(int x, int y) { set(x, y, 0, 0) ; }
    // 3) 번생성자
    Rectangle(const Point& leftTop, const Point& rightBottom=Point(0,0)) {
        set(leftTop.x, leftTop.y, rightBottom.x, rightBottom.y) ;
    }
    // 4번 생성자: 불필요
    // Rectangle(const Point& leftTop) { set(leftTop.x, leftTop.y, 0, 0) ; }
    void set(int x1, int y1, int x2, int y2) {
        leftTopX = x1 ; leftTopY = y1 ;
        rightBottomX = x2 ; rightBottomY = y2 ;
    }
```

객체 초기화
with braced initialization

```
class Array {
    int myData[5];
  public:
    //braced initialization
    Array(): myData{1,2,3,4,5} {}
};
class MyClass {
  public:
    int x;
    double y;
};
class MyClass2 {
    int x;
    double y;
  public:
    MyClass2(int first, double second) :
      x{first}, y{second} {};
};
```

```
int main() {
    Array arr;
    // Initializations using aggregation
    // 1. 모든 멤버 변수가 public
    // 2. 사용자 정의 생성자가 없는 경우
    MyClass myClass{2011, 3.14};
    MyClass myClass1= {2011, 3.14};

    // Initializations using the constructor
    MyClass2 myClass2{2011, 3.14};
    MyClass2 myClass3= {2011, 3.14};
}
```

객체 배열의 초기화

```
class Point {
public:
    int x, y ;
    Point(int _x=0, int _y=0) x(_x), y(_y) {}
} ;
class Rectangle {
    …
public:
    // 기본(default) 생성자
    Rectangle() { … }
    // 1) 번 생성자
    Rectangle(int x1, int y1, int x2, int y2) {…}
    // 2) 번 생성자
    Rectangle(int x, int y) { set(x, y, 0, 0) ; }
    // 3) 번 생성자
    Rectangle(const Point& leftTop,
       const Point& rightBottom) { … }
    // 4) 번 생성자
    Rectangle(const Point& leftTop) { … }
    …
}
```

```
int main() {
   // 기본생성자 호출
   Rectangle retangles1[5] ;

   Rectangle rectangles2[5] = {
       Rectangle(),       // 기본생성자
       Rectangle(10, 10, 20, 20), // 1)
       Rectangle(10, 10), // 2)
       Rectangle(Point(10, 10), Point(20,20)), // 3)
       Rectangle(Point(10,10)) // 4)
   } ;  //(수정) 배열의 원소는 5개로 가정함

   int rectNo ;
   cin >> rectNo ;
   Rectangle* const pRectangles1  =
       new Rectangle[rectNo];
   delete [] pRectangles1 ;

   Rectangle* const pRectangles2  =
       new Rectangle[rectNo] {
          Rectangle(10, 10, 20, 20),
          Rectangle(10, 10)
       };
     delete [] pRectangles2;
}
```

(추가) initializer_list 를 이용한 초기화

- Lightweight proxy object that provides access to an array of objects of type const T.
- brace initialization 방식 혹은 배열의 초기화 방식으로 초기화 가능

```
#include <initializer_list>
class Phonebook {
    std::map<std::string, int> contacts;
public:
    Phonebook(std::initializer_list <std::pair<string, int>> lst) {
        for (const auto &l : lst)
            contacts.insert(l);
    }
    void print() const {
        for(const auto& c : contacts)
            std::cout << c.first << " " << c.second << '₩n';
    }
};
```

```
int main() {
    Phonebook p = {{"Kim", 24}, {"Lee", 21}};
    p.print();
    return 0;
}
```

Good Design :
constructor

- 생성자는 완전히 초기화된 객체를 생성해야 함
- 생성자가 유효한 객체를 생성하지 못했다면 예외를 던짐

```
class X2 {
  FILE* f;
  // ...
public:
  X2(const string& name)
    :f{fopen(name.c_str(), "r")} {
    if (!f)
      throw runtime_error{"could not open" + name};
      // ...
    }
    void read();      // read from f
};
```

```
void f() {
    X2 file {"Zeno"}; // throws if file isn't open
    file.read();          // fine
    // ...
}
```

https://github.com/isocpp/CppCoreGuidelines

Good Design :
declare single-argument constructors explicit (C.46)

- 의도하지 않은 형변환(convertions)을 피한다.
- 복사(copy) 및 이동(move) 생성자는 형변환을 수행하지 않으므로 명시적(explicit)으로 만들어서는 안됨
  - explicit 복사/이동 생성자는 값에 의한 전달(call by value) 및 반환(return value)을 어렵게 함

```
class Vector {
    int size;
    int* data;
public:
    Vector(int x);   // BAD, 크기가 n개인 벡터 생성
    Vector(int x) explicit;
    // ...
};
Vector addVector (Vector a, Vector b){
     …
}
```

```
int main() {
    //묵시적 형변환이 일어남
    //크기가 3인 벡터가 두 개 생성됨
    Vector vec = addVector(3, 3);
}
```