

모던 C++

초기화 / Value categories

초기화

```
int x; // x = ?  
std::cout << x;  
x = 1;
```

```
float y; // y = ?  
std::cout << y;
```

Trap representation 이 존재 할 수 있음

표준에서는 **unsigned char** 타입은
trap representation이 없다고 정의

```
std::cout << std::sqrt(-1) << '\n';  
std::cout << std::numeric_limits<double>::signaling_NaN();
```

```
if (std::numeric_limits<double>::has_signaling_NaN) {  
    std::cout << "NaN\n";  
} else {  
    std::cout << std::sqrt(-1) << '\n';  
}
```

```
int extract_int(float f) {  
    union { int i; float f; } u;  
    u.f = f;  
    return u.i;  
}
```

번외 – undefined / unspecified / implementation-defined behavior

Undefined Behavior

무슨 일이 일어날지 전혀 알 수 없음

- 프로그램 종료
 - 컴퓨터 종료
 - 이메일 발송
-
- Null 포인터 참조(dereference)
 - **end()** 참조
 - **const** 객체의 값 변경

Unspecified / implementation-defined behavior

- **std::vector** 의 메모리 재할당 정책
- 람다 함수의 반환 타입

초기화

```
int y = 0;  
int arr[10] = {}; // arr = {0, 0, ..., 0}
```

```
struct MyClass {  
    MyClass() { /* ... */ }  
    MyClass(int a, int b) { /* ... */ }  
};
```

```
MyClass a;
```

```
MyClass b{y, z};  
MyClass c(y, z);  
MyClass d = {y, z};
```

초기화 – implicit / explicit

```
struct ClassA {  
    ClassA(int x) {}  
};  
  
struct ClassB {  
    explicit ClassB(int x) {}  
};
```

```
ClassA a = 10;  
ClassB b = 20; // 컴파일 에러!  
ClassB b2(20);
```

```
void MyFunction(ClassA) {}
```

```
struct ClassC {  
    ClassC(ClassA) {}  
};
```

```
MyFunction(20);  
ClassC c = 10; // 컴파일 에러!
```

Implicit conversion

암시적 형 변환 (implicit type conversion)
T -> U

- 함수 인자
- 연산자의 인자
- 함수 반환시
- **switch** 구문
- **if** 문

```
std::sqrt(1);  
std::sqrt(1.0);
```

```
1 + 1  
1 + 1.0
```

```
double relu(double x) {  
    if (x < 0)  
        return 0;  
    return x;  
}
```

초기화

```
template<typename T>
T Create() {
    return /* ? */
}
```

```
auto x = Create<int>(); // x = 0
auto y = Create<MyClass>(); // MyClass::MyClass()
```

```
template<typename T>
T Create() {
    if constexpr (std::is_class<T>()) {
        return T();
    } else {
        return 0;
    }
}
```

Uniform Initialization

```
int z = 0;
```

```
MyClass b;
```

```
int z{}; // z = 0  
int arr[10]{}; // arr = {0, 0, ... 0}
```

```
MyClass b{};
```

기본 타입: 0으로 초기화
클래스: 기본 생성자 호출

```
int z();
```

```
MyClass b();
```

잘못된 선언! 함수로 인식함

```
int my_function();
```

```
int my_function() {
```

```
}
```

선언(declare) 과 구현(define) 이 분리된 함수의
선언과 동일한 모양새

Uniform Initialization

```
struct MyClass {  
    MyClass() = default;  
  
    int my_function();  
  
    std::vector<int> my_vector_a(); // 함수로 인식 함  
    std::vector<int> my_vector_b(10); // 컴파일 에러!  
};  
  
int MyClass::my_function() {}
```

```
struct MyClass {  
    MyClass() = default;  
  
    int my_function();  
  
    std::vector<int> my_vector_a{};  
    std::vector<int> my_vector_b{10};  
};  
  
int MyClass::my_function() {}  
  
int main() {  
    MyClass a;  
  
    std::cout << a.my_vector_a.size() << std::endl;  
    std::cout << a.my_vector_b.size() << std::endl;  
  
    return 0;  
}
```

Uniform Initialization – 주의 사항: Initializer list

{1, 2, 3, 4}

타입이 없는 중괄호 배열은 `std::initializer_list` 객체

C++ language feature 이기 때문에 이 동작은 바꿀 수 없음

Member functions

(constructor) creates an empty initializer list
(public member function)

Capacity

size returns the number of elements in the initializer list
(public member function)

Iterators

begin returns a pointer to the first element
(public member function)

end returns a pointer to one past the last element
(public member function)

Container



```
for (auto x : { 1, 2, 3, 4 }) {  
    std::cout << x << ' ' ;  
}
```

```
// C++ 20  
for (int x : std::views::iota(1, 5)) {  
    std::cout << x << ' ' ;  
}
```

Uniform Initialization – 주의 사항:_INITIALIZER list

```
namespace std {  
  
template< class T >  
class initializer_list;  
  
} // namespace std
```

템플릿 클래스 -> T 가 추론 가능해야 함

```
auto a = {1, 2, 3}; // std::initializer_list<int>  
auto b = {1.0, 2.0, 3.0}; // std::initializer_list<double>  
  
auto c = {}; // 아무것도 아님!  
auto d = {1, 2.0} // T 가 일정하지 않음!
```

```
std::vector<int> v1 = {1, 2, 3};
```

```
// 컴파일 안 됨! (narrowing conversion)  
std::vector<int> v2 = {1, 2.0, 3};
```

```
// 컴파일 가능 (T = double 로 명시되어 있음)  
std::vector<double> v2 = {1, 2.0, 3};
```

```
// C++17
```

```
// OK. T = int  
std::vector v1 = {1, 2, 3};
```

```
// 컴파일 안 됨!  
std::vector v2 = {1, 2.0, 3};
```

Uniform Initialization – 주의 사항:_INITIALIZER list

중괄호 사용 시 `std::initializer_list` 를 인자로 받는 생성자가 최고 우선순위로 호출됨

```
std::vector<int> a(1, 10); // a = {1, 1, 1, ... 1}  
std::vector<int> b{1, 10}; // b = {1, 10}
```

좋은 작성법: 명시적인 기본 생성만 중괄호 사용

Uniform Initialization – 주의 사항:_INITIALIZER list

```
struct MyClass {  
    MyClass() {  
        std::cout << "MyClass::MyClass()" << '\n';  
    }  
  
    MyClass(int) {  
        std::cout << "MyClass::MyClass(int)" << '\n';  
    }  
  
    MyClass(int, int) {  
        std::cout << "MyClass::MyClass(int, int)" << '\n';  
    }  
  
    MyClass(std::initializer_list<int>) {  
        std::cout << "MyClass::MyClass(std::initializer_list)" << '\n';  
    }  
};
```

```
MyClass a = 1;  
MyClass b = {1, 2};  
MyClass c = {1, 2, 3};  
  
MyClass d(1, 2);  
MyClass e(1, 2, 3); // 컴파일 에러!  
  
MyClass f{};  
MyClass g = {};
```

특수 멤버 함수 (Special Member Functions)

```
struct MyClass {  
    MyClass();  
    MyClass(int x);  
  
    MyClass(const MyClass& other);  
    MyClass(MyClass&& other) noexcept;  
  
    MyClass& operator=(const MyClass& other);  
    MyClass& operator=(MyClass&& other) noexcept;  
  
    ~MyClass();  
};
```

- 기본 생성자 (default constructor)
- 복사 생성자 (copy constructor)
- 복사 대입 연산자 (copy assignment operator)
- 이동 생성자 (move constructor)
- 이동 대입 연산자 (move assignment operator)
- 소멸자 (destructor)

`MyClass(int x);` 는 특수 멤버 함수가 아님!

복사 생성자 (copy constructor)

```
std::string a = "Hello";
```

```
int a = 1;
```

```
std::string b(a);  
std::string c = a;
```

```
int b(a);  
int c = a;
```

원본과 동일한 복사본을 가짐
생성과 동시에 초기화

`std::string b(a);` ➡ 명시적 형 변환 (explicit conversion)

`std::string c = a;` ➡ 암시적 형 변환 (implicit conversion)

복사 생성자 (copy constructor)

```
struct MyClass {  
    MyClass(const MyClass& other)  
        : data(other.data) {}  
  
    int data;  
};
```


복사 대입 연산자 (copy assignment operator)

```
std::string a = "Hello";  
std::string b = "World";
```

```
b = a;
```

원본과 동일한 복사본을 가짐
현재 값을 지우고 복사본을 저장

```
int a = 1;  
int b = 2;
```

```
a = b;
```

```
std::string b = "World";  
b = a;
```

➡ = 이 있지만 생성자가 호출됨
➡ 복사 대입 연산자 호출

복사 대입 연산자 (copy assignment operator)

```
struct MyClass {  
    MyClass& operator=(const MyClass& other) {  
        data = other.data;  
        return *this;  
    }  
    int data;  
};
```

이동 생성자 (move constructor)

```
std::string a = "Hello";
```

```
std::string b(std::move(a)); // b = "Hello ", a = ""  
std::string c = std::move(a); // c = "", a = ""
```

```
int a = 1;
```

```
int b(std::move(a)); // b = 1, a = 1  
int c = std::move(a); // c = 1, a = 1
```

원본을 복사하지 않고 그대로 가지고 옴. 대신 원본은 비어 있게 됨

이동 생성이 불가능한 객체들은 복사 생성자가 호출됨

이동 생성자 (move constructor)

```
struct MyClass {  
    MyClass(MyClass&& other) noexcept  
        : data(std::move(other.data)) {}  
  
    std::string data;  
};
```

이동 대입 연산자 (move assignment operator)

```
std::string a = "Hello";  
std::string b = "World";
```

```
b = std::move(a); // b = "Hello ", a = ""
```

```
int a = 1;  
int b = 2;
```

```
b = std::move(a); // b = 1, a = 1
```

현재 값을 지우고 추가적인 비용 없이 원본을 저장

이동 대입이 불가능한 객체들은 복사 대입 연산자가 호출됨

```
std::string b = "World";
```

```
b = a;
```



= 이 있지만 생성자가 호출됨



복사 대입 연산자 호출

이동 대입 연산자 (move assignment operator)

```
struct MyClass {  
    MyClass& operator=(MyClass&& other) noexcept {  
        if (this != &other) {  
            data = std::move(other.data);  
        }  
        return *this;  
    }  
  
    std::string data;  
};
```

명시적/암시적 - 생성/삭제

사용자가 직접 delete 를 사용한 경우 명시적으로 삭제되었다고 함(explicitly deleted)

사용자가 직접 delete 를 사용한 적이 없는데도, 삭제되는 경우가 있는데 이를 암시적으로 삭제되었다고 함(implicitly deleted)

사용자가 직접 default 를 사용한 경우 명시적으로 defaulted 되었다고 함(explicitly defaulted)

기본 생성 된 경우 암시적으로 정의되었다고 함(implicitly defined)

```
struct A {  
    A() = default; // explicitly defaulted  
};
```

```
struct B {  
    B() = delete; // explicitly deleted  
};
```

```
struct C {  
    // C() is implicitly defined  
    int x;  
};
```

특수 멤버 함수가 기본 생성 되지 않는 경우

멤버 변수들 중 하나 이상이 복사/이동이 불가능한 경우

-> 현재 클래스도 복사/이동 생성자 및 이동 연산자가 implicitly define 되지 않음

(기본 생성자를 제외한) 생성자를 하나 이상 작성 하거나 명시적으로 defaulted 한 경우

-> 기본 생성자가 implicitly define 되지 않음

복사 생성자 혹은 복사 대입 연산자를 직접 작성한 경우

-> 이동 생성자 / 이동 대입 연산자는 implicitly define 되지 않음

이동 생성자 혹은 이동 대입 연산자를 직접 작성한 경우

-> 복사 생성자 / 복사 대입 연산자는 implicitly define 되지 않음

Rule of five

기본 생성자를 제외한 나머지 5개의 특수 멤버 함수들 중 하나 이상을 직접 구현하였다면, (논리적으로) 나머지 특수 멤버 함수들도 직접 구현 해야 할 확률이 높습니다

<예제 9>

동적 배열 `MyVector` 를 구현해 봅시다. (2)

`MyVector` 다음 기능을 추가로 가져야 합니다.

- 복사 생성자 `copy constructor`
- 복사 대입 연산자 `copy assignment operator`
- 이동 생성자 `move constructor`
 - 이동은 $O(1)$ 시간에 완료되어야 하고 불필요한 자원을 사용하지 않습니다
- 이동 대입 연산자 `move assignment operator`
 - 이동은 $O(1)$ 시간에 완료되어야 하고 불필요한 자원을 사용하지 않습니다

```
template<typename T>
class MyVector {
public:
    // Existing codes...

    MyVector(const MyVector&) { /* ... */ }
    MyVector(MyVector&&) { /* ... */ }

    MyVector& operator=(const MyVector&) { /* ... */ }
    MyVector& operator=(MyVector&&) { /* ... */ }

};
```

Trivially copyable type

- **void**를 제외한 기본 자료형(**char, int, float, ...**) 및 포인터형
- 이러한 자료형들로만 이루어진 클래스
 - 단, 가상함수가 없고 부모 클래스도 가상 함수가 없어야 함
 - 특수 멤버 함수를 직접 정의해선 안 됨



POD (Plain Old Data) type

이러한 자료형들은 레퍼런스를 사용하지 않고 객체를 복사해서 넘겨주는게 일반적

```
int add(int a, int b) {  
    return a + b;  
}
```

// 문제는 없으나 굳이 이렇게 사용 할 필요가 없음

```
int add(const int& a, const int& b) {  
    return a + b;  
}
```

Trivially copyable type – 예시

```
std::vector<int> v = {1, 2, 3, 4, 5};  
  
for (int x : v) {  
    std::cout << x << ' ' ;  
}
```

레퍼런스는 결국 포인터이다
-> 64비트 시스템에서는 포인터의 크기는 64비트

int 는 일반적으로 32비트
-> fixed width integer types 사용
int32_t, int_fast8_t 등 (헤더: <stdint>)

```
// std::string 원본이 어딘가에 존재해야 함  
void read_file(const std::string& path) {}
```

```
// std::string 객체 생성 시 문자열 복사가 이루어짐  
read_file("/Users/data.txt");
```



```
// C++ 17  
void read_file(std::string_view path) {}
```

```
// 문자열 복사가 이루어지지 않음  
read_file("/Users/data.txt");
```

Trivially copyable type – 예시

```
// C++ 20
// 연속된 메모리 구조를 갖는 컨테이너의 뷰
int get(std::span<int> array_view, std::size_t index) {
    return array_view[index];
}
```

```
int arr[] = {1, 2, 3, 4, 5};
get(arr, 2);
```

```
std::vector<int> v = {1, 2, 3, 4, 5};
get(v, 2);
```

```
// 3개만 볼 수 있다고 전달
get({v.data(), 3}, 2);
```

➡ iterator 를 사용하는 boilerplate code 를 줄일 수 있음

Trivially movable type

- Trivially copyable 한 타입

결국 진정한 의미에서의 "이동"은 존재하지 않습니다. 모든 값은 복사를 통해 이루어집니다.

실습 - Git

형상 관리 도구



Lorem ipsum dolor
sit amet,
Consectetur
adipiscing elit, sed
do eiusmod
tempor incididunt
ut labore et dolore
magna aliqua.

Version 1

Lorem ipsum dolor
sit amet,
[REDACTED]
do eiusmod
tempor incididunt
ut labore et dolore
magna aliqua.
Ut enim ad minim
veniam, quis

Version 2

...

Hello, world!

Version 123

실습 - Git



실습 – Git – commit

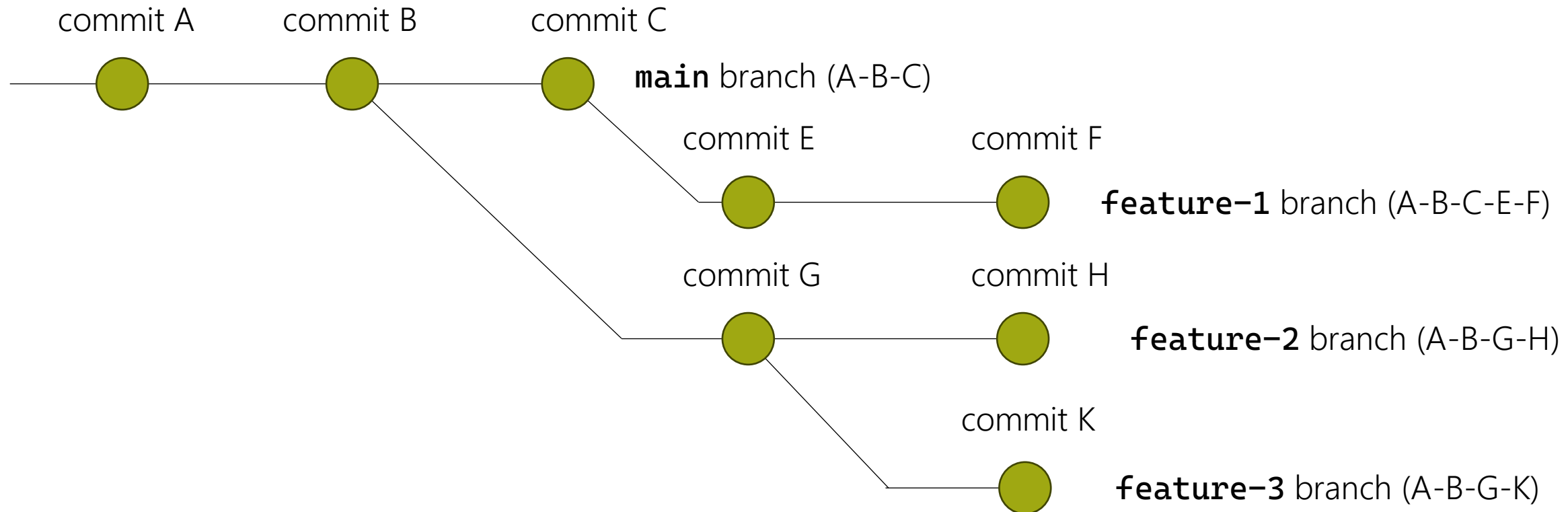
- 각 commit 은 이전 commit 에 대해 1개 이상의 변경사항을 담고 있습니다
- 각 commit 은 고유한 해시값을 가지고 있습니다



실습 – Git – branch

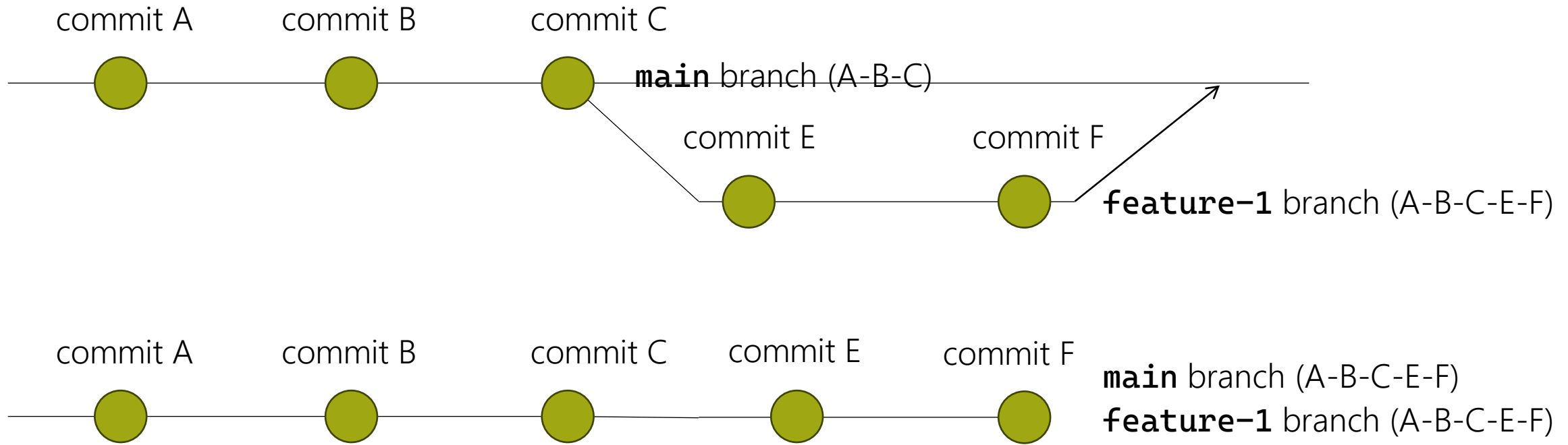
- branch 는 분기된 commit 흐름의 집합입니다

> git checkout [-b] <branch_name>

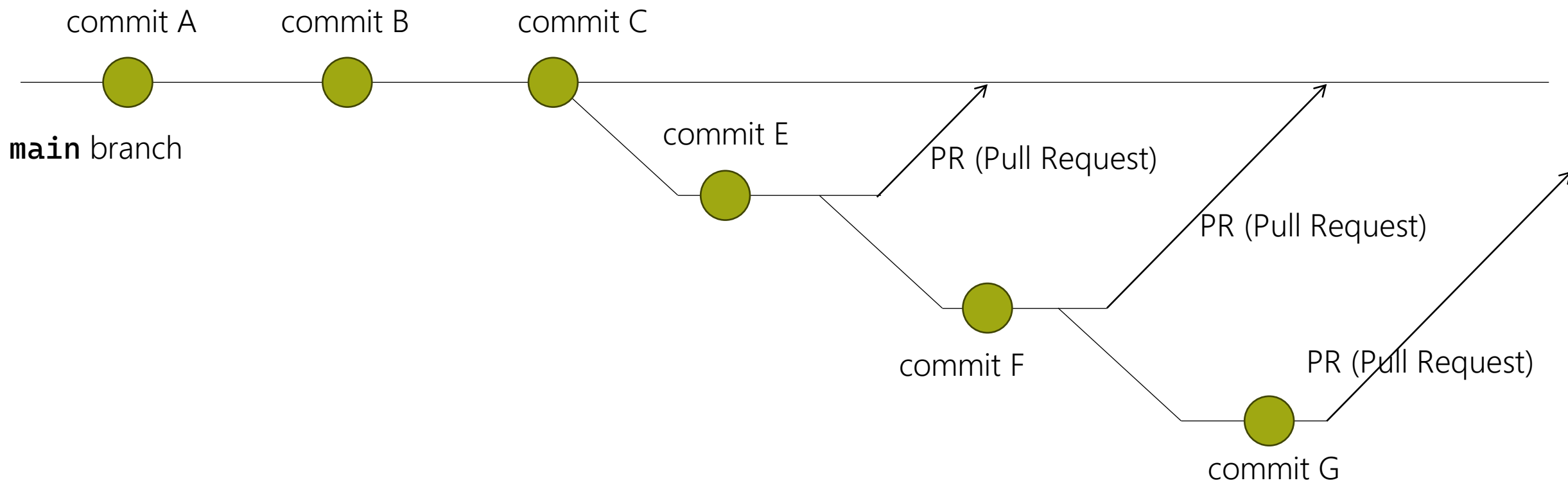


실습 – Git – merge

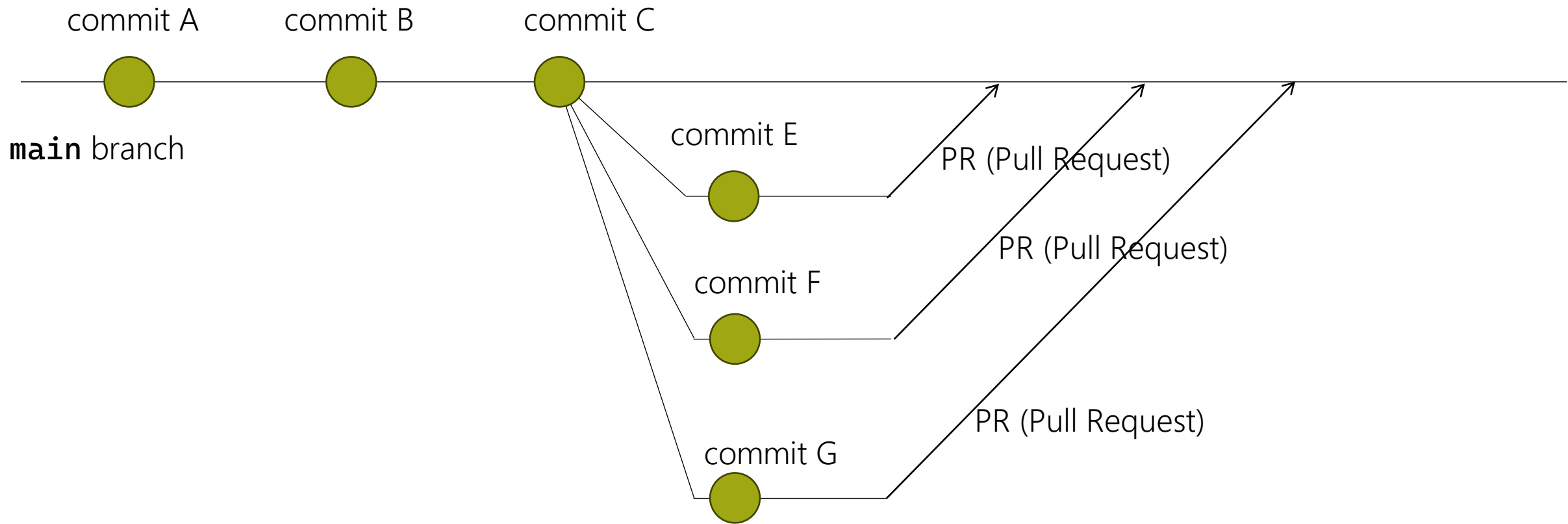
- 분기된 branch 들을 합칠 수 있습니다



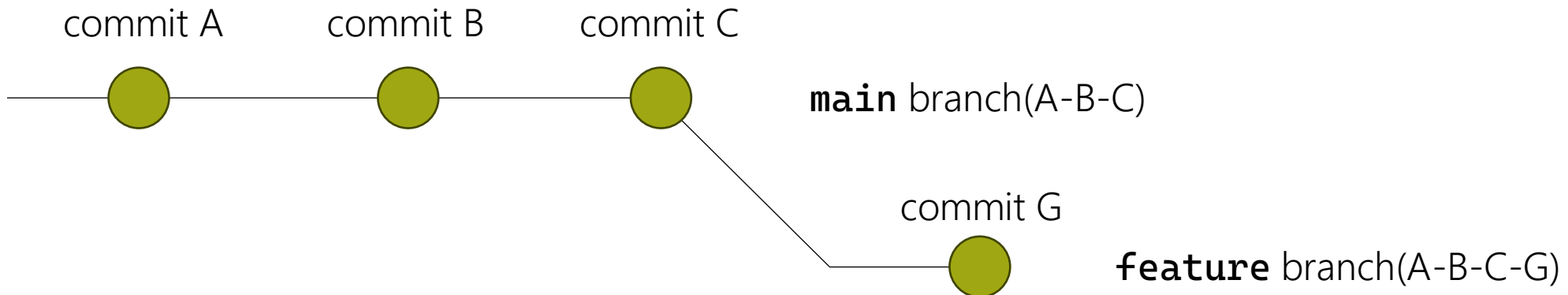
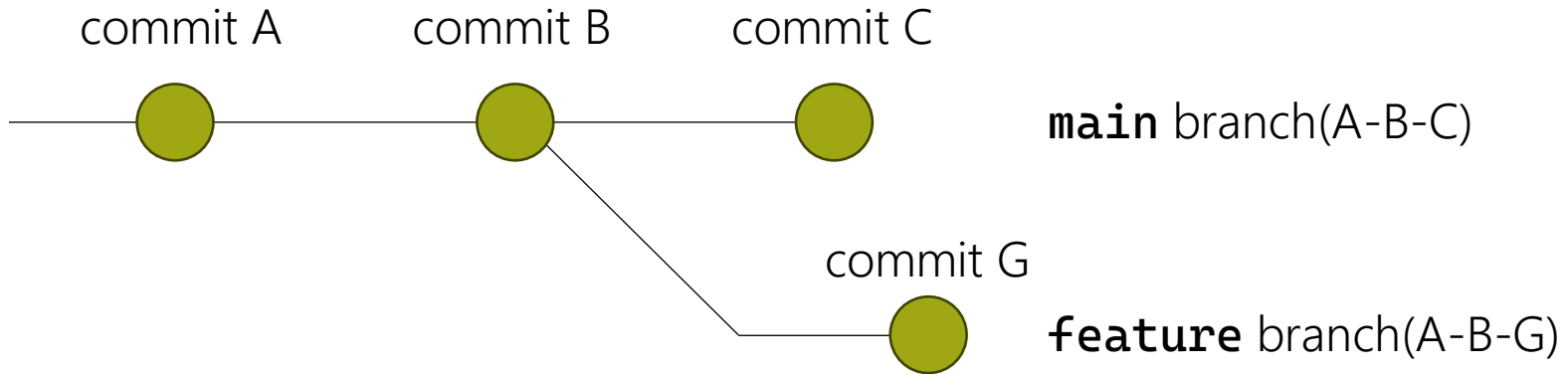
실습 – Git – merge



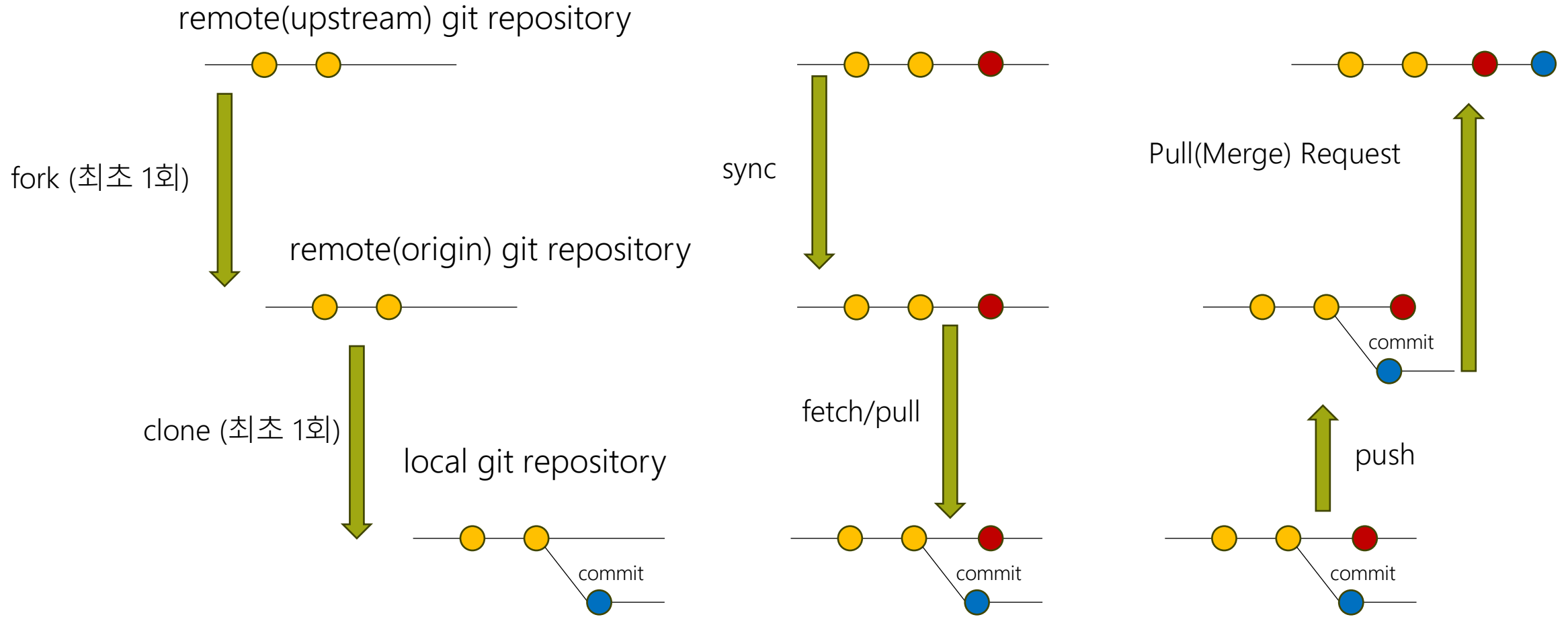
실습 – Git – merge



실습 – Git – rebase



실습 – Git – remote



실습 - 8

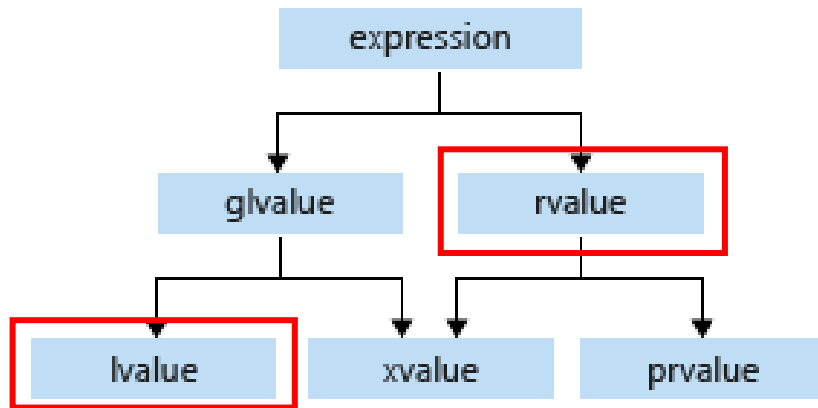
POD 클래스 `MyApplication` 을 정의하고,
모든 특수 멤버 함수 호출 시에 현재 함수의 signature 를 출력하도록 합니다.

```
class MyApplication {  
    /*  
        Implementation  
    */  
};  
  
int main() {  
    // print: MyApplication::MyApplication()  
    MyApplication a;  
  
    // print: MyApplication::MyApplication(const MyApplication&)  
    MyApplication b = a;  
  
    // print: MyApplication::~~MyApplication()    (b destroyed)  
    // print: MyApplication::~~MyApplication()    (a destroyed)  
    return 0;  
}
```

MSVC 에서는 `__FUNCSIG__` 매크로를,
gcc 및 clang 에서는 `__PRETTY_FUNCTION__`
매크로를 사용하면 함수의 signature 를 쉽게
얻을 수 있습니다.

표준은 `__func__` 매크로 입니다

Value Categories



출처: <https://learn.microsoft.com/>

Lvalue

대입 시 왼쪽에 올 수 있는 값

```
int x = 1;  
int y = 2;
```

```
x = y;  
y = x;
```

```
x = y + 1;  
y + 1 = x; // 컴파일 안 됨
```

```
x = 1 + 2;  
1 + 2 = x; // 컴파일 안 됨
```

Rvalue

대입 시 왼쪽에 올 수 ~~없는~~ 값

- 중간 값
- 임시 값

```
int x = 1 + 2;
```

```
int my_function() {  
    return 1;  
}
```

```
my_function() = 4; // 컴파일 안 됨: rvalue
```

Rvalue Reference

임시 값들을 나타내는 타입

scope 를 벗어나면 임시 객체는 파괴됨

```
struct MyClass {  
    MyClass(std::string&& str)  
        : data(std::move(str)) {}  
  
    std::string data;  
};
```

other 자체로는 lvalue

Rvalue Reference

```
struct MyClass {  
    MyClass(std::string&& other);  
  
    MyClass& operator=(std::string&& other);  
};
```

```
MyClass a = std::string("Hello, world!");
```

```
std::string b = "Hi, world!";  
MyClass c = std::move(b);
```

Perfect Forwarding

lvalue 와 rvalue 모두에 대해서 동작하는 매커니즘을 함수 1개만으로 정의할 수 있을까?

```
struct MyClass {  
    MyClass(std::string& str) : data(str) {}  
  
    MyClass(std::string&& str) : data(std::move(str)) {}  
  
    std::string data;  
};
```

lvalue → 복사 생성자 호출

rvalue → 이동 생성자 호출

Perfect Forwarding

```
struct MyClass {  
    template<typename T>  
    MyClass(T&& str) : data(std::forward<T>(str)) {}  
  
    std::string data;  
};
```

std::string&  T = std::string&


std::string&&  T = std::string

Variadic Template

0개 이상의 인자를 받는 템플릿 파라미터

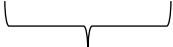
```
template<typename... T>
void print(const T&... arg) {
}
```

```
std::printf(const char* fmt, ...);
```



C의 가변 인자(variadic argument)와 다른 개념! 가변 인자는 타입에 대한 정보가 손실됨

```
std::printf("%d, %s", 3, "hello");
```



뒤에 나오는 int와 const char*의 타입에 대한 정보가 없기 때문에 포맷을 프로그래머가 직접 작성해야 함

Variadic template

```
template<typename T>
void print(const T& arg) {
    std::cout << arg;
}
```

```
template<typename T, typename... Ts>
void print(const T& arg, const Ts&... args) {
    print(arg);
    print(args...);
}
```

```
// C++ 17 (fold expression)
template<typename... Ts>
void print(const Ts&... args) {
    (std::cout << ... << args);
}
```


Perfect forwarding variadic template

```
template<typename... T>
std::string make_string(T&&... args) {
    return std::string(std::foward<T>(args)...);
}
```

전달 expand

```
auto string_maker = [](auto&&... args) {
    return std::string(std::forward<decltype(args)>(args)...);
};
```

전달

std::forward

```
_EXPORT_STD template <class _Ty>
_NODISCARD _MSVC_INTRINSIC constexpr _Ty&& forward(remove_reference_t<_Ty>& _Arg) noexcept {
    return static_cast<_Ty&&>(_Arg);
}
```

```
_EXPORT_STD template <class _Ty>
_NODISCARD _MSVC_INTRINSIC constexpr _Ty&& forward(remove_reference_t<_Ty>&& _Arg) noexcept {
    static_assert(!is_lvalue_reference_v<_Ty>, "bad forward call");
    return static_cast<_Ty&&>(_Arg);
}
```

```
struct MyClass {
    template<typename T>
    MyClass(T&& str) : data(std::forward<T>(str)) {}

    std::string data;
};
```

std::string&	➡	T = std::string&	➡	_Arg = std::string&
std::string&&	➡	T = std::string	➡	_Arg = std::string&&

std::forward

```
_EXPORT_STD template <class _Ty>
_NODISCARD _MSVC_INTRINSIC constexpr _Ty&& forward(remove_reference_t<_Ty>& _Arg) noexcept {
    return static_cast<_Ty&&>(_Arg);
}
```

```
_EXPORT_STD template <class _Ty>
_NODISCARD _MSVC_INTRINSIC constexpr _Ty&& forward(remove_reference_t<_Ty>&& _Arg) noexcept {
    static_assert(!is_lvalue_reference_v<_Ty>, "bad forward call");
    return static_cast<_Ty&&>(_Arg);
}
```

레퍼런스 규칙

T&& & = T&
T&& && = T&&

`_Ty = std::string&` ➡ `_Arg = std::string&` ➡ `_Ty&& = std::string& && = std::string&`
`_Ty = std::string` ➡ `_Arg = std::string&&` ➡ `_Ty&& = std::string&&`

Move Semantics

```
struct MyClass {  
    MyClass(MyClass&& other) noexcept;  
    MyClass& operator=(MyClass&& other) noexcept;  
};
```

```
std::string a = "Hello";
```

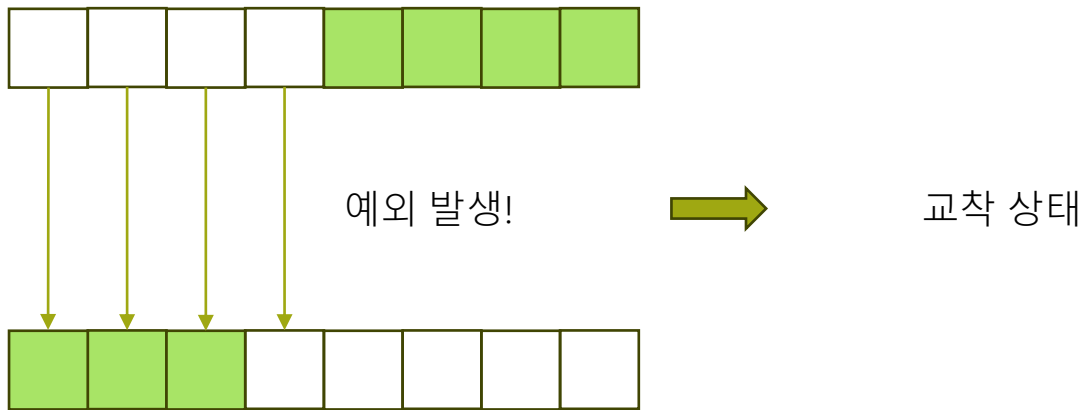
```
std::string b(std::move(a)); // b = "Hello ", a = ""  
std::string c = std::move(a); // c = "", a = ""
```

```
template <class _Ty>  
constexpr remove_reference_t<_Ty>&& move(_Ty&& _Arg) noexcept {  
    return static_cast<remove_reference_t<_Ty>&&>(_Arg);  
}
```

객체를 이동시키는게 아니라
rvalue casting 하고 있음

std::vector 와 noexcept

- `MyApplication` 을 갖는 `std::vector` 를 만들어 봅니다
- `MyApplication` 의 이동 생성자의 `noexcept` 지시자가 붙어 있다면 되어 있다면 해당 지시자를 제거합니다
- `std::vector<MyApplication>` 객체 생성 후, `emplace_back()` 을 10번 호출하고 어떤 내용이 출력되는지 확인합니다



실습 12 – 완벽해지기

- `MyVector` 에 다음 함수를 추가합니다
- `begin()`
- `end()`
- `std::initialize_list` 를 인자로 받는 생성자

```
const MyVector<int> v = {1, 2, 3, 4, 5};
```

```
for (const auto& x : v) {  
    std::cout << x << ' ';  
}
```