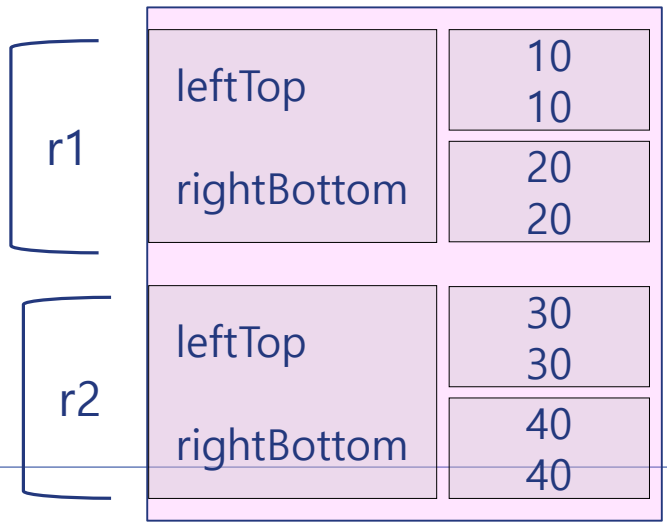
객체 데이터 멤버 – 멤버 초기화 목록

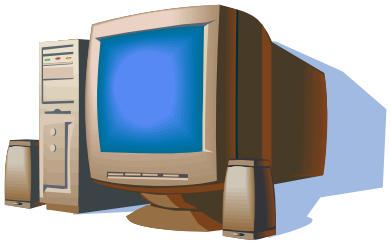
```
class Point {
private:
  int x ;
  int y ;
public:
  Point( int _x=0, int _y=0) {
    x = _x  ;  y = _y ;
  }
} ;
```

```
class Rectangle {
private:
  Point leftTop ;
  Point rightBottom ;
public:
  Rectangle(int x1, int y1, int x2, int y2)
    : leftTop{x1, y1}, rightBottom{x2, y2}
    { }
  ...
} ;
```

```
int main() {
    Rectangle r1(10, 10, 20, 20) ;

    Rectangle r2(30, 30, 40, 40) ;

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



Computer and Monitor



samsungPC
(samsungMonitor)

hpPC
(hpMonitor)

Computer and Monitor

예제 프로그램	실행 결과

int main() {
  Monitor samsungMonitor("SamsungMonitor", 100) ;
  Computer samsungPC("Samsung",
  samsungMonitor,
  samsungMonitor.getPrice() + 200) ;
 	
  cout << samsungPC.getPrice() << endl ;
  samsungPC.run("Hello C++") ;
}
	300
	Runs on Samsung
	Samsung Monitor: Hello C++

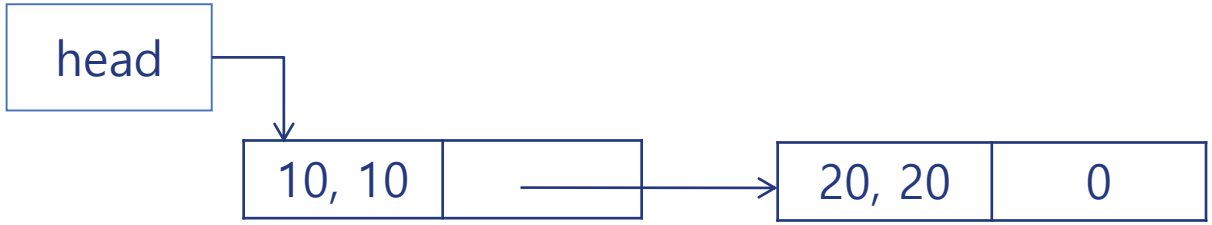
Monitor

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

class Monitor {
 string maker ;
 int price ;
public:
 Monitor(const string& _maker,  const int _price)
   : maker(_maker) {
   price = _price ;
 }
 void display(const string& msg) {
   cout << maker << ": " << msg << endl ;
 }
 int getPrice() { return price ; }
} ;
```

Computer

```
class Computer {
  string maker ;              // string 객체를 데이터 멤버로 가짐
  Monitor monitor ;           // Monitor 객체를 데이터 멤버로 가짐
  int price ;
public:
  Computer(const string& _maker, const Monitor& _monitor, const int _price)
     : maker(_maker), monitor(_monitor) {
     price = _price ;
   }
  void run(const string& msg) {
     cout << "Runs on " << maker << endl ;
     monitor.display(msg) ;
  }
  int getPrice() { return price ; }
} ;
```

단방향 연결 리스트




```
class Point {
  int x, y ;
  Point* pNext ;
public:
  Point(int _x=0, int _y=0, Point* const _pNext=nullptr) {
    x = _x ; y = _y ;
    pNext = _pNext ;
  }
  Point* getNext() { return pNext ; }
  void setNext(Point* const _pNext) { pNext = _pNext ; }
  void print() {
    cout << x << ", " << y << endl ;
  }
}
```

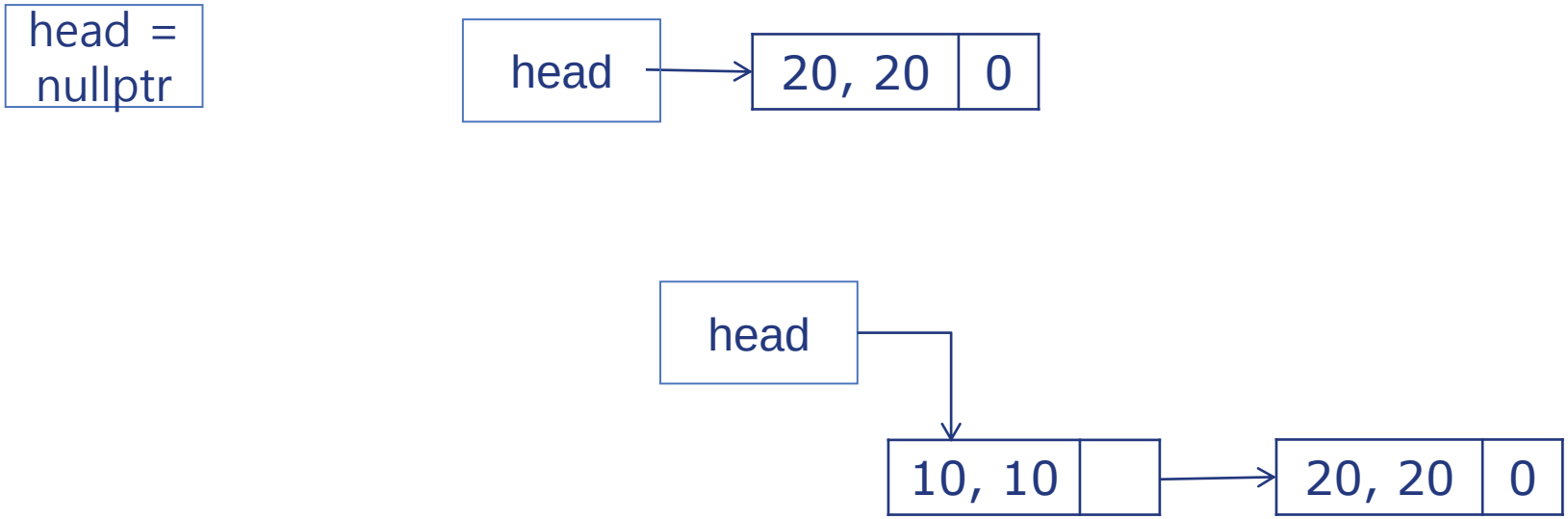
nullptr
Since C++11

단방향 연결 리스트

```
class SinglyLinkedList {
  Point* head ;
public:
  SinglyLinkedList() { head = nullptr ; }
  void print() {
    Point* pPoint = head ;
    while ( pPoint != nullptr ) {
      pPoint->print() ;
      pPoint = pPoint->getNext() ;
    }
  }
```

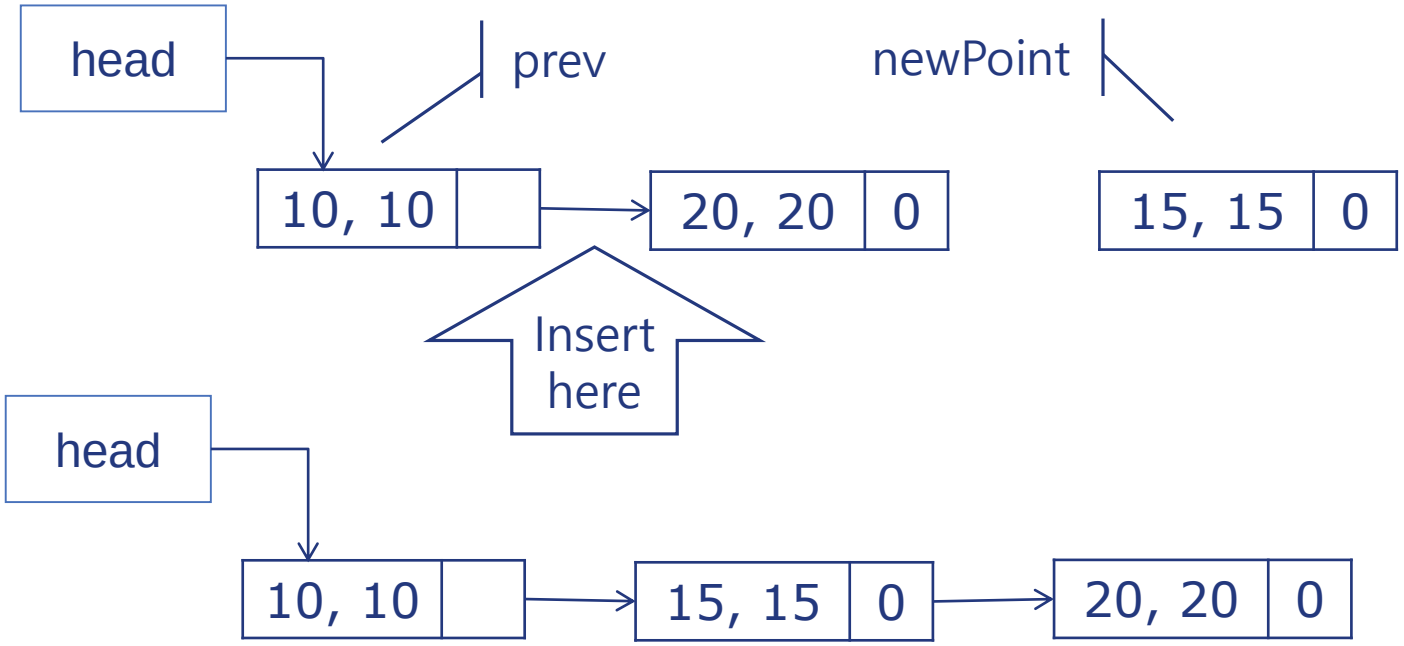
단방향 연결 리스트

```
void prepend(Point* const newPoint) {
  if ( head == nullptr ) head = newPoint ;
  else {
    newPoint->setNext(head) ;
    head = newPoint ;
  }
}
```



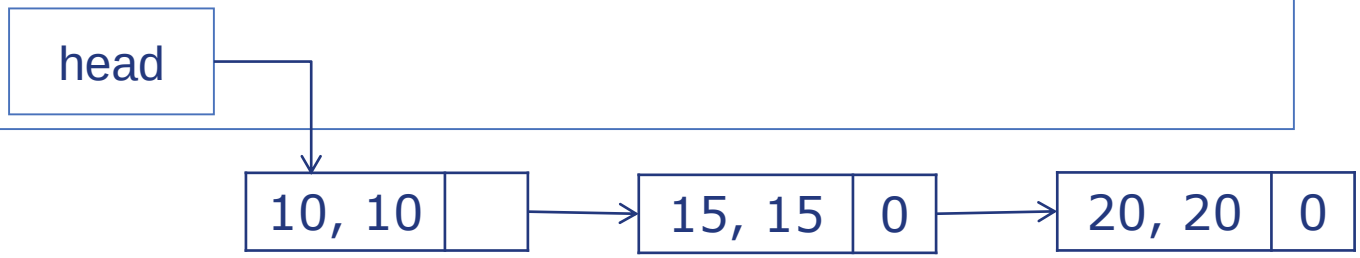
단방향 연결 리스트

```
void insertAfter(Point* const prev, Point* const newPoint) {
   assert ( prev != nullptr );
   newPoint->setNext(prev->getNext()) ;
   prev->setNext(newPoint) ;
}
```



단방향 연결 리스트

```
  void remove(Point* const toBeRemoved) {
    if ( toBeRemoved == head ) {
      head = head->getNext() ; return ;
    }
    Point* pPoint = head ;
    while ( pPoint != nullptr ) {
      if ( pPoint->getNext() == toBeRemoved ) {
        Point* pNextOfToBeRemoved = toBeRemoved->getNext() ;
        pPoint->setNext(pNextOfToBeRemoved) ;
        break ;
      }
      pPoint = pPoint->getNext() ;
    }
  }
} ;
```



this

- pointer to the object itself

```
# ifndef __RECTANGLE_H
# define __RECTANGLE_H
class Rectangle {
  int leftTopX, leftTopY ;
  int rightBottomX, rightBottomY ;
public:
  …
  // this의 활용 예1)
  void setLeftTopX(const int leftTopX) { this->leftTopX = leftTopX ; }
  void setLeftTopY(const int leftTopY) { this->leftTopY = leftTopY ; }
  void setRightBottomX(const int rightBottomX) { this->rightBottomX = rightBottomX ; }
  void setRightBottomY(const int rightBottomY) { this->rightBottomY = rightBottomY ; }
} ;
# endif
```

this의 활용 예

```
# ifndef __RECTANGLE_H
# define __RECTANGLE_H
class Rectangle {
public:
  Rectangle* copy() const {
     Rectangle* r = new Rectangle ;
     r->setLeftTopX(getLeftTopX()) ;
     r->setLeftTopY(getLeftTopY()) ;
     r->setRightBottomX(getRightBottomX()) ;
     r->setRightBottomY(getRightBottomY()) ;
     return r ;
  }
Rectangle* copy() const {
     return new Rectangle(*this) ;
  }
  bool isEqual(const Rectangle& r) const {
     return leftTopX == r.leftTopX && leftTopY == r.leftTopY
        && rightBottomX == r.rightBottomX
        && rightBottomY == r.rightBottomY ;
  }
} ;
# endif
```

```
# include <iostream>
# include <cassert>
# include "Rectangle.h"
using namespace std ;

int main() {
   Rectangle r ;
   r.set(0, 0, 100, 200) ;

   Rectangle* pR = r.copy() ;
   assert ( pR->isEqual(r) ) ;

   delete pR ;
}
```

this의 활용 예

```
# include <iostream>

# include "Rectangle.h"
using namespace std ;

int main() {
  Rectangle r ;
  r.set(0, 0, 100, 200) ;

  cout << r.moveBy(10, 10).print() << endl ; // 10 10 110 210
  // expected: 30 30 130 230, but actually 20 20 120 220
  cout << r.moveBy(10, 10).moveBy(10, 10).print() << endl ;

  r.moveBy(10, 10).print() ;
  cout << r.moveBy(10, 10).print().getArea() << endl ;
}
```

this의 활용 예

```
# ifndef __RECTANGLE_H
# define __RECTANGLE_H
class Rectangle {
public:
  ...
  Rectangle& moveBy(int deltaX, int deltaY) ;
  const Rectangle& print() const {
     cout << leftTopX << '₩t' << leftTopY << '₩t' << rightBottomX
        << '₩t' << rightBottomY << endl ;
     return *this ;
  }
   int getArea() const { ... }
} ;
# endif
```

```
# include "Rectangle.h"
...
Rectangle& Rectangle::moveBy(int deltaX, int deltaY) {
  setLeftTop(leftTopX+deltaX, leftTopY+deltaY) ;
  setRightBottom(rightBottomX+deltaX, rightBottomY+deltaY) ;
  return *this ;
}
```

Good Design: 빌더 (Builder)

- 한줄의 생성자 호출로는 만들 수 없는 복잡한 객체를 생성하는 방법

```
class PersonBuilder;

class Person {
private:
    string name, address, post_code;

public:
    Person(string name) : name(name) {};
    static PersonBuilder PersonBuilder
                        create(string name) {
        return PersonBuilder{name};
    }
    friend class PersonBuilder;

};
```

```
class PersonBuilder {
    typedef PersonBuilder self;
    Person person;
public:
    PersonBuilder(string name) :
                                person(name) {}
    self& at (string address){
        person.address = address;
        return *this;
    }
    self& postcode (string post_code) {
        person.post_code = post_code;
        return *this;
    }
};
```

```
Person p = Person::create("Kim")
                    .at("2, Busandaehak-ro 63beon-gil")
                    .postcode("46241");
```

Nested Classes

```
class Rectangle {
public:
  class Point {
  public:
    int x, y ;
    void print() const { cout << x << '₩t' << y ; }
    bool isEqual(const Point& p) const { return x == p.x && y == p.y ; }
  } ;

  Point leftTop, rightBottom ;
  void setLeftTop(int x, int y) { leftTop.x = x ; leftTop.y = y ; }
  void setRightBottom(int x, int y) { rightBottom.x = x ; rightBottom.y = y ; }
  bool isEqual(const Rectangle& r) const {
    return  leftTop.isEqual(r.leftTop) && rightBottom.isEqual(r.rightBottom) ;
  }
  const Rectangle& print() const {
    leftTop.print() ; cout << '₩t' ; rightBottom.print() ;
    return *this ;
  }
  ...
}
```

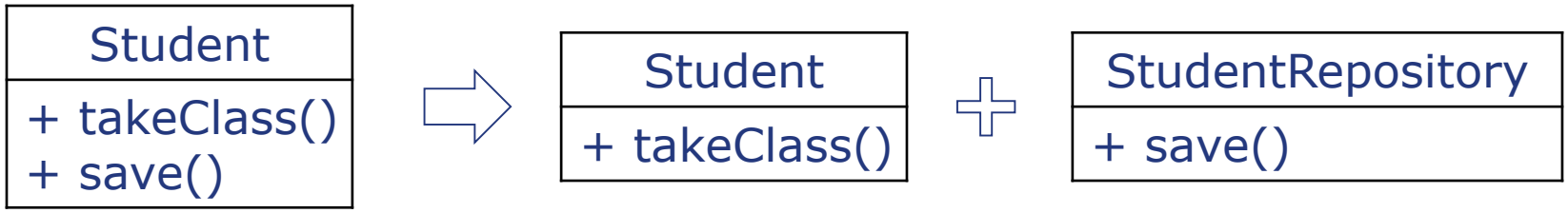
Nested Classes

```
# include <iostream>
# include <string>
using namespace std ;
# include "Rectangle.h"

int main() {
    Rectangle r1 ;
    r1.set(0, 0, 100, 200) ;

    Rectangle r2 ;
    r2.set(10, 10, 110, 210) ;

    r1.print() ; cout << endl ;
    r2.print() ; cout << endl ;

    string msg = r1.isEqual(r2) ? "same" : "different" ;
    cout << msg << endl ;

    Rectangle::Point pt ;
    // 가능은 하지만, 이럴 필요가 있으면 Point를 nested class로 하지 않는 것이 좋다
}
```

Good Design:
단일 책임 원칙 (Single Responsibility Principle)

클래스는 단 한 가지의 변경 이유만 가져야 한다!
A class should have only one reason to change.

- 책임 (responsibility)
  - 클래스가 변경되는 이유
  - 클래스를 변경하기 위한 한 가지 이상의 이유가 있다면 그 클래스는 여러가지 책임을 맡고 있다고 판단함
  - 책임을 묶어서 생각하는데 익숙해서 알아 차리기 어려움
  - 변경이 실제로 일어날 때가 진짜 변경이며, 책임을 잘못 분리하면 오히려 경직성(rigidity, 변경이 어려움) 와 취약성(fragility, 맨날 수정함) 가 발생함



- 실제로 객체 지향 디자인 관련 개념은 추상적이라서 많은 훈련이 필요함

Good Design: 책임 (Responsibility)

- 어떤 객체가 어떤 요청에 대해 대답해 줄 수 있거나, 적절한 행동을 할 의무가 있는 경우 해당 객체가 ***책임***을 가진다고 함
- 객체의 ***책임***은 객체가 알고 있는 것과 할 수 있는 것으로 구성됨
  - 하는 것: 객체를 생성하거나 계산하는 등의 스스로 하는 것, 다른 객체의 행동을 시작시키는 것, 다른 객체의 활동을 제어하고 조절하는 것
  - 아는 것: 자신의 정보, 관련 있는 객체에 관한 정보, 자신이 유도하거나 계산할 수 있는 정보
- 객체의 ***책임***을 이야기 할 때는 일반적으로 외부에서 접근 가능한 public 멤버 함수의 관점에서 이야기 함
  - 객체의 외부에 제공해 줄 수 있는 정보 (아는 것) 과 외부에 제공해 줄 수 있는 서비스 (하는 것)의 목록
- 어떤 객체가 수행하는 ***책임***의 집합은 객체가 협력 안에서 수행하는 ***역할*** (role) 을 암시함
- ***협력*** (collaboration)에 참여하는 객체들이 너무도 유사하게 협력해서 하나의 협력으로 다루고 싶을 때, ***역할*** (role)을 사용하여 하나의 협력으로 추상화 할 수 있음
  - 동일한 역할을 수행하는 객체들이 동일한 책임의 집합을 수행할 수 있다는 의미

조영호, 객체지향의 사실과 오해, 위키북스, 2015

Q&A