constructor

핵심 정리

- 상속과 생성자
  - ⇨ 파생 클래스 생성자에서는 기반 클래스의 생성자를 호출해야 한다.
  - ⇨ 기반 클래스 생성자를 호출하는 코드를 만들지 않으면, 컴파일러가 추가해 준다.
  - ⇨ 컴파일러가 추가하는 코드는 "항상 기반 클래스의 디폴트 생성자를 호출" 한다.

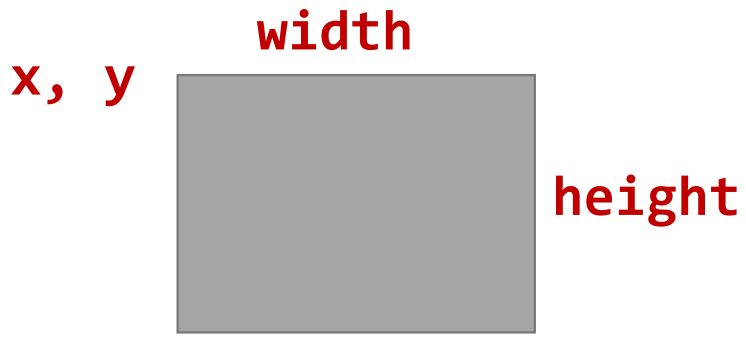
- 기반 클래스에 디폴트 생성자가 없는 경우.
  - ⇨ 파생 클래스에서 반드시 기반 클래스 생성자를 명시적으로 호출해야 한다.

핵심 정리

- **protected 생성자의 의미**
  - ⇨ 자신의 객체는 생성할 수 없지만(추상적 존재)
  - ⇨ 파생 클래스의 객체는 생성할 수 있도록 하겠다는 의도.

shape example

핵심 정리



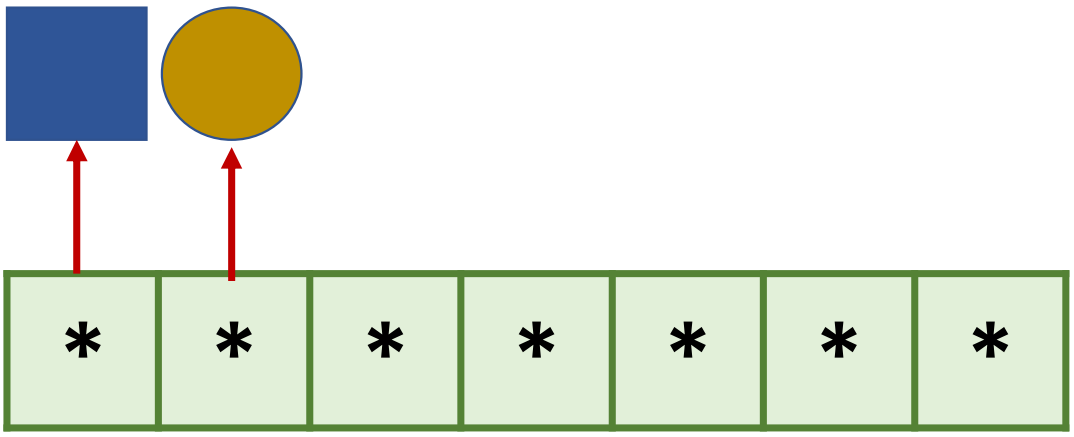
- 핵심 1. 모든 도형을 타입으로 설계 한다.

  ⇨ 핵심만 설명하기 위해 멤버 데이타와 생성자 생략

- 핵심 2. 기반 클래스가 있다면 모든 도형을 하나의 컨테이너에 보관할 수 있다.

  ⇨ Shape 도입.

핵심 정리




```
std::vector<Shape*> v;
```

- 기반 클래스 타입의 포인터로는 파생 클래스의 고유 멤버에 접근할 수 는 없다.

- 해결책

  ① 기반 클래스 포인터를 파생 클래스 타입으로 캐스팅.

  ② 기반 클래스인 Shape 에도 draw() 를 제공한다.

- 정답은 2 번

  ⇨ 캐스팅하는 것은 왜 나쁜 까요 ?

  ⇨ 이어지는 강의에서 설명.

핵심 정리

- 핵심 3. 모든 파생 클래스에 공통의 특징은 반드시 기반 클래스에도 있어야 한다.
  - ⇨ draw() 는 shape 에도 있어야 한다.
  - ⇨ 그래야 기반 클래스 포인터 타입으로 해당 특징을 사용할 수 있다.
  - ⇨ 문법적인 규칙이 아닌 디자인 관점에서의 규칙

- 핵심 4. 기반 클래스 함수 중 파생 클래스가 재정의하게 되는 것은 반드시 가상함수로 만들어라.
  - ⇨ “가상함수가 아니면 재정의 하지 말라” 라는 격언도 존재

shape #2

핵심 정리

- **다형성(polymorphism)**
  - ⇨ 동일한 표현식이 상황에 따라 다르게 동작하는 것
  - ⇨ “**p->draw()**” 는 상황(실제 가리키는 객체의 종류)에 따라 다르게 동작한다.
  - ⇨ “**Triangle**” 등의 새로운 도형타입이 추가 되어도 p->draw() 는 “변경할 필요 없다.”

- **OCP 원칙**
  - ⇨ 기능확장에는 열려 있고(Open, 새로운 클래스(모듈)이 추가되어도)
  - ⇨ 코드 수정에는 닫혀 있어야(Close, 기존 코드는 수정되지 말아야 한다)
  - ⇨ 이론(Principle)
  - ⇨ **Open-Close Principle (개방 폐쇄의 법칙)**

핵심 정리

- k 번째 도형의 복사본을 만드는 방법

- 방법 1. dynamic_cast 등으로 타입을 조사
  - ⇨ 새로운 도형이 추가되면 코드가 수정되어야 한다.
  - ⇨ OCP 위반

- 방법 2. clone() 가상함수
  - ⇨ 새로운 도형이 추가되어도 코드를 수정할 필요 없다.
  - ⇨ OCP 만족

- Refactoring
  - ⇨ **“Replace Conditional With Polymorphism”**
  - ⇨ 제어문 대신 다형성(가상함수)를 사용하라는 의미

핵심 정리

- 디자인 패턴이란 ?

  ⇨ 특정 문제를 해결하기 위해 만들어진 코딩 패턴에 이름을 부여 한 것.

- **prototype 패턴**

  ⇨ 기존 객체를 복사해서 새로운 객체를 만드는 패턴.

핵심 정리

가상함수 vs 순수 가상함수

가상함수	파생 클래스가 override 하지 않아도 된다. override 하지 않으면 기본 구현 제공
순수 가상함수	파생 클래스는 반드시 override 해서 구현을 제공해야 한다.

Template method

핵심 정리

- GUI 환경에서 윈도우에 그림 그리기
  - ⇨ 대부분의 라이브러리에는 “그림을 그리기 위한 클래스”를 제공
  - ⇨ “화면 깜박임 등을 방지(flicker free)” 하기 위해 다양한 방법을 제공(더블 버퍼링 등)

```
PainterPath path;
path.begin();

// path 객체를 사용해서
// 원하는 그림을 그린다.

path.end();

Painter surface;
surface.draw_path(path);
```

핵심 정리

```
void draw()
{
    PainterPath path;
    path.begin();

    // path 객체를 사용해서
    // 원하는 그림을 그린다.

    path.end();

    Painter surface;
    surface.draw_path(path);
}
```

함수의 전체적인 구조 (알고리즘의 처리 과정) 는 변하지 않는다.

이 부분은 파생 클래스 (도형) 마다 변경되어야 한다.

- 핵심
  - ⇨ "변하지 않은 코드 내부에 있는 변해야 하는 코드를 찾는다."
  - ⇨ "변해야 하는 코드는 가상함수로 분리" 한다.
  - ⇨ 파생 클래스는 알고리즘의 처리 과정을 물려 받으면서 가상함수 재정의를 통해서 "변경이 필요한 부분만 다시 만들 수" 있다.

Strategy

핵심 정리

- **Edit 클래스**

  ⇨ 사용자에게 입력을 받을 때 사용하는 GUI Widget(컨트롤)

- Edit 를 사용해서 나이를 입력 받고 싶다.

  ⇨ “**숫자만 입력 되도록 제한(validation)**” 해야 한다.

- Edit 의 “**Validation 정책은 변경될 수 있어야**” 한다.

  ⇨ Edit 클래스를 어떻게 디자인 해야 할까 ?

핵심 정리

- Validation 정책 변경 방법 #1
  - ⇨ 변하는 것을 가상함수로 분리
  - ⇨ **template method** 패턴.

- iscomplete(data)
  - ⇨ 입력 값이 완성되었는지 확인

입력된 값이 이메일 형식이 아니면
버튼을 **Disable** 되어야 한다.

핵심 정리

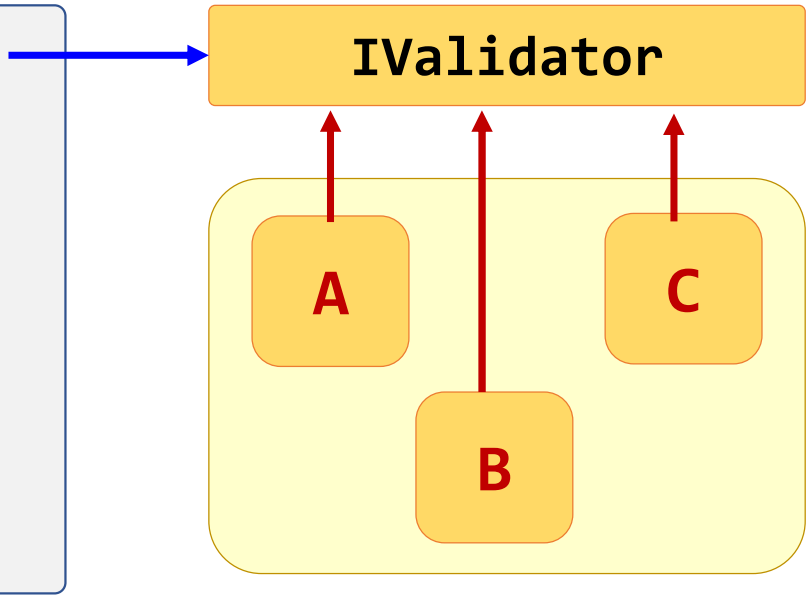
- Edit 의 Validation 정책 교체 방법
  - ⇨ **template method** 가 최선의 방법일까 ?
  - ⇨ 다른 방법은 없을까 ?

핵심 정리

- 변하는 것을 분리할 때 사용하는 2가지 방법
  - ⇨ 변하는 코드를 가상함수로 분리
  - ⇨ 변하는 코드를 다른 클래스로 분리

인터페이스를 먼저 만들고 **Edit** 에서 약한결합으로
다양한 **Validation** 정책 클래스 사용

```
class Edit
{
    get_text()
    {
        ————————
        ————————

    }
};
```



validation 정책을
담은 다양한 클래스

Policy Base Design

핵심 정리

- 단위 전략 디자인 (Policy Base Design)
  - ⇨ 전통적인 객체지향 디자인 패턴 23개 분류에는 포함되지 않음.
  - ⇨ C++ 진영에서 널리 사용되는 디자인 기술
  - ⇨ STL 의 구현에서 많이 사용
  - ⇨ “전략 패턴(strategy) + 성능 향상”

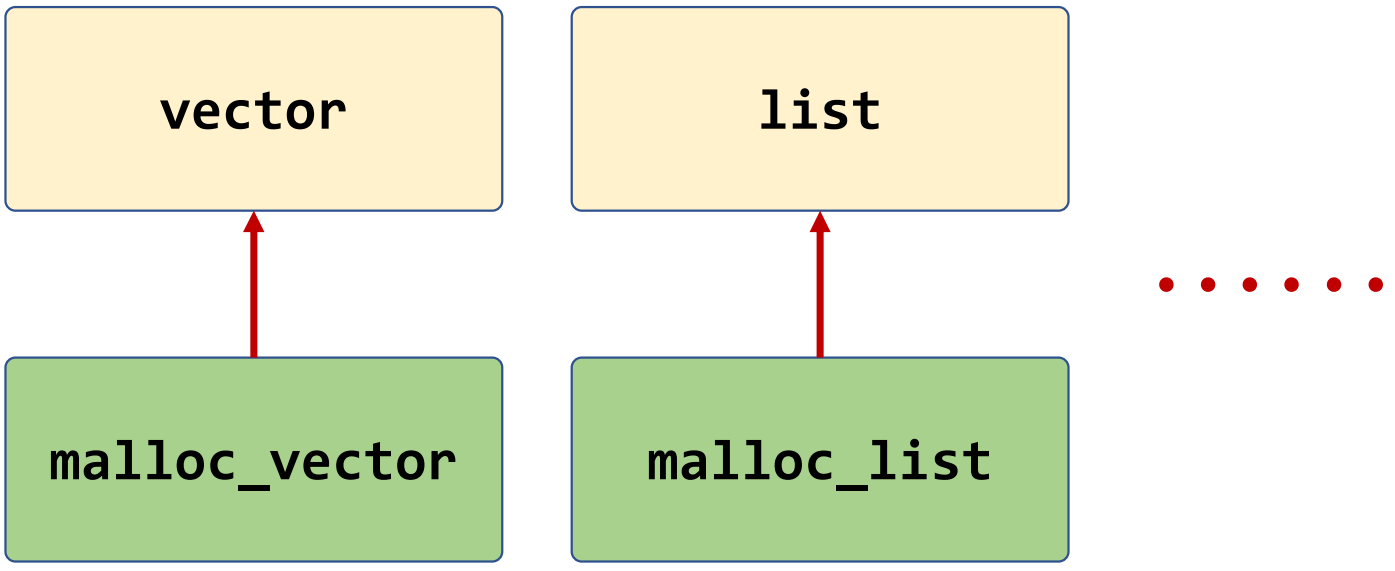
핵심 정리

- **vector 클래스의 멤버 함수에서 메모리를 할당/해지 해야 한다면 ?**
  - ⇨ C++ 에는 메모리를 할당/해지 하는 방법이 많이 있다.
  - ⇨ new, malloc, system call, memory pooling
  - ⇨ vector 의 사용자가 메모리 할당 방식을 변경할 수 있도록 만들 수 없을까 ?

핵심 정리

- 방법 1. 변하는 것을 가상함수로!
  - ⇨ resize 알고리즘은 변하지 않지만, 메모리 할당/해지 방식은 교체 가능해야 한다.
  - ⇨ template method 패턴의 모양을 활용
  - ⇨ 메모리 할당/해지 하는 부분을 가상함수로 분리

- 단점
  - ⇨ 메모리 할당하는 코드를 재사용하기 어렵다.
  - ⇨ vector 뿐 아니라 list, set, map 등도 메모리 할당 방식을 변경하려면 파생클래스를 만들어야 한다.



핵심 정리

- 방법 2. 변하는 것을 다른 클래스로
  - ⇨ strategy 패턴

- 장점
  - ⇨ 메모리를 할당하는 코드(정책)을 다른 컨테이너에서도 사용가능 하다.

- 단점
  - ⇨ 할당/해지를 위한 함수가 가상함수 이므로 느리다.

- 다른 방법은 없을까 ?
  - ⇨ policy base design

핵심 정리

- **방법 3. policy base design**

  ⇨ 메모리 할당/해지를 담은 정책 클래스(메모리 할당기)를 "인터페이스 기반으로 교체 하지 말고 템플릿 인자를 사용해서 교체"

- **strategy vs policy base design**

  ⇨ 클래스가 사용하는 정책(알고리즘)을 다른 클래스로 분리 하는 것은 유사.

  ⇨ 하지만 어떻게 교체할 것인가의 차이점.

strategy	인터페이스를 사용해서 교체 가상함수 기반이므로 느리다. 실행시간에도 교체 가능하다.
policy base	템플릿 인자를 사용해서 교체 가상함수가 아닌 인라인 치환도 가능. 실행시간에도 교체 할 수 는 없다.

핵심 정리

- **STL 의 대부분의 컨테이너는**
  - ⇨ 메모리 할당기를 템플릿 인자로 전달 받아서 사용.

```
template<typename T,
         typename Allocator = std::allocator<T>>
class vector;

template<typename T,
         typename Allocator = std::allocator<T>>
class list;
```

- **컨테이너의 메모리 할당 방식을 교체 하려면**
  - ⇨ 사용자 정의 메모리 할당기를 만들어서 템플릿 인자로 전달.
  - ⇨ 메모리 할당기 만들때 **지켜야 하는 규칙**을 알아야 한다.
  - ⇨ cppreference.com 의 **named requirement** 에서 **“Allocator”** 참고(예제)

Composite

핵심 정리

- C 언어에서 프로그램 이란 ?
  - ⇨ main 함수의 1번째 줄부터 “순차적으로 실행되는 실행 흐름”.
  - ⇨ 실행순서를 변경하려면 제어문, 반복하려면 반복문을 사용한다.
  - ⇨ 프로그램의 기본 단위는 “함수” 이다

- 장점
  - ⇨ 프로그램의 구조가 이해 하기 쉽다.
  - ⇨ CPU 의 동작 방식과 동일하다. 빠르다.
  - ⇨ 메모리 사용량도 작다.

- 단점
  - ⇨ 확장성이 부족하고, 변화에 유연하지 않다.
  - ⇨ 유지 보수가 쉽지 않다.
  - ⇨ 새로운 메뉴 추가되거나, 하위 메뉴를 추가해 한다면 ?

Step 1. MenuItem 클래스 만들기

- 객체지향 프로그램 에서 프로그램은
  ⇨ 객체들의 집합
  ⇨ 프로그램의 기본 요소는 "함수" 가 아닌 "클래스" 이다.

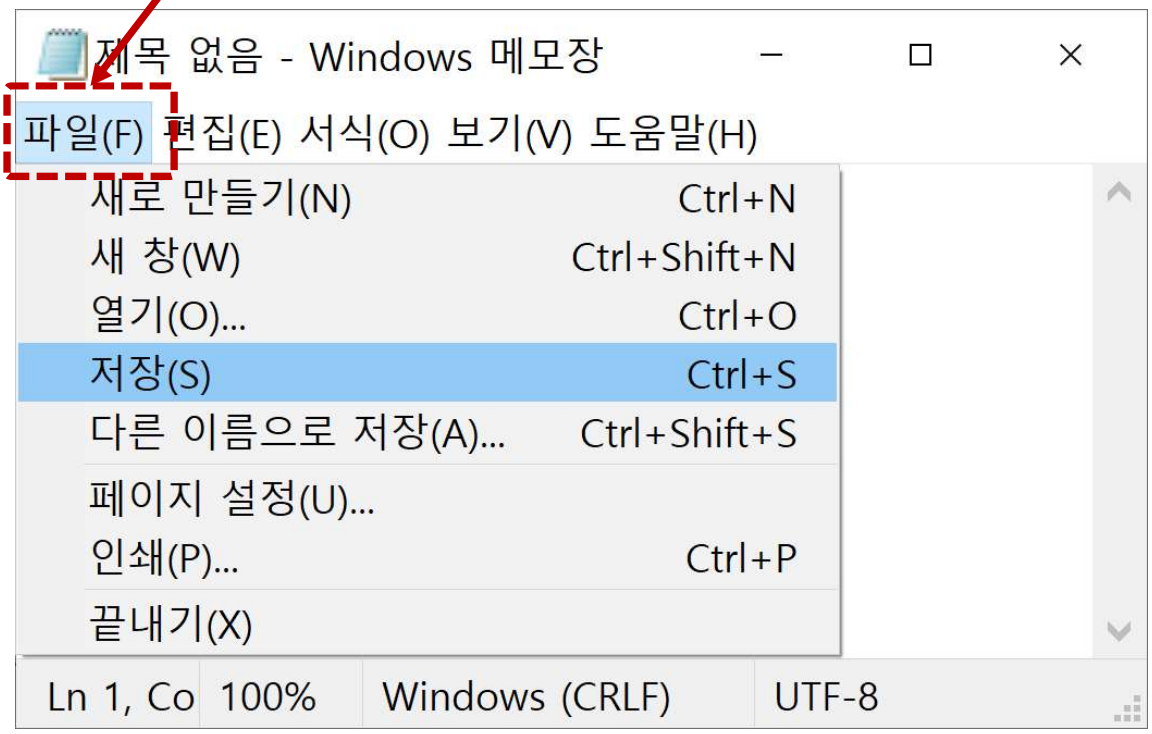
- 메뉴의 각 항목을 나타내는 "**MenuItem**" 클래스 설계

각각의 메뉴 항목을
객체로 관리
(8개의 객체)

Step 2. PopupMenu 만들기

파일 메뉴 를 선택하면

하위 메뉴가 나타난다.



- PopupMenu

  ⇨ 선택 했을 때 하위 메뉴를 열어주는 메뉴

  ⇨ 타이틀이 있고(**std::string title**)

  ⇨ 여러 개의 하위 메뉴(MenuItem)를 보관 해야 한다

  (**std::vector<MenuItem*>**)

Step 2. PopupMenu 만들기

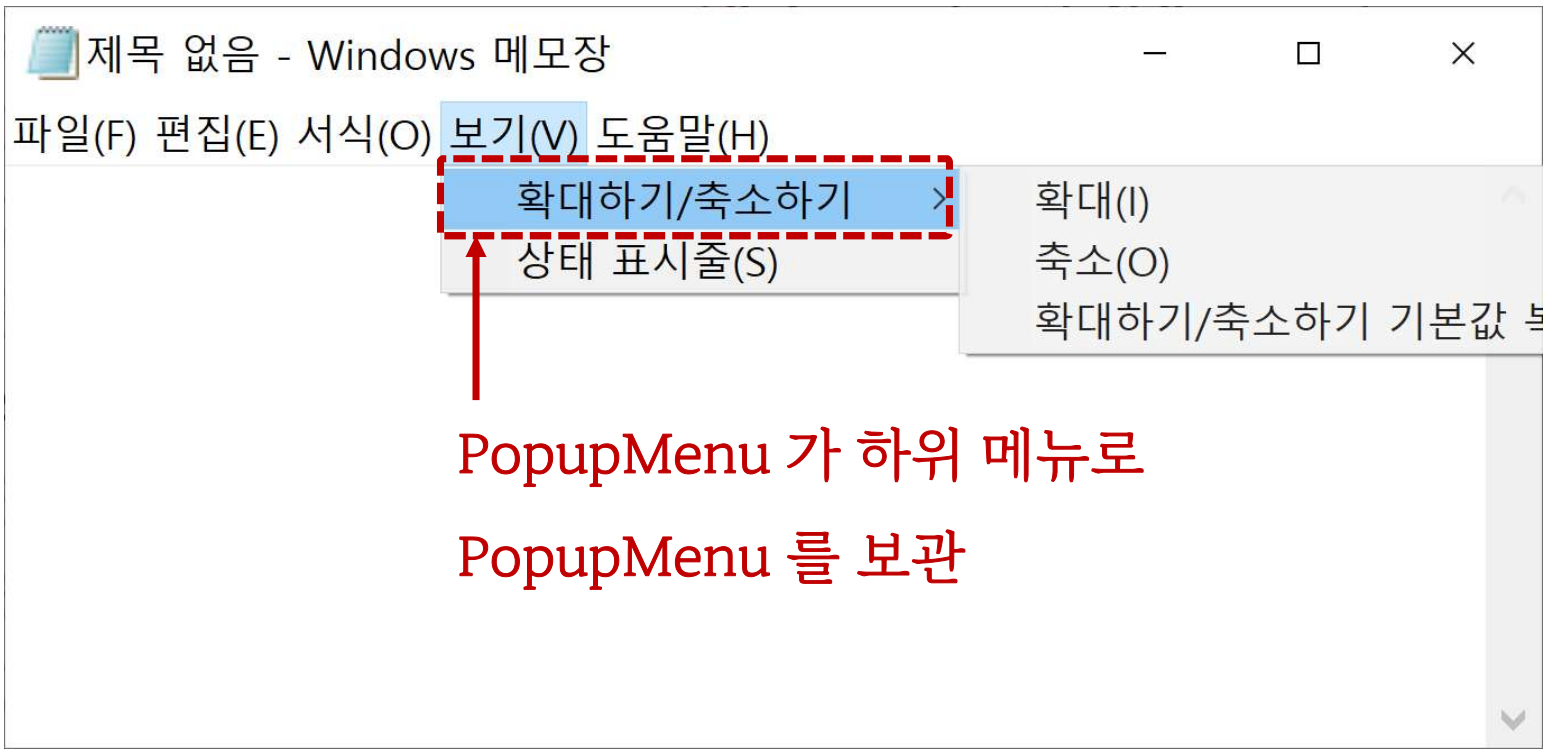
- PopupMenu 를 선택 했을 때 해야 할 일

  ⇨ 하위 메뉴를 보여 주고

  ⇨ 사용자 선택을 입력 받음.

```
1. 김밥
2. 라면
3. 종료
메뉴를 선택해 주세요 >>
```

Step 3. Menu 예제에 Composite 패턴 적용



핵심 정리

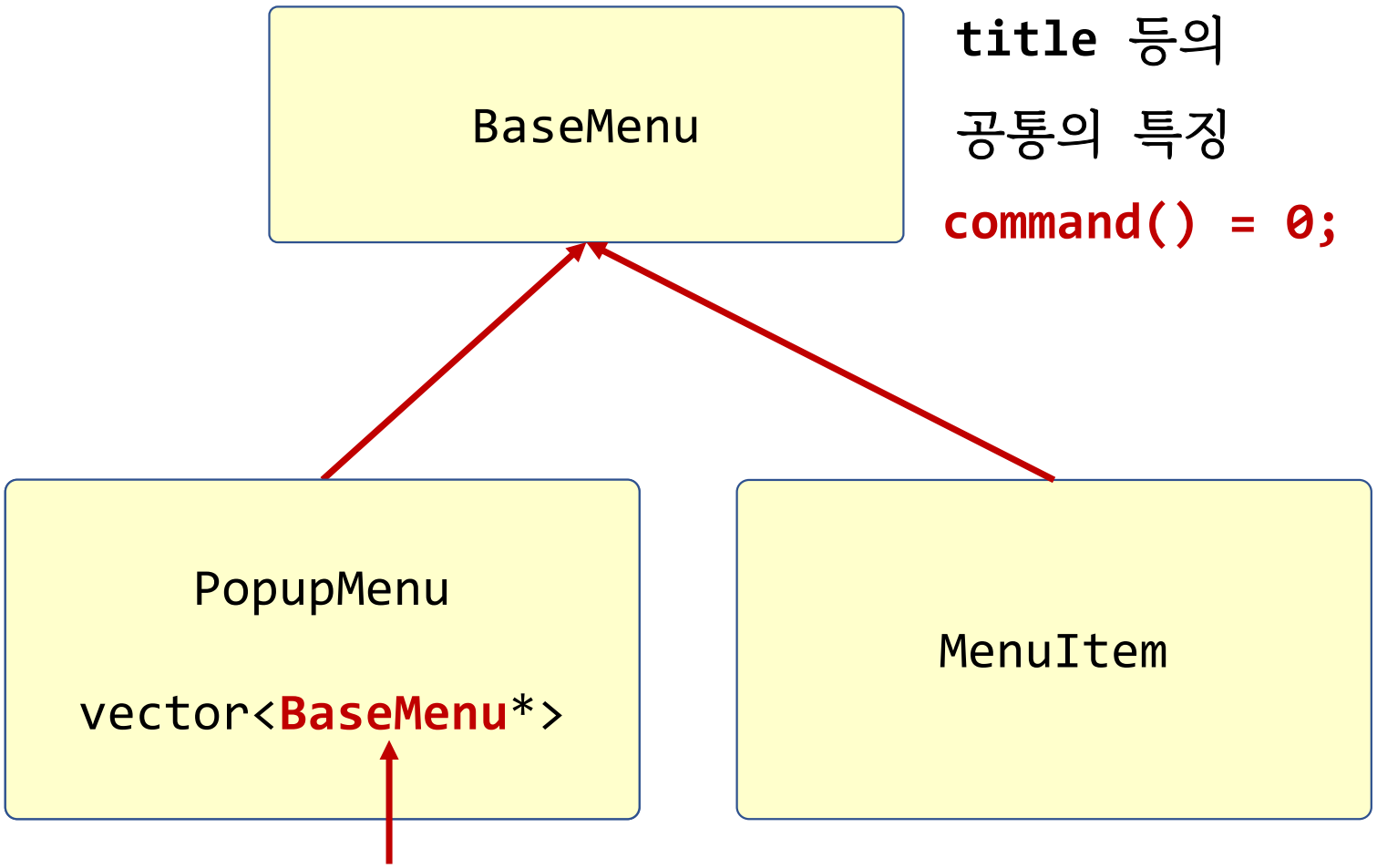
PopupMenu

vector<**MenuItem***>

MenuItem

MenuItem 객체만 보관 가능하다.

핵심 정리

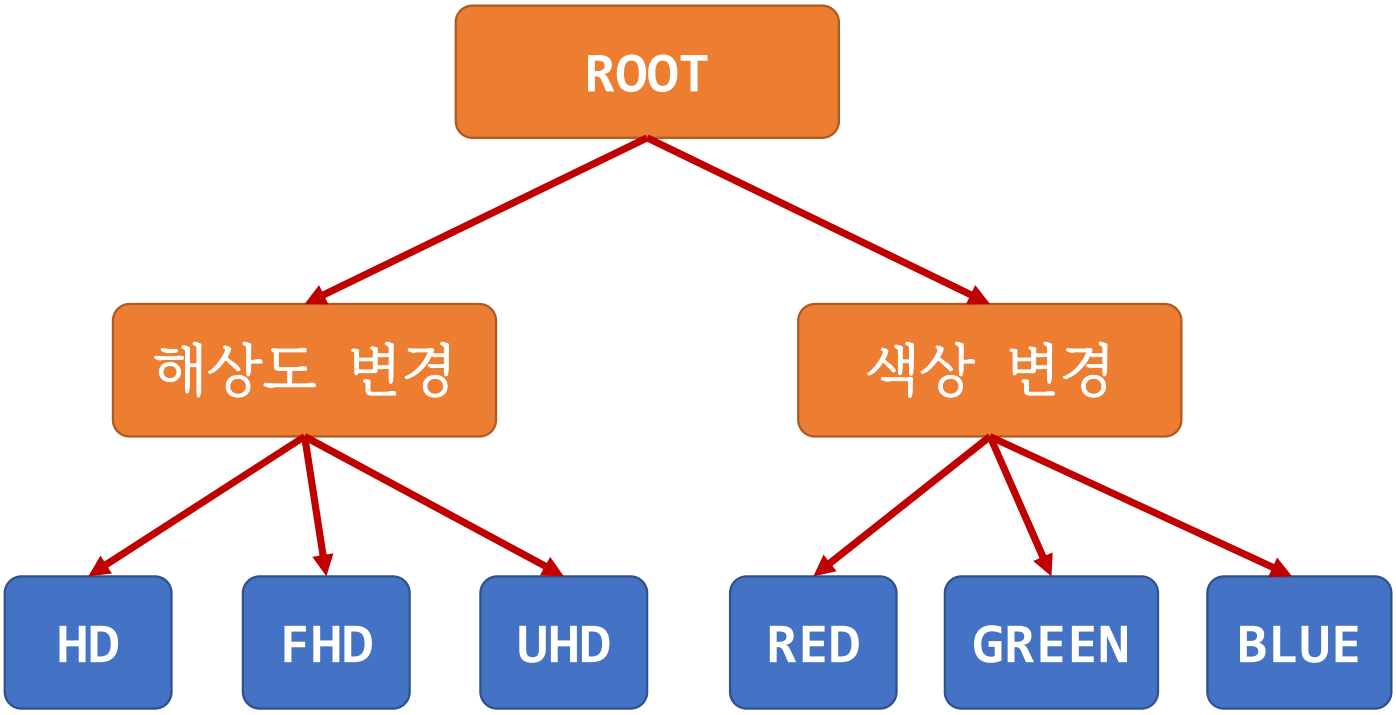


MenuItem 객체와

PopupMenu 객체를 모두 보관

(재귀적인 포함 관계)

핵심 정리



- Composite 패턴의 의도

  ⇨ 부분과 전체의 계층을 표현하기 위해 복합객체를 트리 구조로 만든다.

- 장점

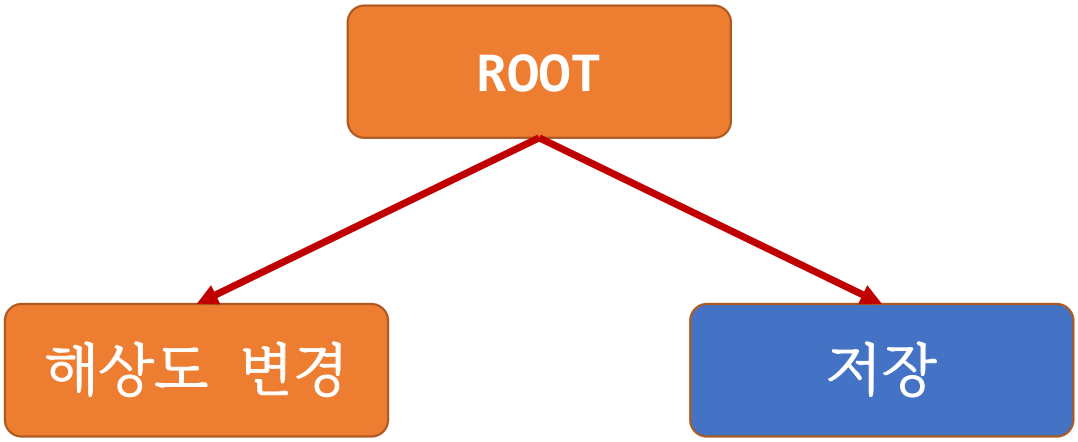
  ⇨ 새로운 메뉴를 추가하거나

  ⇨ 메뉴의 구조를 변경하는 것이 쉽다.

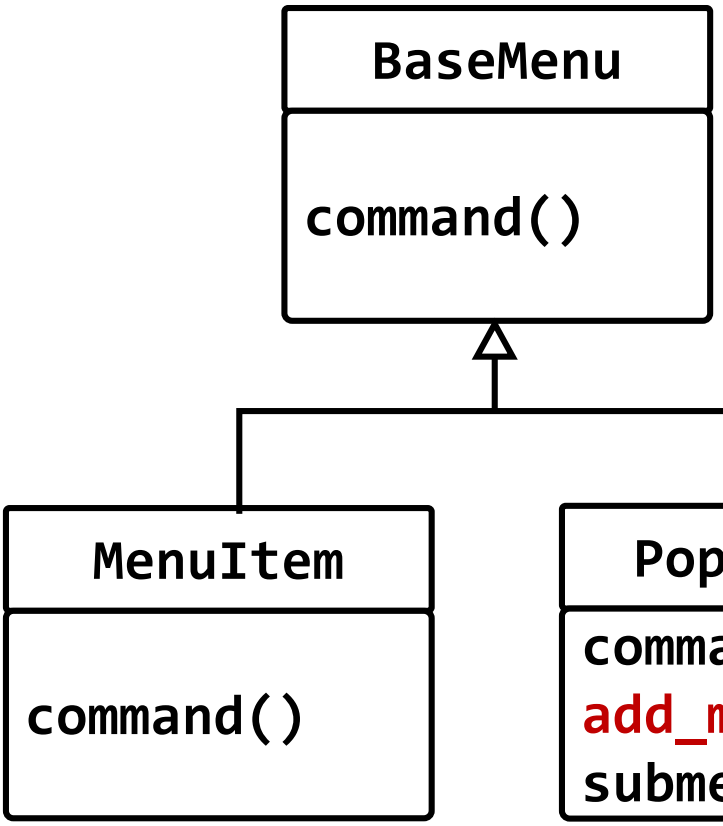
핵심 정리

- 객체지향 프로그래밍에서 프로그램이란 ?
  - ⇨ 객체를 만들고
  - ⇨ 객체 간의 관계를 설정하고
  - ⇨ 객체 간의 메세지를 주고 받는 과정
    (서로 간의 멤버 함수를 호출한다는 의미)

핵심 정리

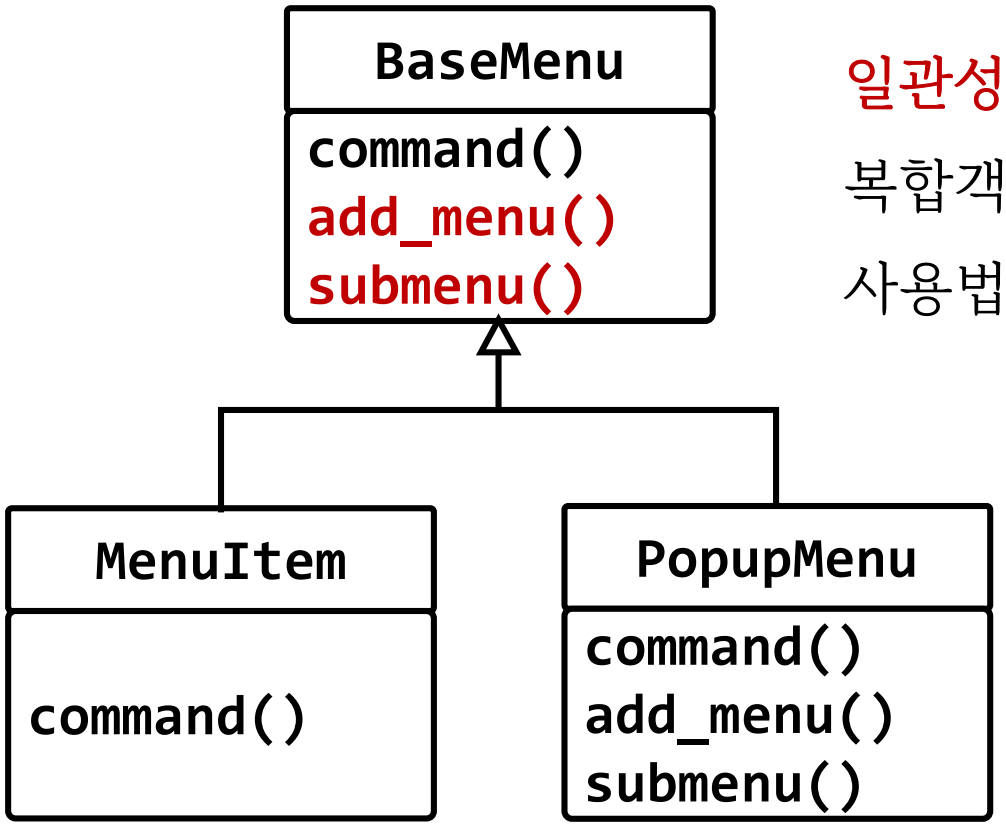


핵심 정리



타입 안정성

(type safety)



일관성(uniformly)

복합객체와 개별객체의 사용법이 동일

Decorator

핵심 정리

- Image 클래스
  - ⇨ 그림을 Load 해서 화면에 출력하는 클래스
  - ⇨ 파일 뿐 아니라 인터넷 다운로드도 지원.

- Load된 그림에 "액자나 말풍선 등의 효과를 추가" 하고 싶다

- 상속을 사용하면 어떨까 ?
  - ⇨ Image 클래스로 부터 모든 기능을 물려 받고, 액자(또는 말풍선)를 그리는 기능을 추가.

핵심 정리

- 상속을 사용한 서비스 추가 특징

① 객체가 아닌 클래스에 기능을 추가 한 것

```
Image img("www.image.com/car.jpg");
```

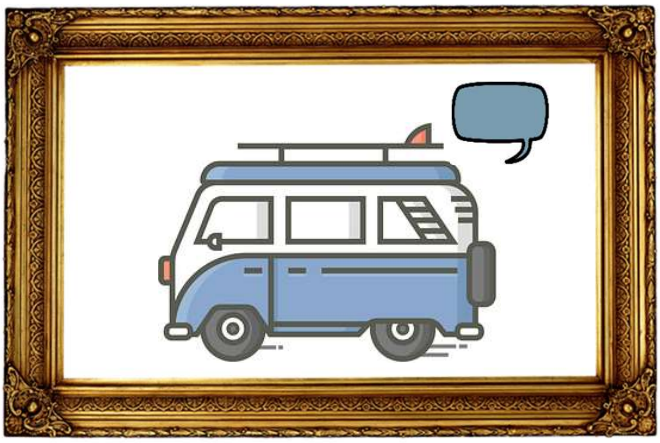
이 순간 이미 그림을 Load 되었고,

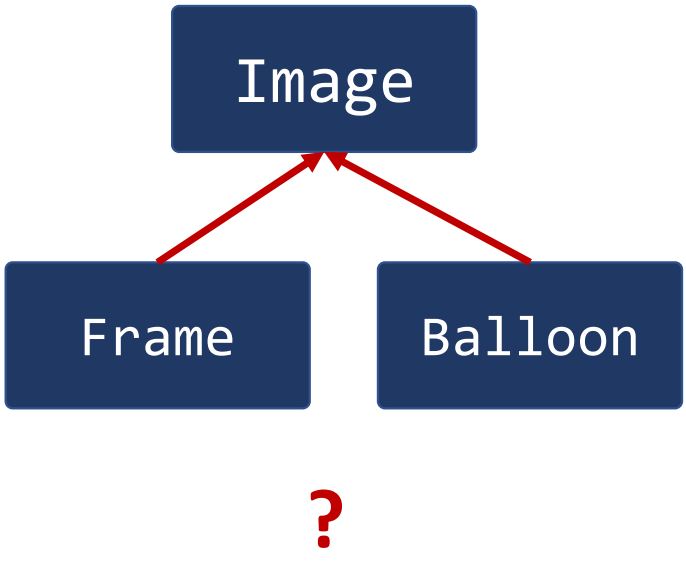
img 객체가 관리 한다.

img 라는 객체에 새로운 기능을 추가할 수 없을까 ?

상속은 **Image** 라는 클래스에 서비스를 추가 한다.

② 여러 개의 서비스를 중복해서 추가하기 어렵다.





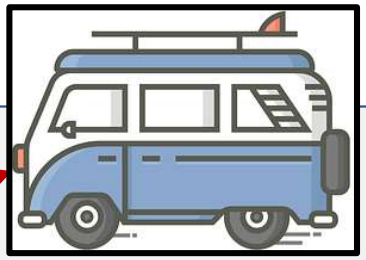
?

핵심 정리

- 클래스가 아닌 객체에 기능을 추가하려면

  ⇨ "상속이 아닌 포함" 을 사용한 기능의 추가

이미 생성된 객체



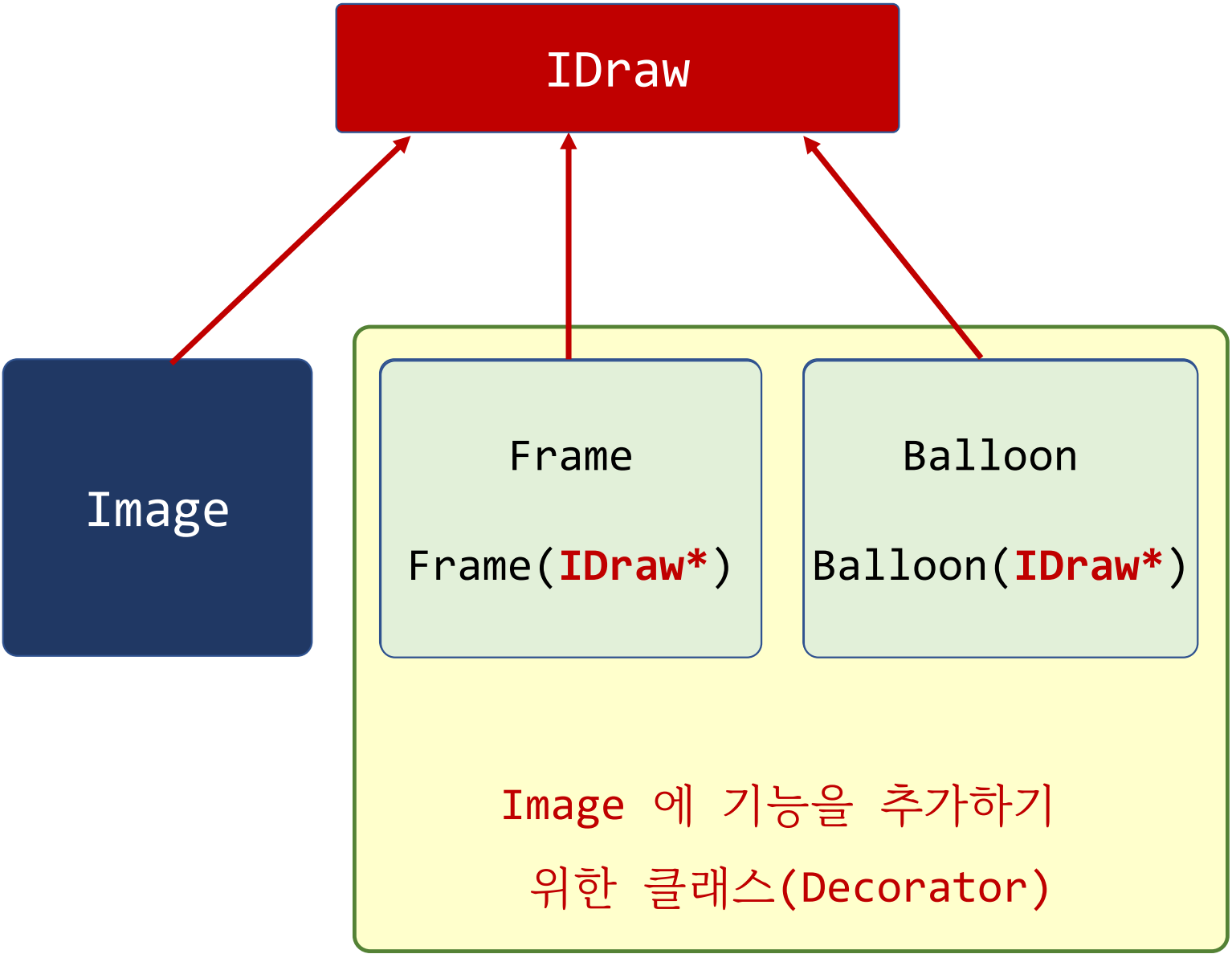
```
class Frame : public Image
{
    Image* img;

public:
    Frame(Image* img) : img(img) {}

    void draw() const
    {
        // 추가할 기능 작성

        // 객체(img)의 원래 기능 사용
        img->draw();
    }
};
```

포인터이므로 이미 생성된 객체를 가리키겠다는 것

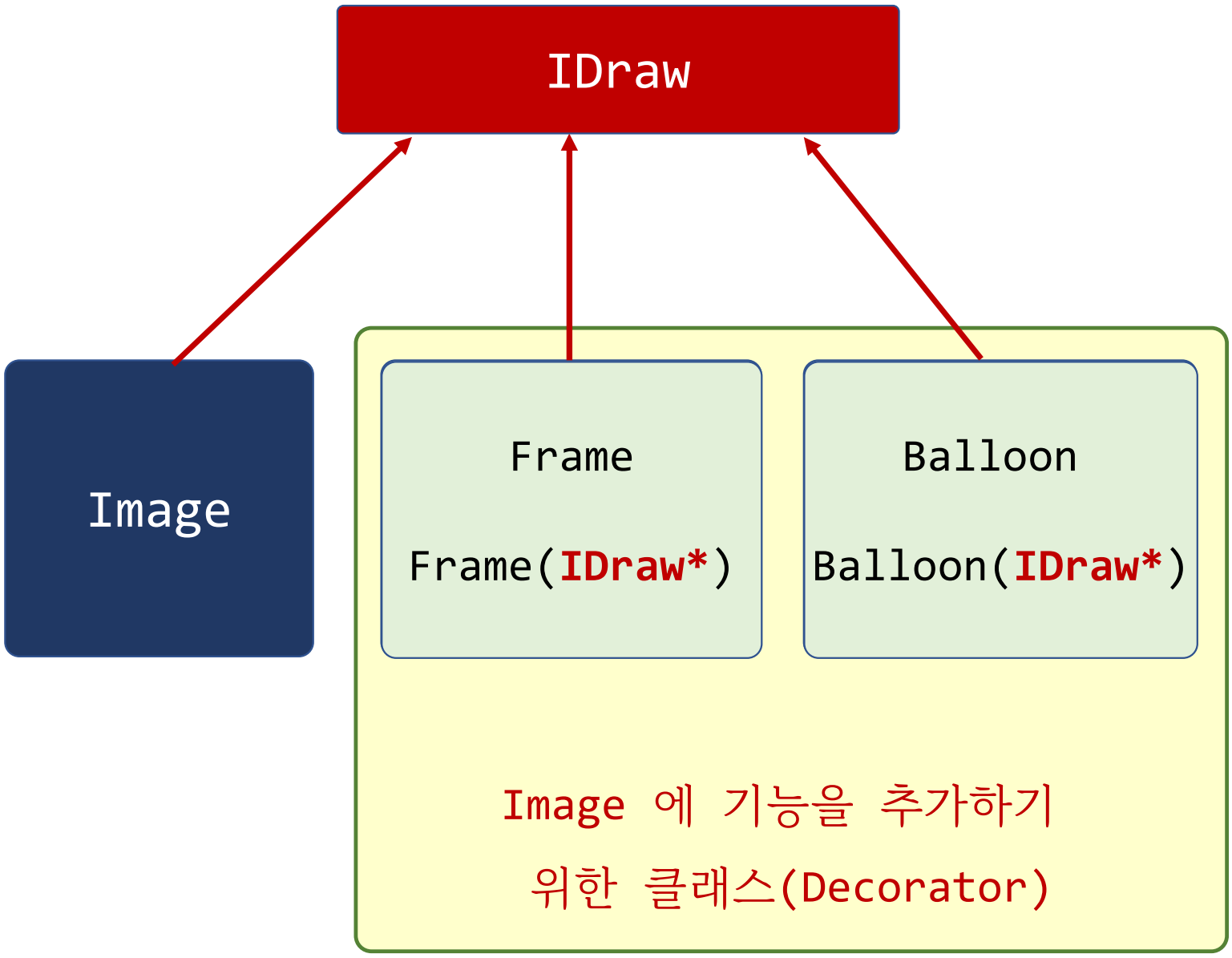
핵심 정리

- 객체에 기능을 중복으로 계속 추가.

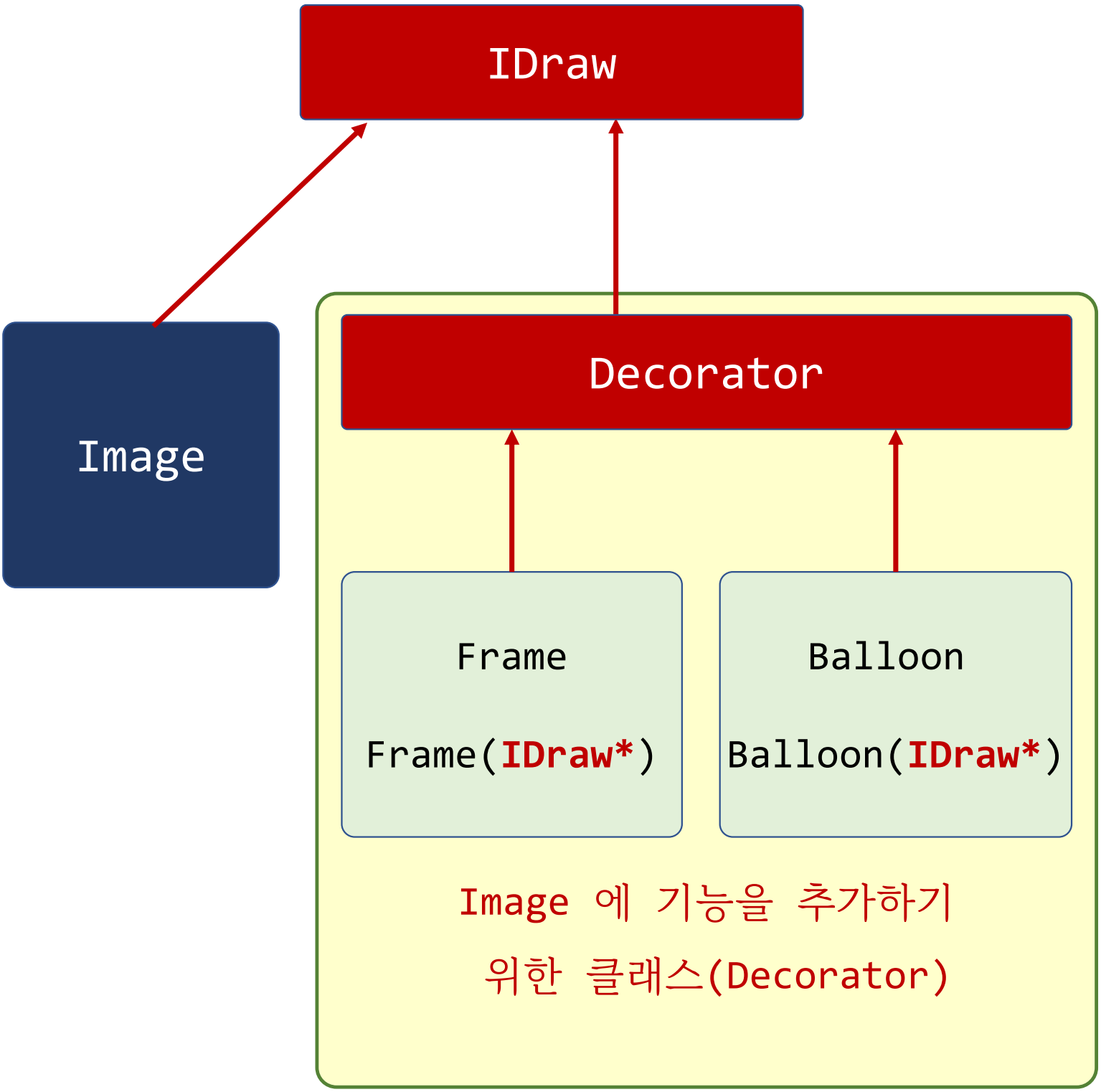
Image 클래스와 기능 추가 클래스를 위한 공통의 인터페이스



핵심 정리



핵심 정리



decorator example

핵심 정리

- FileStream 클래스
  - ⇨ 특정 파일에 입출력을 위한 다양한 기능을 지원하는 클래스

- NetworkStream, PipeStream
  - ⇨ 다양한 장치에 입출력을 위한 클래스들
  - ⇨ 모든 Stream 클래스는 사용법이 동일하게 설계되는 것이 좋다.
  - ⇨ 공통의 기반 클래스 설계

- Stream 클래스
  - ⇨ 모든 입출력 Stream 타입의 기반 클래스

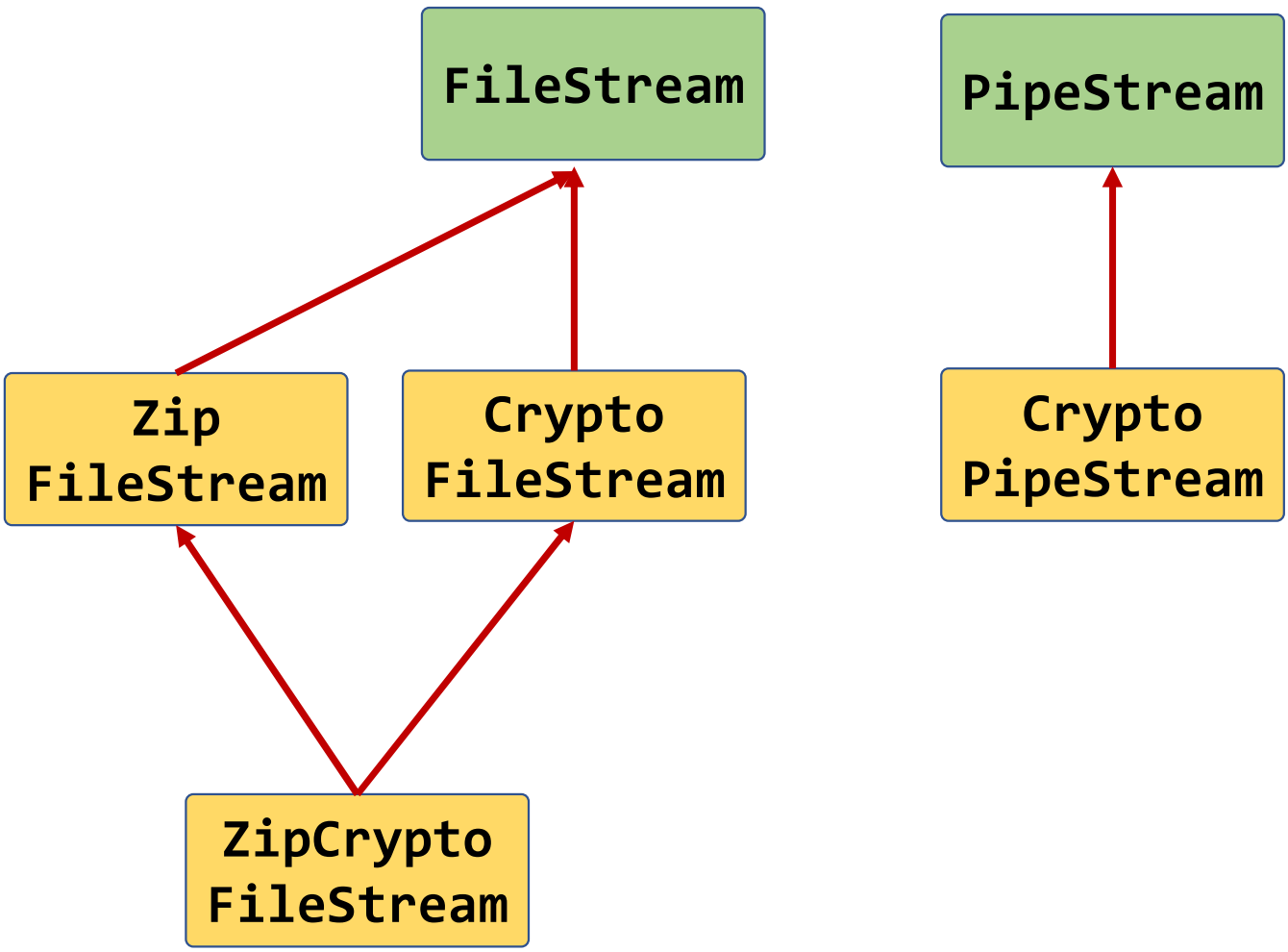
핵심 정리

- 파일에 데이타를 write 할 때 암호화 하는 기능이 필요하다.

  ⇨ 어떻게 설계하는 것이 좋을까 ?

핵심 정리

- 방법 1. 상속을 사용한 기능의 추가
  - ⇨ 단점 1. 모든 Stream 클래스의 파생 클래스가 필요
  - ⇨ 단점 2. 기능을 중첩해서 추가 하기 어렵다.



핵심 정리

- decorator 패턴을 사용하면 어떨까 ?
  - ⇨ 객체에 중첩해서 기능을 추가할 수 있다.
  - ⇨ **C# 언어의 Stream 클래스의 원리.**

Adaptor

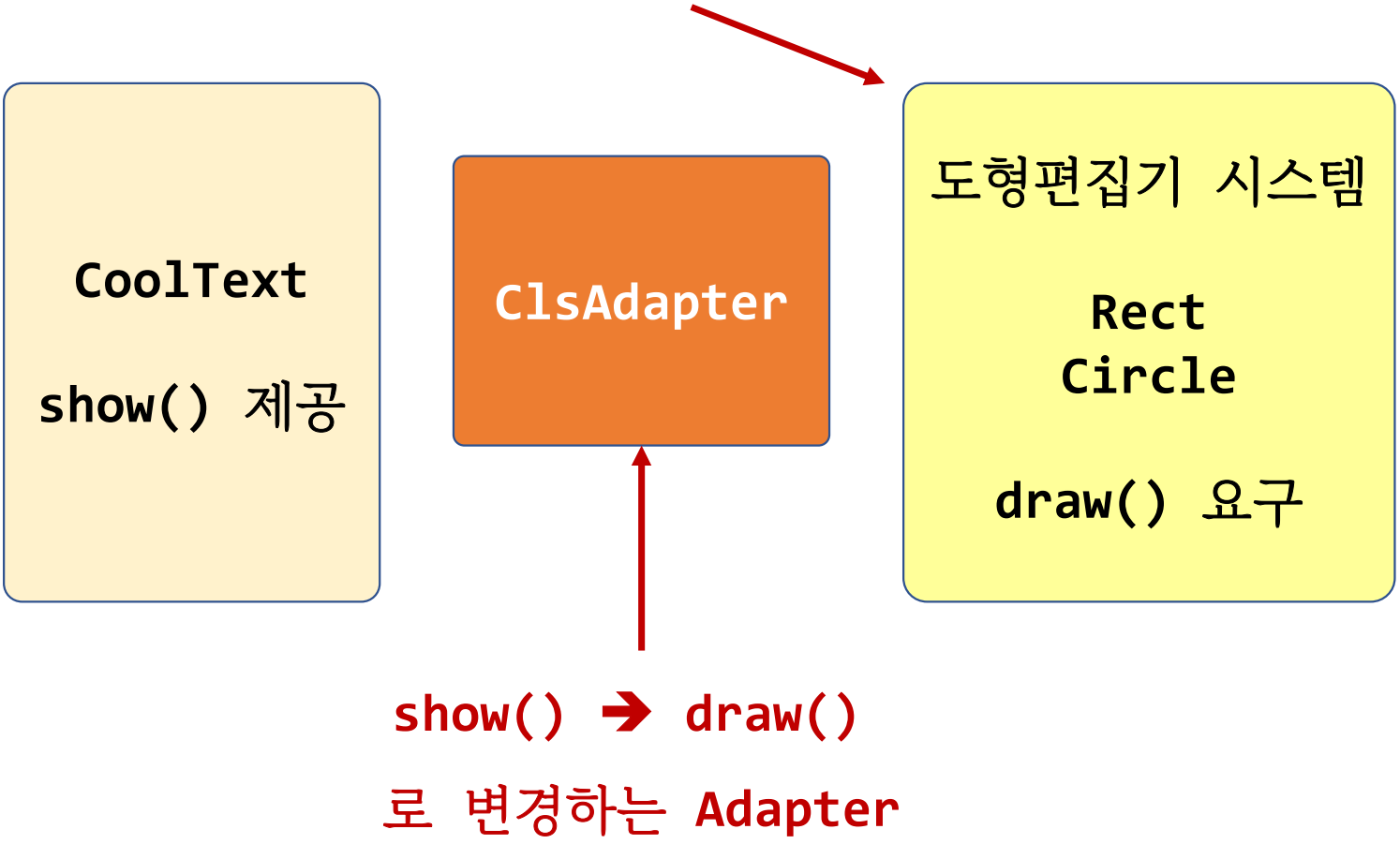
핵심 정리

- **CoolText**
  - ⇨ 문자열을 보관했다가 화면에 출력해주는 클래스.
  - ⇨ 문자열을 이쁘게 출력할 수 있는 다양한 기능을 제공.
  - ⇨ 예전 부터 가지고 있는 클래스

- 도형편집기 시스템 에 Rect, Circle 외에 "문자열 편집 출력하는 클래스를 추가"하고 싶다.
  - ⇨ 예전 부터 가지고 있던 "**CoolText**" 를 사용할 수 있을까 ?

핵심 정리

- 도형편집기에서 사용하려면
  - ⇨ "**Shape 클래스로 부터 상속**" 받아야 한다.
  - ⇨ "**draw() 함수를 제공**"해야 한다.



핵심 정리

1 **CoolText** 가 가진 문자열을 관리/출력 하는 모든 기능을 물려 받고

```
class ClsAdapter : public CoolText,
                   public Shape
{
public:
    ClsAdapter(const std::string& text)
        : CoolText(text) {}

    void draw() override { CoolText::show();}
};
```

2 도형편집기 시스템의 요구사항도 만족

show() 라는 이름을 **draw()** 이름으로 변경한 것

핵심 정리

- Adapter 의 종류

클래스 어답터	클래스의 인터페이스를 변경
객체 어답터	객체의 인터페이스를 변경

핵심 정리

- Object Adapter

이미 생성된 **CoolText** 객체

```
class ObjAdapter : public Shape
{
    CoolText* ct;
public:
    ObjAdapter(CoolText* ct) : ct(ct) {}

    void draw() override { ct->show();}
};
```

기존에 존재하는 객체의 인터페이스 변경

STL & adapter

핵심 정리

- **STL 과 Adapter 패턴**
  - ⇨ C++ 표준 라이브러리인 STL 에도 adapter 패턴을 사용한 설계가 많이 있음.
  - ⇨ container adapter
  - ⇨ iterator adapter
  - ⇨ range adapter
  - ⇨ ...

핵심 정리

- STL 에는 다양한 sequence container 가 제공됨.
  ⇨ std::list, std::vector, std::deque 등

- STL 에는 std::stack 도 제공된다.
  ⇨ 어떻게 구현했을까 ?

- 방법 1. stack 의 모든 기능을 직접 구현하는 방법
  ⇨ 메모리 관리 등의 모든 기능을 직접 구현해야 한다.
  ⇨ 코드 메모리도 증가하게 된다.

- 방법 2. 기존에 존재하는 sequence container(list등)의 함수 이름만 변경해서 stack 처럼 보이게 한다.
  ⇨ adapter 패턴
  ⇨ STL 이 사용한 방법
  ⇨ 이미 STL 있지만, 직접 만들어가면서 구현원리를 이해

핵심 정리

- **방법 1. 상속을 사용한 stack adapter**
  - ⇨ std::list 로 부터 상속받아서
    push_back ➔ push 로 변경

- 단점
  - ⇨ std::list 로 부터 push_front 등의 함수도 물려 받게 된다.
  - ⇨ 스택은 한쪽 방향으로만 사용해야 한다.
  - ⇨ 좋지 않은 디자인
  - ⇨ java 의 stack 이 vector 로 부터 상속 받고 있다.

핵심 정리

- **방법 2. private 상속을 사용한 stack adapter**
  - ⇨ 기반 클래스의 멤버 함수를 파생 클래스의 내부에서만 사용.
  - ⇨ 기반 클래스 멤버 함수를 파생 클래스가 외부에 노출하지는 않음.
  - ⇨ 구현은 물려 받지만 인터페이스는 물려 받지 않겠다는 의미.
  - ⇨ C++ 진영에서 가끔 사용하는 디자인
  - ⇨ 다른 객체지향 언어에는 없는 C++ 만의 문법

핵심 정리

- **방법 3. 포함을 사용한 stack adapter**

  ⇨ std::list 로 부터 상속 받지 말고 포함을 사용

상속	std::list 의 모든 멤버 함수를 stack 도 외부에 노출 (public 상속)
포함	std::list 의 모든 멤버 함수는 stack 이 내부적으로만 사용.

- **private 상속 vs 포함**

private 상속	std::list 에 **가상함수가 있다면** **stack 이 override 할 수 있다.**
포함	std::list 에 가상함수를 stack 이 override 할 수 없다.

⇨ std::list 에는 가상함수는 없다.

핵심 정리

- **stack 을 만들 때 반드시 std::list 를 사용해야 할까 ?**
  - ⇨ std::vector, std::deque 를 사용하면 안될까 ?
  - ⇨ 사용자가 선택할 수 있게 하면 어떨까 ?

- **단위 전략 디자인 (policy base design)**
  - ⇨ 클래스가 사용하는 정책을 템플릿 인자를 통해서 교체 할 수 있는 기술

- **C++ 표준의 std::stack**
  - ⇨ sequence container 을 stack 처럼 사용할수 있게 하는 adapter
  - ⇨ policy base design 을 사용해서 컨테이너의 종류를 사용자가 결정할 수 있다.
  - ⇨ container 를 지정하지 않으면 std::deque 를 디폴트로 사용.
  - ⇨ **“container adapter”** 라고 부름.

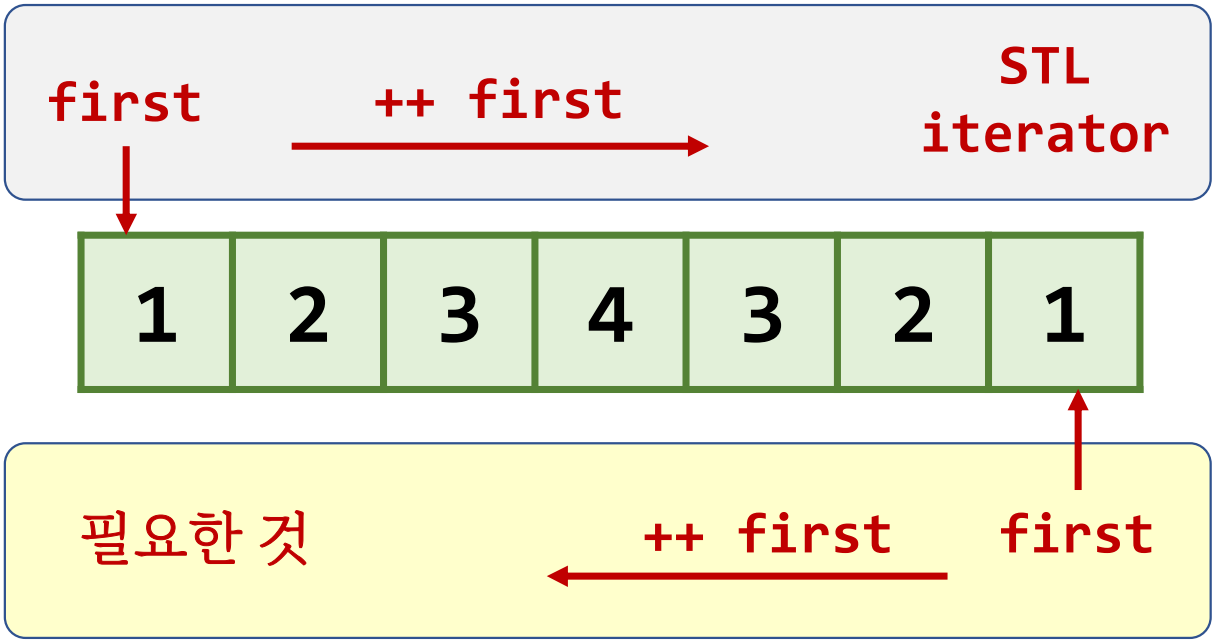
핵심 정리

- **std::stack 은**
  class adapter 일까 ?
  object adapter 일까 ?

클래스 어답터	클래스의 인터페이스를 변경
객체 어답터	객체의 인터페이스를 변경

iterator adapter

핵심 정리



- **std::find 로 검색시 뒤에서 부터 검색할 수 없을까 ?**

  ⇨ std::find 는 1번째 인자로 받은 반복자를 **++로 이동**하면서 검색 한다.

- 방법 1. 거꾸로 동작하는 반복자를 새롭게 만들자

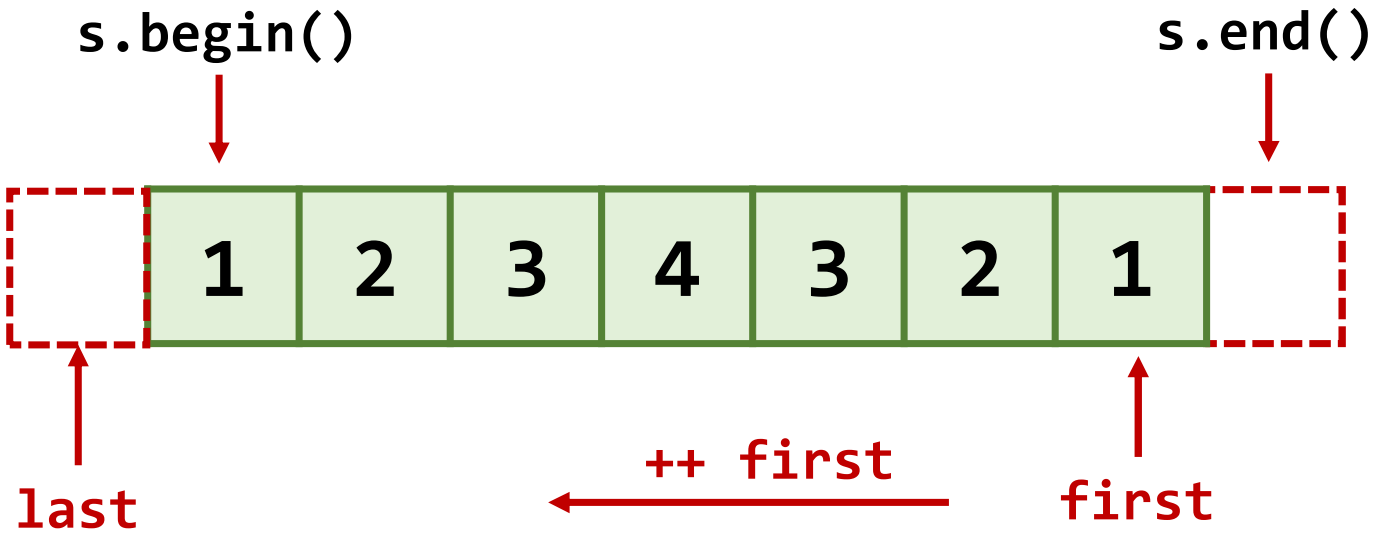
  ⇨ vector 뿐 아니라 list, deque 등 모든 컨테이너에 새로운 반복자를 만들어야 한다.

- 방법 2. ++ 연산시 기존 반복자의 --연산을 수행하는 adapter 를 만들자.

  ⇨ STL 이 선택한 방법.

  ⇨ **iterator adapter**

핵심 정리



- **std::reverse_iterator**
  - ⇨ 기존 반복자의 동작을 거꾸로 수행하는 adapter
  - ⇨ ++ 연산시 기존 반복자의 -- 연산수행
  - ⇨ C++17 이전 환경의 경우 템플릿 인자 전달.

- **이외에도 STL 에는 많은 iterator adapter 가 존재**
  - ⇨ cppreferece.com 참고

proxy

핵심 정리

- **Image**

  ⇨ 그림 파일을 Load 해서 관리하는 클래스

- Image 를 그릴 필요는 없고, 크기 정보만 얻고 싶다

  ⇨ 크기 정보는 그림 파일 전체를 메모리에 로드 할 필요 없고,

  ⇨ 그림 파일의 헤더 부분에서 읽어 올 수 있다.

- 지연된 생성을 위한 대행자

  ⇨ 필요한 경우(draw) 만 객체를 생성하자.

핵심 정리

- ⇨ 지연된 생성
- ⇨ 보안(인증)의 추가
- ⇨ 기능의 추가(로그 기록)
- ⇨ 원격지 서버에 대한 대행자

facade

핵심 정리

- C 언어로 만든 TCP Server 프로그램
  - ⇨ C 언어는 데이타와 함수가 분리 되어 있고, 생성자/소멸자 등의 문법도 없다.
  - ⇨ 코드가 복잡해 보인다.

- 객체지향 프로그래밍
  - ⇨ 네트워크 프로그램에 사용되는 다양한 개념을 각각 클래스로 설계.

핵심 정리

- TCP Server 를 만들려면
  - ⇨ 몇개의 클래스를 사용해서
  - ⇨ 몇가지의 절차를 거쳐야 한다.

- TCPServer 클래스
  - ⇨ TCP Server 를 만드는데 필요한 “일련을 절차에 대한 포괄적 개념의 인터페이스를 제공”하는 클래스.
  - ⇨ 사용하기 쉽게 하기 위한 클래스

bridge

핵심 정리

- **People 이 IPod 을 직접 사용하면**
  - ⇨ 강한 결합
  - ⇨ 다른 제품으로 교체 불가능 하다.

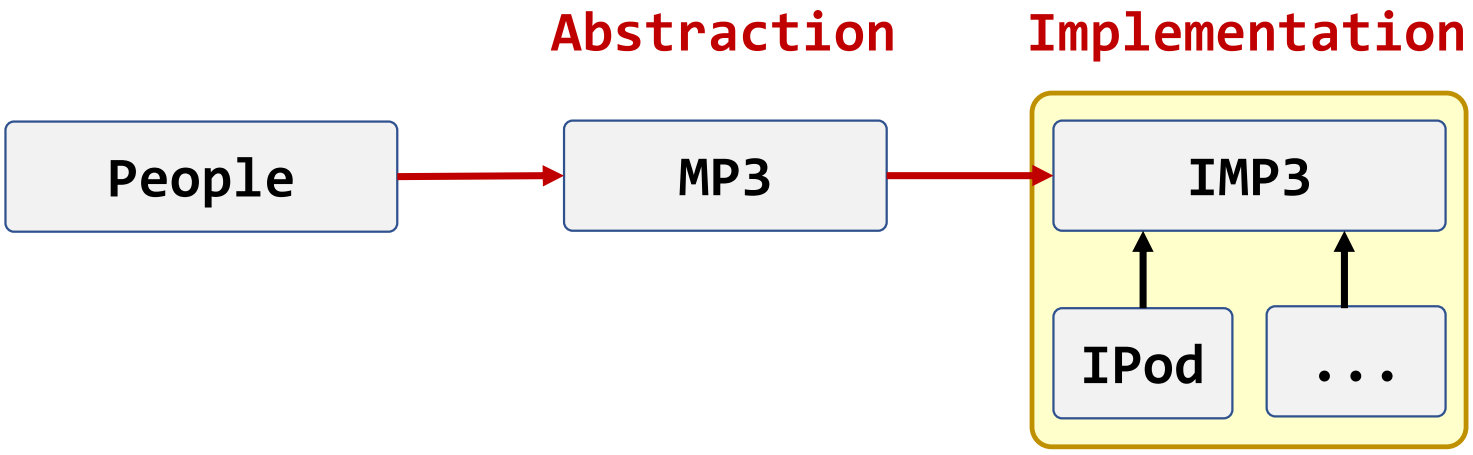
- **People 이 IMP3 인터페이스를 사용하면**
  - ⇨ 약한 결합
  - ⇨ "다른 제품으로 교체 가능"하다.

- **사용자가 새로운 기능을 요구하면**
  - ⇨ IMP3 인터페이스가 수정되어야 한다.
  - ⇨ 인터페이스의 변경은 모든 제품을 변경을 의미.

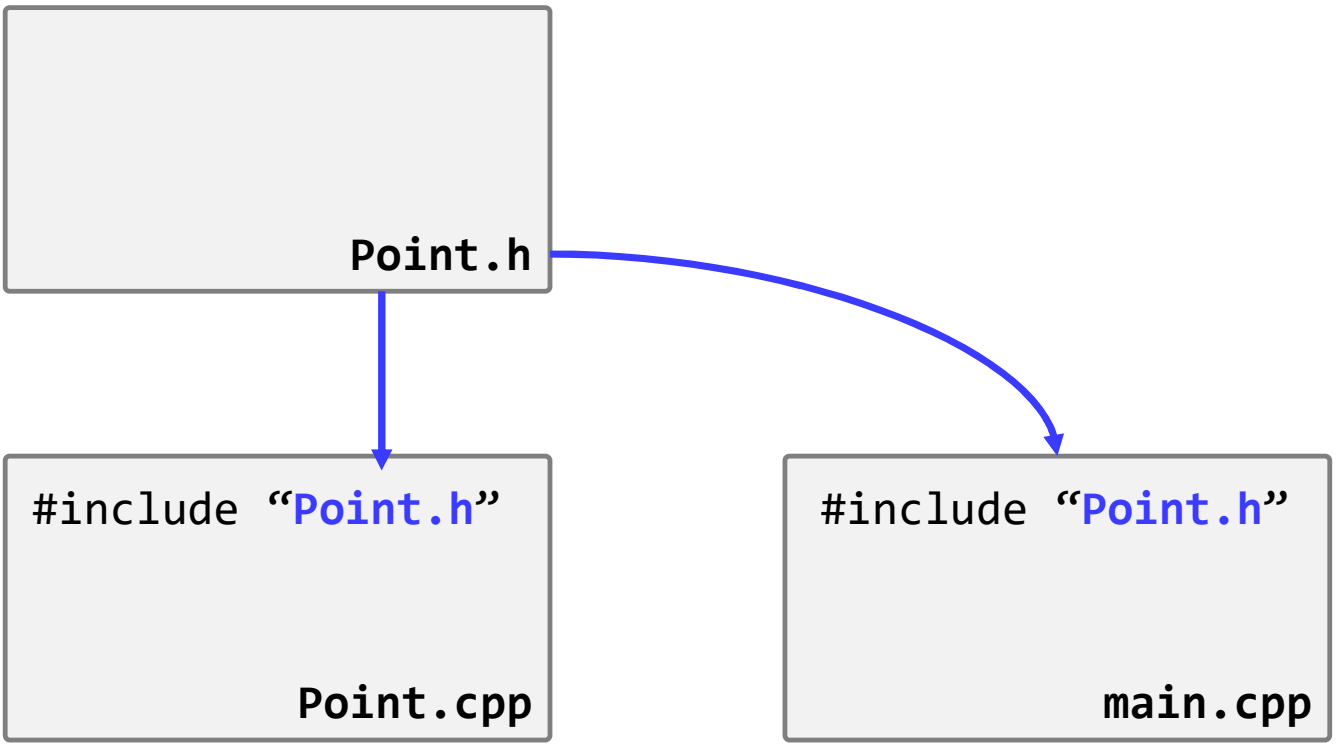
- **IMP3 인터페이스가 변경되면**
  - ⇨ 모든 사용자 코드가 변경되어야 한다.

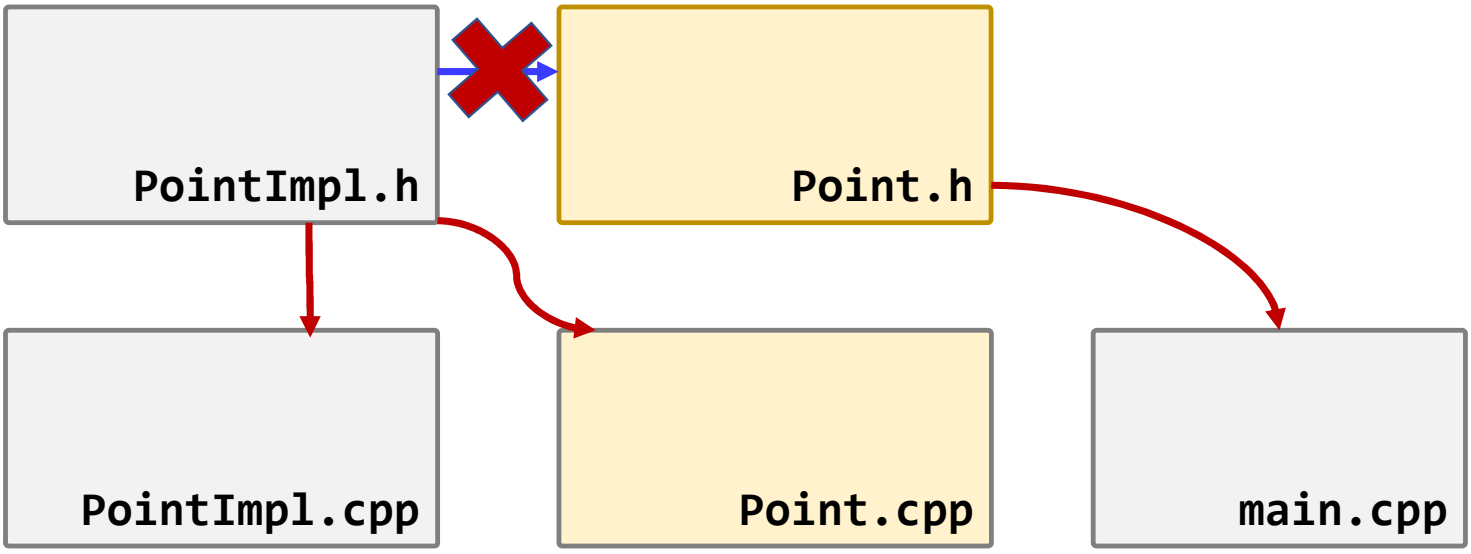
  - ⇨ 사용자와 IMP3 구현의 update 를 상호 독립적으로 만들 수 없을까 ?

핵심 정리



- **PIMPL**
  - ⇨ C++ 진영에서 Bridge 패턴을 나타내는 또 다른 용어
  - ⇨ **P**ointer to **IMPL**ementation

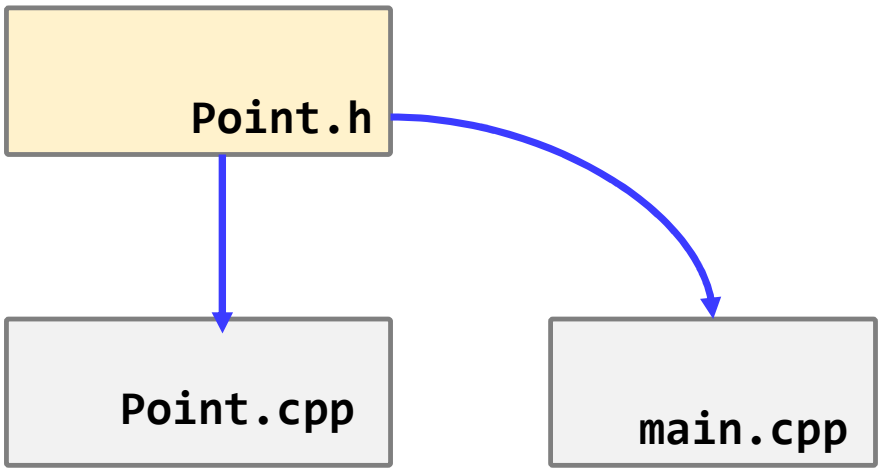

Point.h
#include “Point.h”
Point.cpp
#include “Point.h”
main.cpp

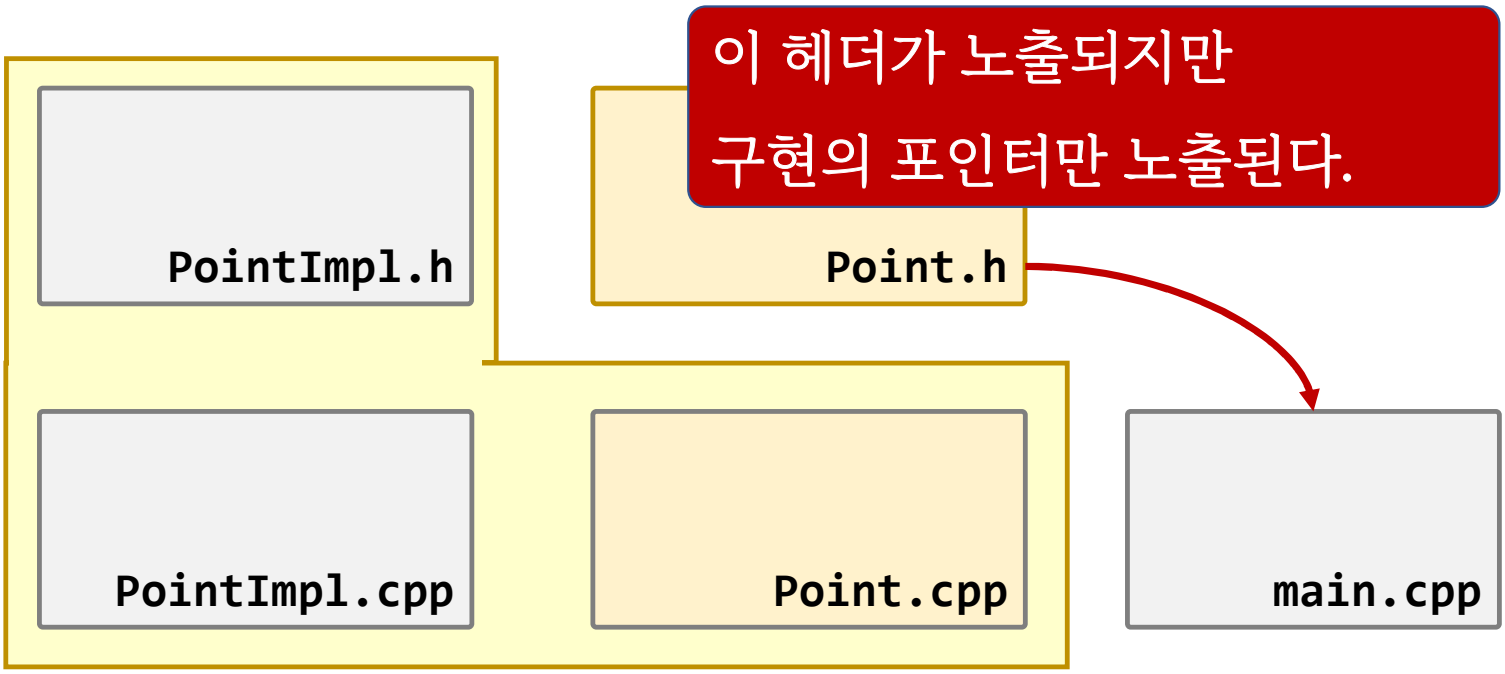

PointImpl.h
Point.h
PointImpl.cpp
Point.cpp
main.cpp

핵심 정리

- **PIMPL 기술의 장점**
  - ⇨ 컴파일 속도가 빨라진다.(컴파일러 방화벽)
  - ⇨ 완벽한 정보 은닉을 할 수 있다.

소스는 빌드해서
배포 할 수 있지만
헤더 파일을
내부가 노출된다.



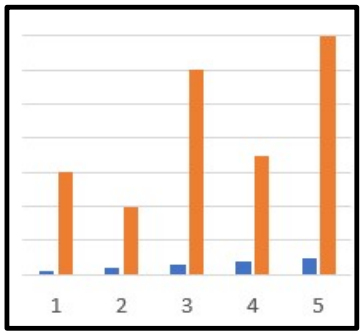


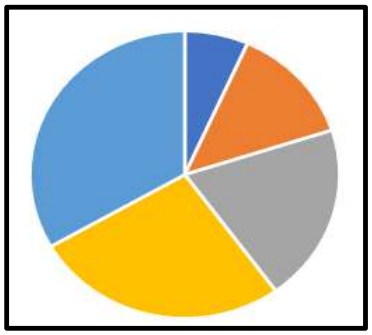
observer

핵심 정리

- 테이블의 데이타가 변경되면 연결된 모든 차트가 수정되어야 한다.

1	2	3	4	5
30	20	60	35	70





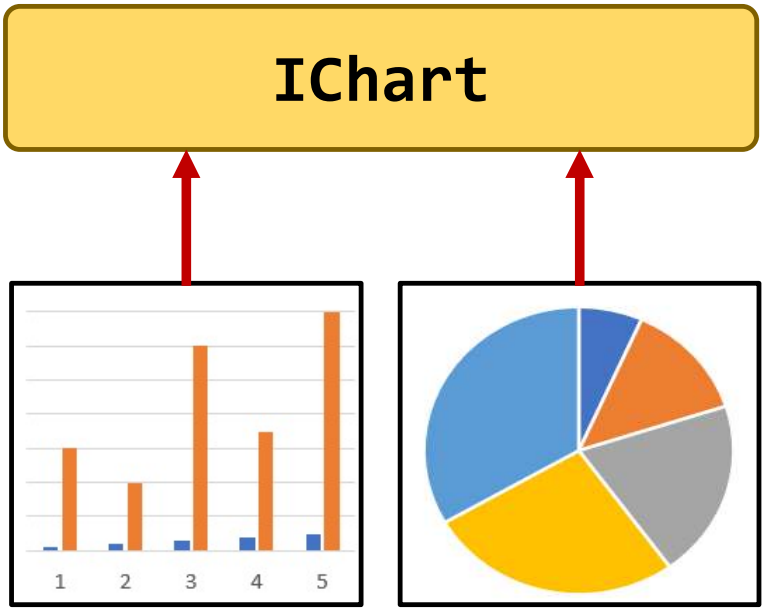
- 방법 1. 차트에서 루프를 돌면서 테이블이 변하는지 관찰한다.
  ⇨ 차트가 여러 개 라면 모든 차트에서 루프를 돌면서 테이블을 관찰해야 한다.

- 방법 2. 차트를 테이블에 등록하고 테이블이 수정되면 등록된 차트에 통보 한다.
  ⇨ **observer** 패턴의 핵심

핵심 정리

1	2	3	4	5
30	20	60	35	70

`std::vector<IChart*>`



① 모든 Chart 의 공통의 인터페이스(IChart) 가 필요하다.

② 테이블에는 다양한 종류의 차트를 보관할 수 있어야 한다. (`std::vector<IChart*>`)

③ 테이블에는 차트를 등록/등록취소 하는 함수가 있어야 한다.

④ 테이블에 있는 데이타에 변화가 생기면 등록된 모든 차트에 통보(약속된 함수 호출)을 해야 한다.

핵심 정리

- 테이블이 관리하는 데이타의 형태가 변경되면
  ⇨ 데이타를 편집하는 방법도 수정되어야 한다.
  ⇨ 하지만, “**observer 패턴의 기본 로직을 제공하는 attach, detach, notify 는 변경되지 않는다.**”
  ⇨ “변경되지 않은 코드(observer 패턴의 기본 로직)을 제공하는 기반 클래스를 제공” 한다.

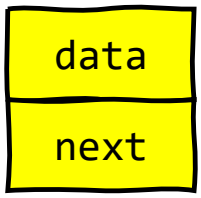
- **Subject**
  ⇨ 관찰의 대상의 기반 클래스
  ⇨ observer 패턴의 기본 로직을 제공.

iterator

CODE NURI

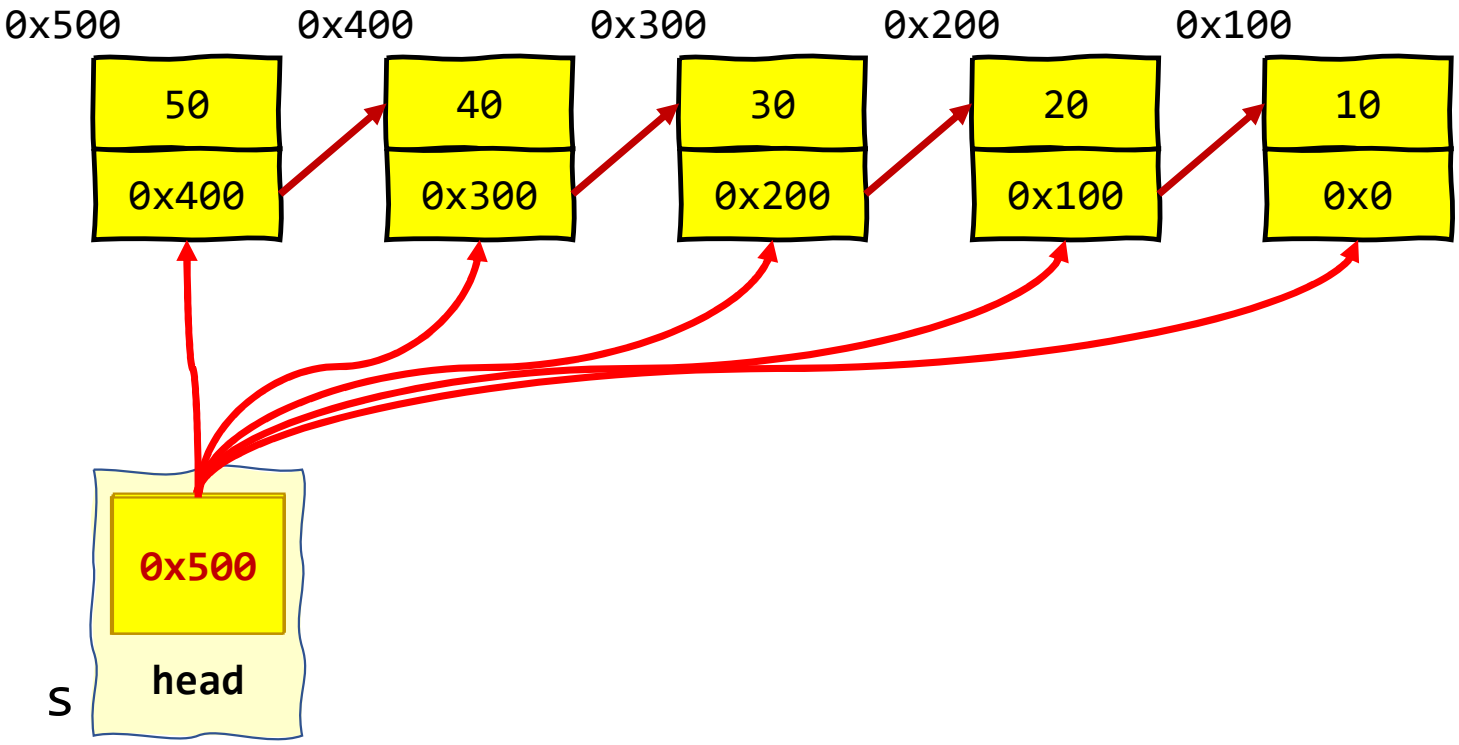
핵심 정리

- **Node<T> 구조체**



- **slist<T>**

  ⇨ Single Linked List



iterator #1

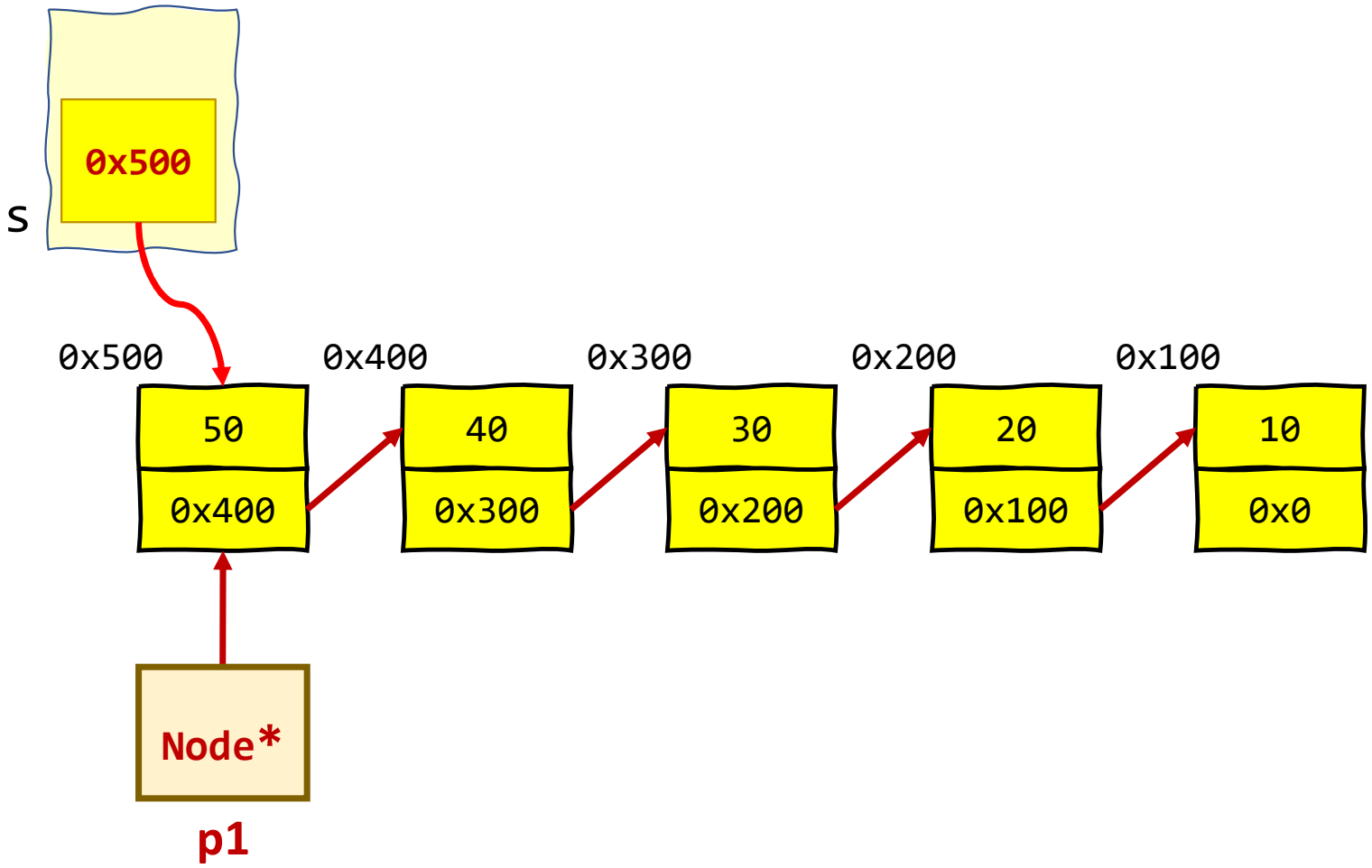
핵심 정리

- **interface 기반의 반복자(iterator)**
  - ⇨ C#, java, python, swift 등의 대부분의 객체지향 언어가 사용하는 방식.
  - ⇨ 강의에서는 "**C#언어**" 와 유사한 이름 사용.
  - ⇨ C# 에서는 반복자를 "**열거자(enumerator)**" 라고도 부름.

- 모든 반복자(iterator)는 사용법이 동일해야 한다.
  - ⇨ 반복자의 인터페이스가 필요 한다.
  - ⇨ **IEnumerator<T>**

- 모든 컨테이너에서는 반복자(iterator)를 꺼낼 수 있어야 한다.
  - ⇨ 반복자를 가진 컨테이너도 인터페이스가 필요.
  - ⇨ **IEnumerable<T>**

핵심 정리



- 반복자(iterator) 구현의 핵심 원리
  - ⇨ 1번째 요소를 가리키는 포인터를 보관하고 있다가
  - ⇨ 약속된 방식으로 다음으로 이동하고, 요소에 접근하는 것.

핵심 정리

- interface 기반의 반복자(iterator) 의 특징(단점)

① s.getEnumerator() 는 동적 할당된 반복자 객체를 반환한다.

⇨ C++ 에서는 반드시 사용 후 delete 해야 한다.

⇨ 라이브러리가 할당하고, 사용자가 delete 하는 것은 좋은 디자인이 아니다.

② moveNext() 와 getObject() 가 가상함수 이다.

⇨ 수천 ~ 수만 개의 요소를 열거한다면 오버헤드가 크다.

③ 모든 컨테이너가 동일한 방법을 사용하지 않는다.

⇨ 배열의 요소를 접근 할 때는 배열의 시작 주소를 보관했다가 ++로 이동하게 된다.

iterator #2

핵심 정리

- STL 방식의 iterator

- 가상함수가 아닌 인라인 치환.

- moveNext() 라는 이름 대신 “**++**” 연산자 재정의
  getObject() 대신 “*****” 연산자 재정의
  ⇨ **raw pointer** 와 동일한 방법으로 사용 가능 하도록
  ⇨ 편의성을 위해 **==, !=** 연산자도 재정의.

- 반복자를 꺼내기 위해
  ⇨ **begin(), end()** 함수 제공.

- 기본 적인 개념을 동일하지만 C++ 만의 특징을 적용

visitor

핵심 정리

- 컨테이너에 담긴 모든 요소를 2배로 하고 싶다면 ?
  - ⇨ for 문으로 모든 요소에 연산(*2)를 수행
  - ⇨ 방문자 패턴을 사용

- 규칙 1. s.accept() 다양한 종류의 방문자 객체를 받을 수 있어야 한다.
  - ⇨ 방문자의 인터페이스가 필요 하다.
  - ⇨ **IVisitor**

- 규칙 2. 방문의 대상(list, vector 등)은 accept() 가 있어야 한다.
  - ⇨ 인터페이스가 필요 하다.
  - ⇨ **IAcceptor**

핵심 정리

- **IVisitor**

  ⇨ 방문자 인터페이스

- 방문자(Visitor)

  ⇨ 요소 한 개에 대한 연산을 정의 하는 객체

  ⇨ IVisitor 인터페이스를 구현해야 한다.

- 방문 대상은 **accept** 가 있어야 한다.

  ⇨ IAcceptor

`e = e * 2`

`1, 2, 3, 4, 5, 6, 7, 8, 9, 10`

TwiceVisitor

요소 한 개에 대한

연산을 정의 하는 객체

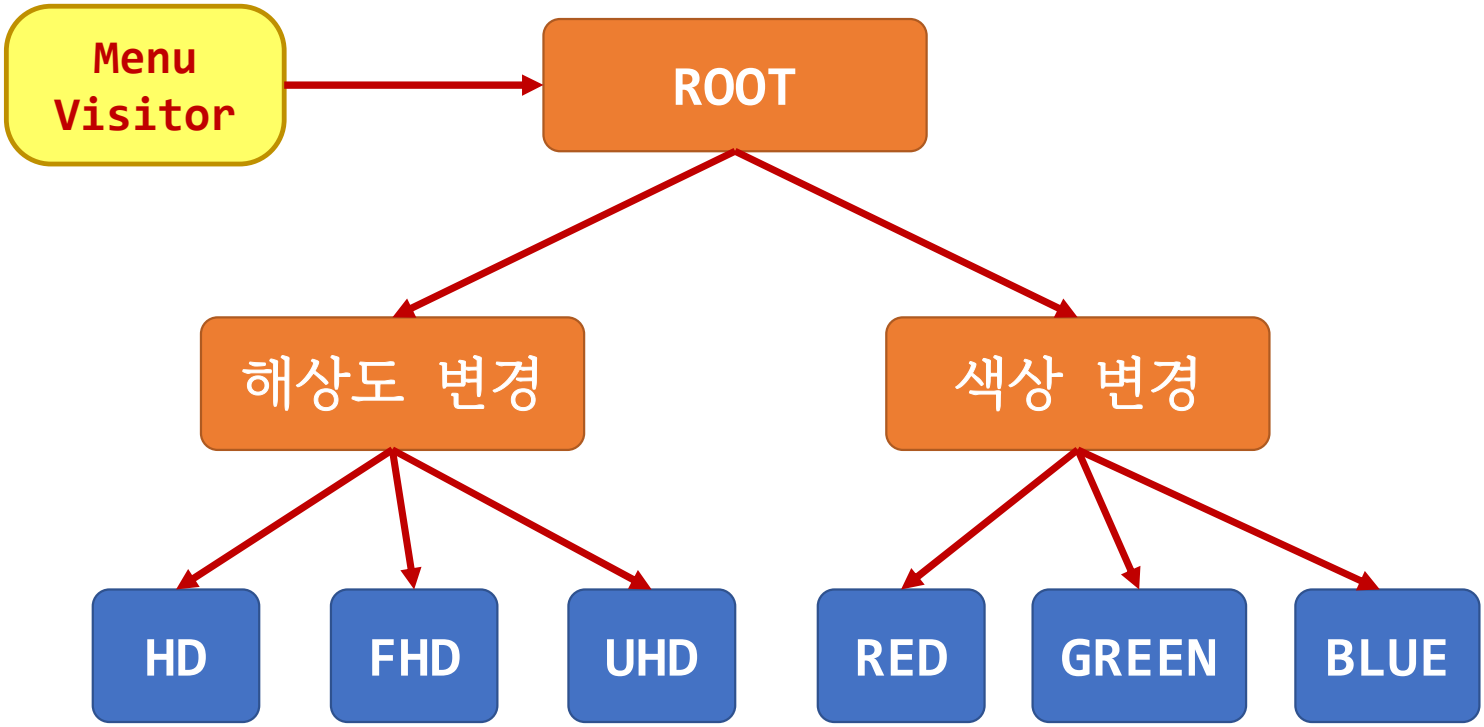
menu visitor

핵심 정리

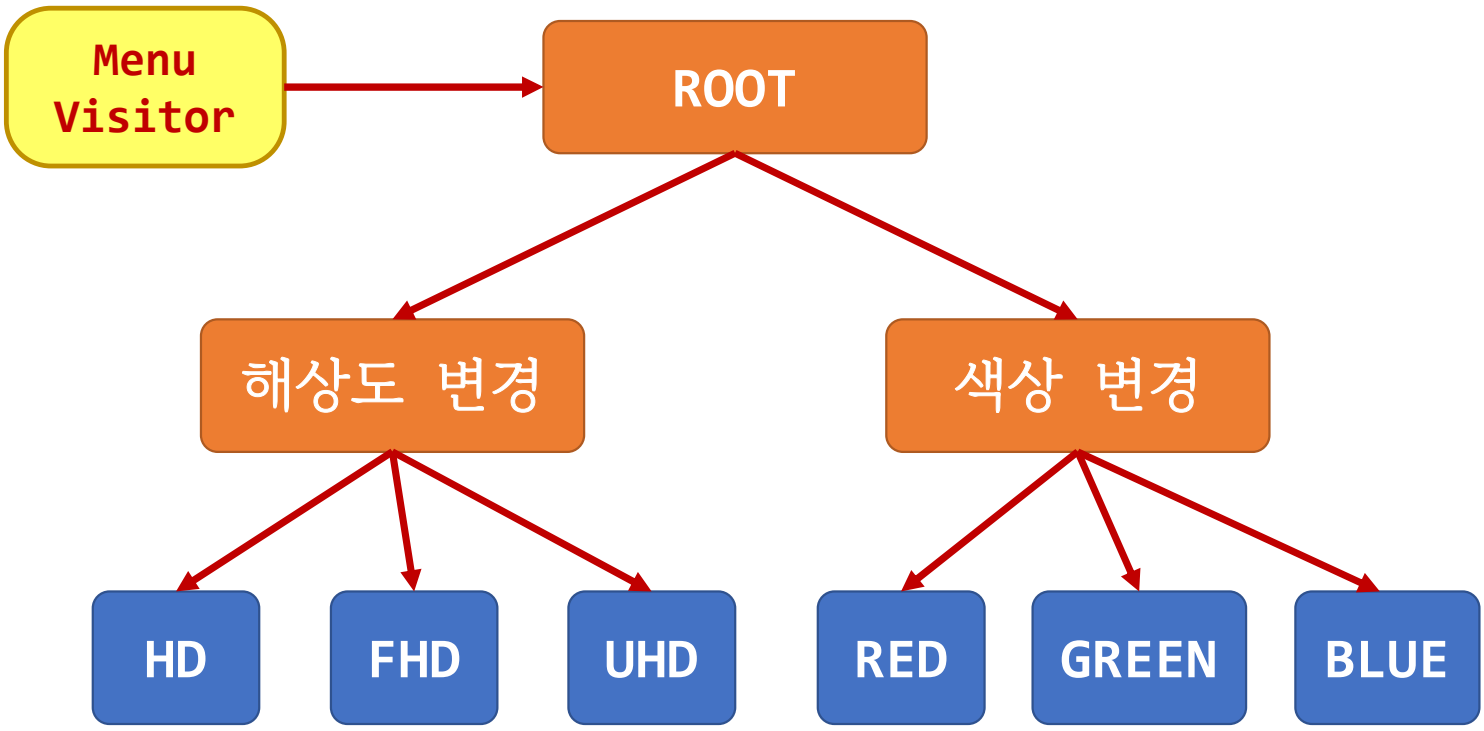
- list
  - ⇨ 모든 요소의 타입이 동일(int) 하다.
  - ⇨ 모든 요소가 선형적인 형태로 보관

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

- Menu
  - ⇨ 요소의 타입이 다양하다(MenuItem, PopupMenu)
  - ⇨ tree 구조로 보관되어 있다.



핵심 정리



singleton

핵심 정리

- 싱글톤 을 구현하는 기본 규칙

- 규칙 1. 외부에서는 객체를 생성할 수 없어야 한다.
  ⇨ **private 생성자**

- 규칙 2. 한 개의 객체는 만들 수 있어야 한다.
  ⇨ **오직 한개의 객체를 만들어서 반환하는 static 멤버 함수**

- 규칙 3. 복사 생성자도 사용할 수 없어야 한다.
  ⇨ **복사/대입 금지 (= delete)**

핵심 정리

- 싱글톤을 구현하는 방법은 여러가지가 있다.

- **Meyer's singleton**
  - ⇨ effective-c++ 의 저자인 scott-Meyer 가 제안한 방법
  - ⇨ 오직 한개의 객체를 "**static 지역변수**" 로 생성

- Meyer's singleton 장점 1. 지연된 초기화
  - ⇨ 처음 get_instance() 를 호출할 때 초기화 된다.
  - ⇨ 사용되지 않으면 생성자가 호출되지 않는다.

- Meyer's singleton 장점 2. **thread-safe**
  - ⇨ 멀티 스레드 환경에서도 Cursor 는 오직 한 개만 생성됨을 보장한다.

singleton #2

핵심 정리

- 유일한 객체를 힙에 만드는 싱글톤
  - ⇨ 멀티 스레드에 안전하지 않다.
  - ⇨ 동기화 필요

- Cursor::get_instance() 를 100번 호출하면
  - ⇨ 최초 호출시 객체를 생성한다.
  - ⇨ 나머지 99번은 생성된 객체를 반환만 한다.
  - ⇨ 그런데, m.lock()/m.unlock() 을 수행하게 된다.

singleton code generator

핵심 정리

- **CRTP**
  - ⇨ Curiously Recurring Template Pattern
  - ⇨ 기반 클래스에서 미래에 만들어질 파생 클래스의 이름을 사용할 수 있게 하는 기술

- **CRTP 원리**
  - ⇨ 기반 클래스를 템플릿으로 만들고
  - ⇨ 파생 클래스에서 자신의 이름을 기반클래스의 템플릿 인자로 전달해 주는 기술.

flyweight

핵심 정리

- **Image 클래스**
  - ⇨ 그림을 그리는 클래스
  - ⇨ 파일 또는 인터넷에서 다운 로드 하는 기능 지원.

- **flyweight 패턴**
  - ⇨ 속성이 동일하면 공유하게 하자.
  - ⇨ 동일한 그림을 관리하는 객체를 2개 만들 필요 없다.

핵심 정리

- **flyweight 패턴 구현**
  - ⇨ static 멤버 함수를 통한 객체의 공유
  - ⇨ factory 클래스를 사용한 객체의 공유

factory

핵심 정리

- 새로운 도형이 추가되어도 “도형을 생성하는 코드가 변경되지 않게” 할 수 없을까 ?
  - ⇨ factory 를 통해서 도형을 생성.

- **factory 를 통한 객체의 생성**
  - ⇨ 객체를 생성하는 장소가 한 곳에 집중된다.
  - ⇨ 객체의 생성 과정을 관리 할 수 있고, 새로운 타입이 추가되어도 한 곳(factory) 만 변경된다.
  - ⇨ 코드 수정을 최소화 할 수 있다.

핵심 정리

- 자신의 객체를 생성하는 **static 멤버 함수**.
  - ⇨ 다양한 기법으로 활용 될 수 있다.
  - ⇨ Rect 객체를 만드는 2가지 방법

```
Rect* r1 = new Rect;

Rect* r2 = Rect::create();
```

- **C++ 에서는 클래스 이름을 자료 구조에 보관할 수 없다.**
  - ⇨ v.push_back("**Rect**") 는 문자열을 보관한 것이지 클래스 정보를 보관한 것은 아님.
  - ⇨ "Rect" 라는 문자열을 가지고 Rect 타입의 객체를 만들 수 없다.

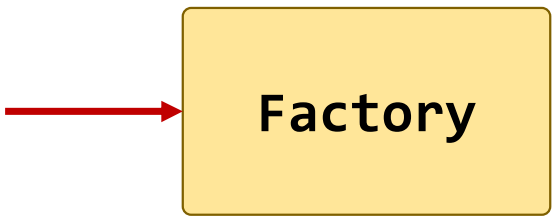
- 하지만 **Rect** 객체를 만드는 생성함수를 자료 구조에 보관 할 수 있다

prototype

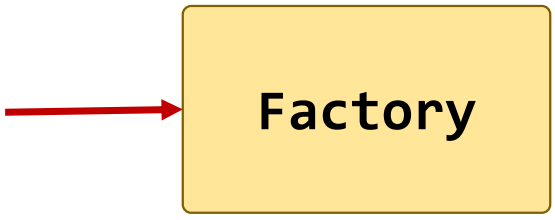
핵심 정리

- **prototype.cpp**
  - ⇨ factory2.cpp 복사해서 사용
  - ⇨ **factory.register_shape(1, &Rect::create);**

Rect, Circle 등의 클래스를 등록
(정확히는 "생성 함수를 등록")



클래스가 아닌 자주
사용하는 "객체를 등록"



abstract factory

핵심 정리

- **GUI 라이브러리안에 2가지 종류의 컨트롤 클래스 제공**

RichButton **Rich**Edit **Rich**...	이쁘고 화려하지만 메모리 사용량이 많고, 성능이 좋지 않음. 고성능의 시스템에 적합
SimpleButton **Simple**Edit **Simple**...	단순하고, 이쁘지 않지만 메모리 사용량이 적고 빠르게 동작

- 사용자가 옵션으로 종류를 선택(변경)가능.

  ⇨ 각 컨트롤 마다 인터페이스가 필요 한다.

memento

핵심 정리

- Graphic 클래스

  ⇨ 윈도우에 그림을 그릴 때 사용하는 많은 함수들을 제공

- 선을 그릴 때 선의 색상이나 두께 등을 변경하고 싶다.

  ⇨ 방법 1. 함수인자로 전달

  ⇨ 방법 2. Graphic 객체의 속성으로 지정

핵심 정리

- memento 패턴
  - ⇨ 객체의 내부상태를 저장했다가 복구 할 수 있게 한다.
  - ⇨ 객체가 스스로를 저장했다가 복구 하므로 캡슐화를 위배 하지 않는다.