모던 C++

Smart Pointers

RAII

Resource Acquisition Is Initialization

자원의 획득은 초기화이다

`std::move`
실제로 무언가를 이동시키지 않음!

결자해지: 네가 메모리 할당했으면 네가 메모리 해제해라

RAII

라이브러리(클래스) 작성자

```
typedef int* IntArray;

IntArray CreateIntArray(int size) {
  IntArray arr = (IntArray)malloc(sizeof(int) * size);
  return arr;
}

void DeleteIntArray(IntArray arr) {
  free(arr);
}
```

라이브러리(클래스) 사용자

```
IntArray array = CreateIntArray(10);
// do something

DeleteIntArray(array);
```

내가 하기 싫다!

RAII

라이브러리(클래스) 작성자

```
template<typename T>
class MyVector {
 public:
  MyVector(int size) {
    arr = new T[size];
  }

  ~MyVector() {
    delete[] arr;
  }

 private:
  T* arr = nullptr;
};
```

소멸자에서 자원 해제

라이브러리(클래스) 사용자

```
MyVector<int> vec(10);
```

배열 말고 객체 1개만을 동적 할당 하려면?

Smart Pointers

- RAII – 자원 해제
- 분명한 소유권

```
void DeleteIntArray(IntArray arr) {
  free(arr);
}
```

```
void SquareIntArray(IntArray arr) {
  arr[i] = arr[i] * arr[i]
}
```

인자의 타입이 같음: 소유권 불분명

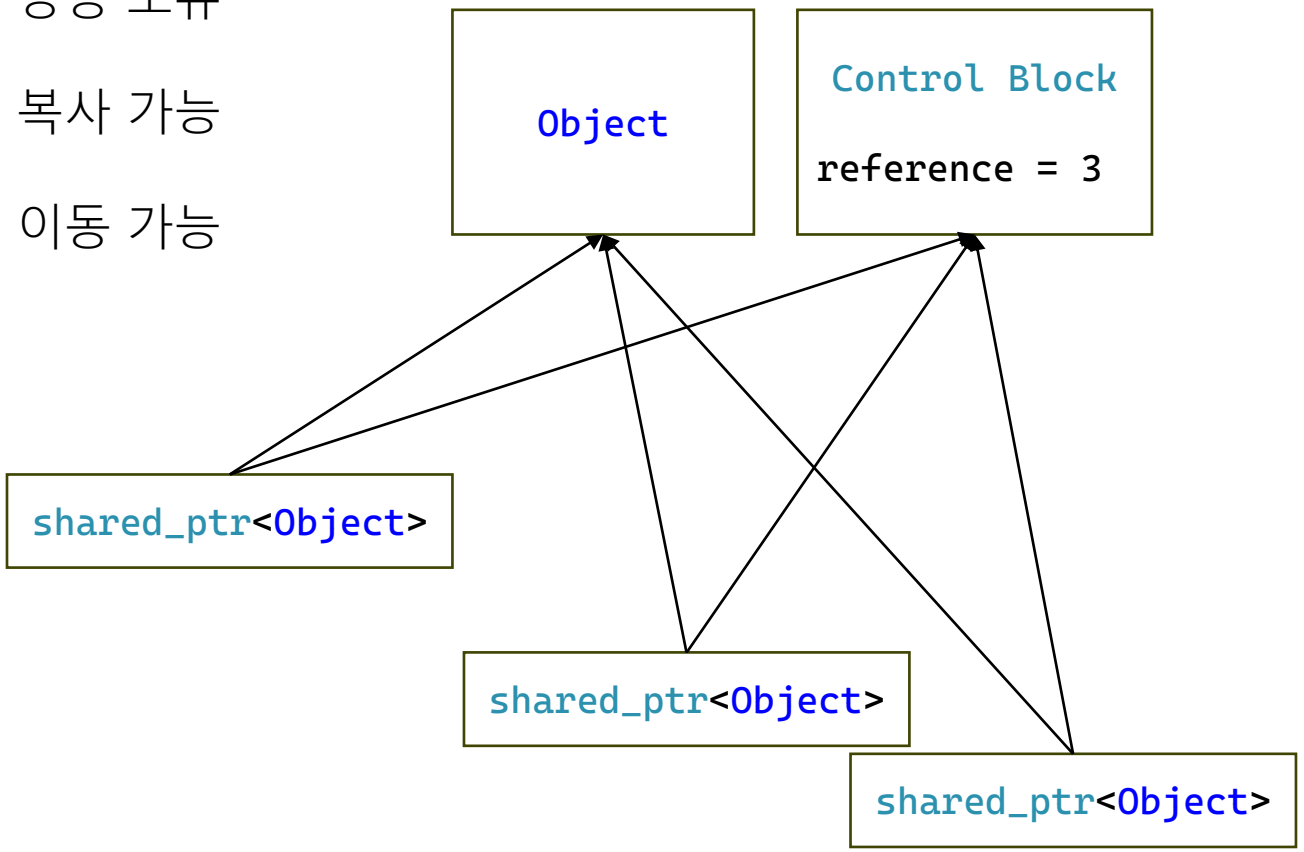
unique_ptr

- 단일 소유
- 복사 불가
- 이동 가능

```cpp
auto x = std::make_unique<int>(10);

std::unique_ptr<double> y = std::make_unique<double>(3.1415);



// 컴파일 안 됨!
std::unique_ptr<double> z = y;

// 컴파일 됨
std::unique_ptr<double> w = std::move(y);
```

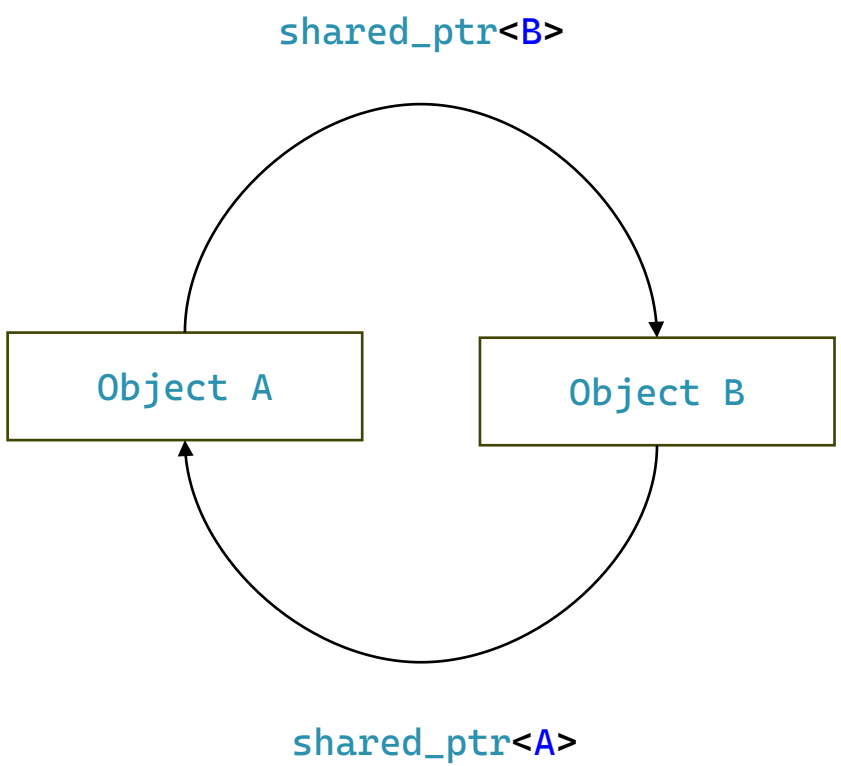
shared_ptr

- 공동 소유
- 복사 가능
- 이동 가능




```
auto x = std::make_shared<int>(10);
auto y = x;
auto z = y;
```

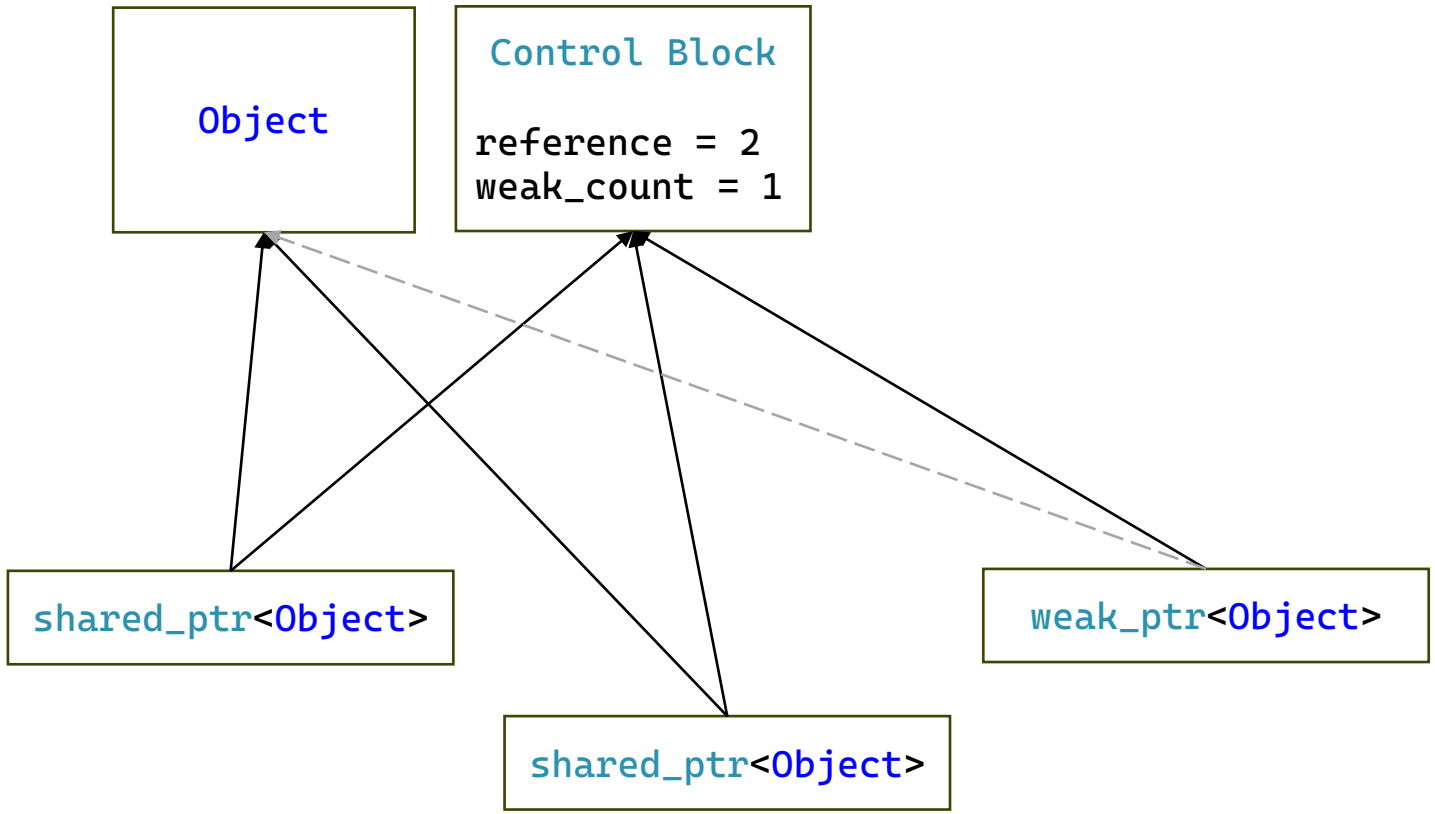
shared_ptr – 순환 참조 문제

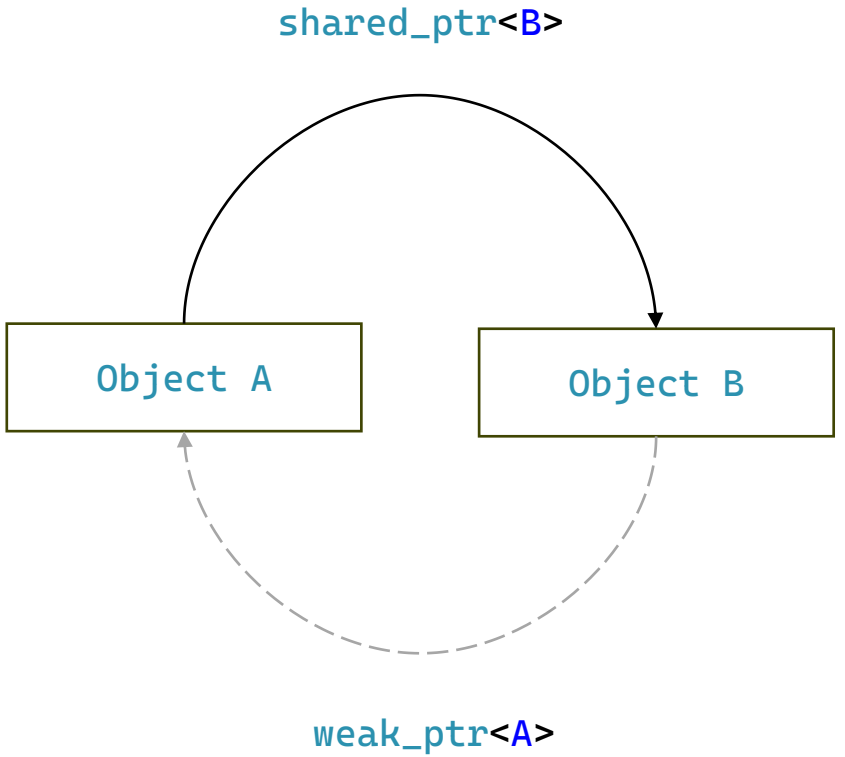


A 의 멤버 변수인 shared_ptr<B> 가 파괴되어야 A 가 파괴된다

B 의 멤버 변수인 shared_ptr<A> 가 파괴되어야 B 가 파괴된다

shared_ptr – 순환 참조 문제



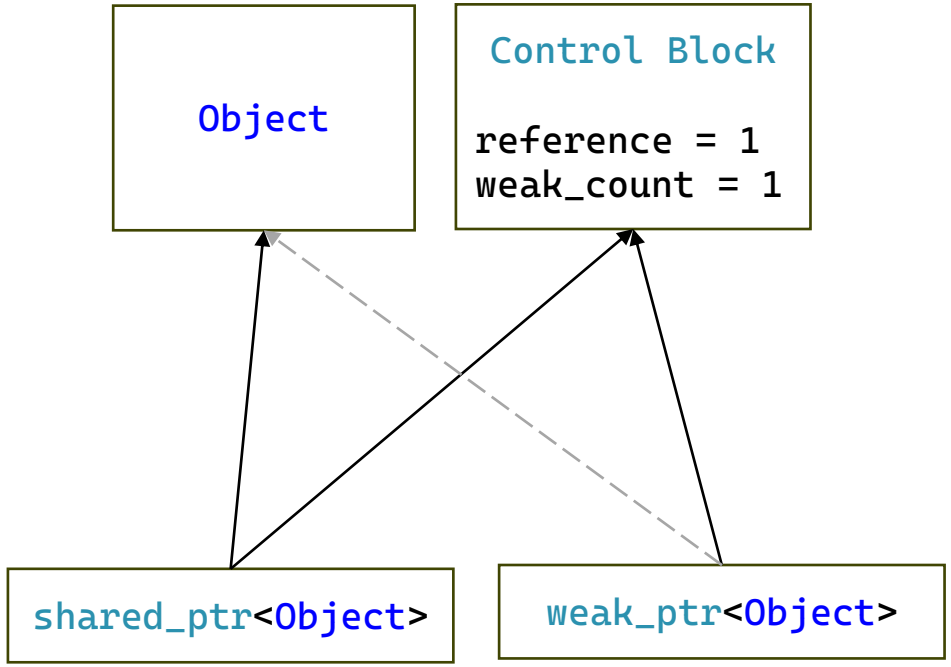


weak_ptr

- 약한 소유권
  - 참조 개수에 영향을 미치지 않음
- 소유권 획득 가능
  - **shared_ptr** 획득 가능
- 복사 가능
- 이동 가능

```
auto x = std::make_shared<int>(10);
auto y = x;
std::weak_ptr z = y;

y.reset();
std::shared_ptr<int> w = z.lock();
```

shared_ptr / weak_ptr - concurrency



참조 횟수가 변경되는 것이 스레드에 대해 안전함(thread safe)

```
std::weak_ptr z = /* */;
std::shared_ptr<int> w = z.lock();

if (w) {
  // do something with w
}
```

다만, 소유권 획득은 확인 절차가 필요

make_unique / make_shared

C++ 17 이전에는 함수 인자로 전달되는 expression 의 평가 순위(evaluation order)에 대한 규약이 없음

```
void my_function(int a, int b) {

}


my_function(funcB(funcA()), funcC());
```

3가지 경우의 수

```
1. funcA()
2. funcB()
3. funcC()
```

```
1. funcA()
2. funcC()
3. funcB()
```

```
1. funcC()
2. funcA()
3. funcB()
```

make_unique / make_shared

```
void my_function(std::unique_ptr<int> a, int b);
```

```
my_function(
    std::unique_ptr<int>(new int(1)),
    funcA());
```

예외가 발생한다면?

1. new int(1)
2. funcA()
3. std::unique_ptr::unique_ptr()

new 를 통한 메모리 할당이 이루어 졌지만 std::unique_ptr 의 생성자가 호출 되기 이전에 예외가 발생한다면 메모리가 leak이 발생하게 된다.

make_unique / make_shared

`std::make_unique<T>(Args&&...)` 의 반환 타입은 `std::unique_ptr<T>`
`std::make_shared<T>(Args&&...)` 의 반환 타입은 `std::shared_ptr<T>`

두 함수 모두 인자 `Args&&...` 를 `T`의 생성자로 perfect forwarding

```
// p1 = std::unique_ptr<int>
auto p1 = std::make_unique<int>(1);

// p2 = std::unique_ptr<std::vector<int>>
auto p2 = std::make_unique<std::vector<int>>(1, 10);

// auto p3 = std::shared_ptr<int>
auto p3 = std::make_shared<int>(1);

std::shared_ptr<int> p4 = std::make_unique<int>(2);
```

사용자 정의 deleter

`std::unique_ptr<T>` 과 `std::shared_ptr<T>` 은 `T` 가 클래스인 경우 모두 소멸자(`~T()`)를 호출합니다

이 대신 사용자가 원하는 소멸 함수가 호출되도록 정의할 수 있습니다.

```
struct MyApplication {};

void MyDeleter(MyApplication* p) {
  std::cout << "MyDeleter\n";
}
```

```
// std::unique_ptr의 경우 타입 정의시 소멸 함수의 타입을 명시해야 함
std::unique_ptr<MyApplication, void(*)(MyApplication*)> p(new MyApplication(), &MyDeleter);

// std::shared_ptr의 경우 type erasure 방식으로 구현되어 있어 타입을 따로 명시해주지 않아도 됨
std::shared_ptr<MyApplication> p2(new MyApplication(), &MyDeleter);
```

Raw pointer를 사용해야 할 때 (1)

- `std::unique_ptr / std::shared_ptr` 와 무관하게 인자로 받아서 동작해야 할 때

  -> 함수 자체는 소유권에 대해서 관심이 없다

```
struct MyApplication {
  MyApplication() = default;

  void DoSomething() {}
};


void Process(MyApplication* app) {
  app->DoSomething();
}
```

```
auto p1 = std::make_unique<MyApplication>();
Process(p1.get());

auto p2 = std::make_shared<MyApplication>();
Process(p2.get());
```

Raw pointer를 사용해야 할 때 (2)

- 함수의 return으로 값을 전달하는게 아니라 인자에 써서 전달하는 경우
  -> 가독성을 위해 C 스타일 사용


```
void add(int a, int b, int& out) {
  out = a + b;
}

int a = 1;
int b = 2;
int c;

// 여기만 봐서는 c 가 뭐하는건지 바로 알기 어렵다!
add(a, b, c);
```

```
void add_easy(int a, int b, int* out) {
  *out = a + b;
}

int a = 1;
int b = 2;
int c;

// c 는 값이 수정된다는것을 바로 알 수 있다!
add_easy(a, b, &c);
```

실습 - 13

1. 실습 9에서 구현한 **MyApplication** 클래스를 가져옵니다.
2. 아래 두 함수를 동일하게 구현합니다.
3. (다음 장에 이어서)

```
void TestUniquePtr() {
  // MyApplication 생성
  auto p = std::make_unique<MyApplication>();

  // MyApplication 이동
  auto p2 = std::move(p);
}
```

```
void TestSharedPtr() {
  // MyApplication 생성
  auto p = std::make_shared<MyApplication>();

  // ?
  auto p2 = p;

  // p2 초기화
  p2.reset();

  // ?
  std::weak_ptr<MyApplication> weak_p = p;

  auto lock = weak_p.lock();
  if (lock) {
    std::cout << "Acquired ownership\n";
  }
}
```

실습 - 13

3. **main** 함수 안에서 **TestUniquePtr, TestSharedPtr** 두 함수 호출 이후
   특수 멤버 함수들이 각각 몇 번씩 호출되었는지 출력합니다.

```
int default_constructor_count = 0;
// ...

int main() {
  TestUniquePtr();
  std::cout << "기본 생성자 호출 횟수: " << default_constructor_count << '\n';
  std::cout << "복사 생성자 호출 횟수: " << /* ... */ << '\n';
  std::cout << "이동 생성자 호출 횟수: " << /* ... */ << '\n';
  std::cout << "복사 대입 연산자 호출 횟수: " << /* ... */ << '\n';
  std::cout << "이동 대입 연산자 호출 횟수: " << /* ... */ << '\n';
  std::cout << "소멸자 호출 횟수: " << /* ... */ << '\n';

  TestSharedPtr();
  std::cout << "기본 생성자 호출 횟수: " << /* ... */ << '\n';
  // ...

  return 0;
}
```

실습 - 14

스마트 포인터를 사용해도 다형성(polymorphism)이 유지되는지 테스트 해보는 실습입니다.

1. `MyBaseClass` 를 정의하고, `virtual void Hello()`를 정의합니다
   -> `Hello()` 호출 시, `"This is Base"`를 출력합니다.
2. `MyBaseClass` 를 `public` 상속 받는 `MyDerivedClass()`를 정의합니다
   -> `Hello()` 호출 시, `"This is Derived"` 를 출력합니다.
3. 아래 코드를 main 에 추가하고, 실행 시 어떤게 출력되는지 확인합니다

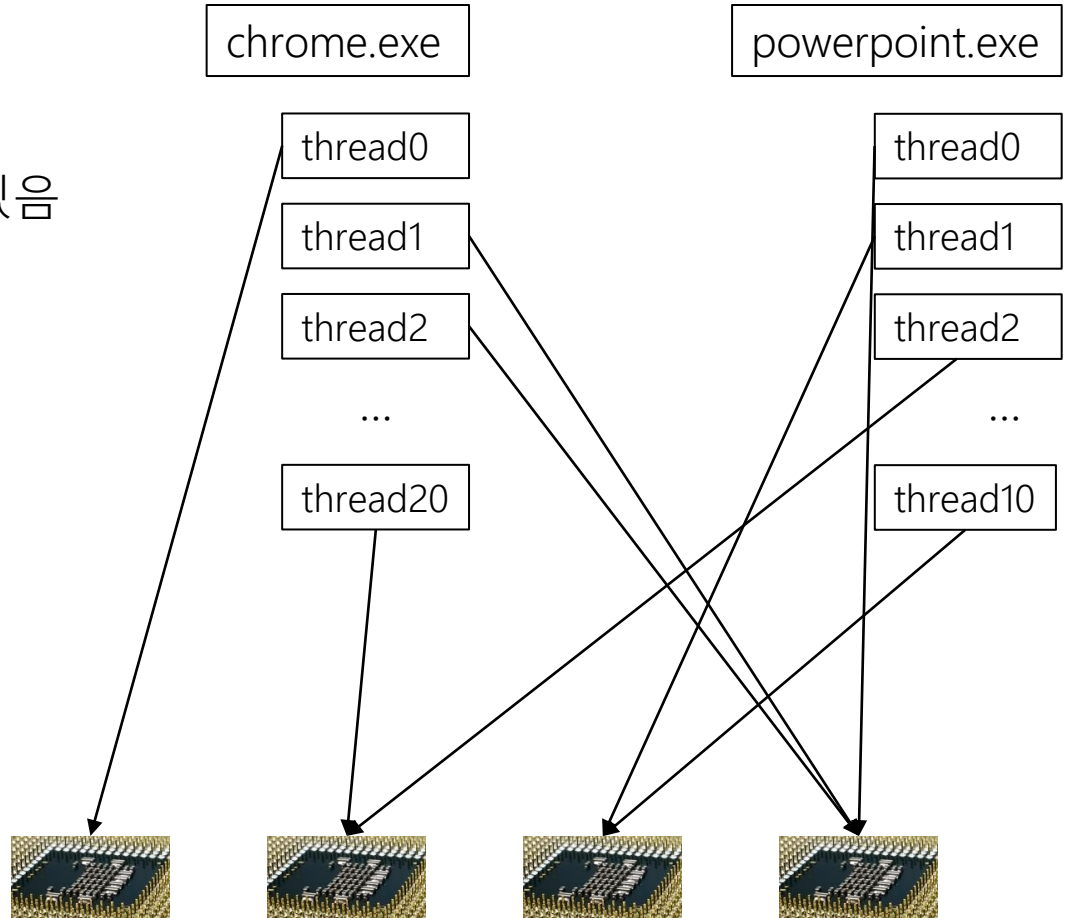
```
int main() {
  std::unique_ptr<MyBaseClass> p = std::make_unique<MyDerivedClass>();
  p->Hello();

  return 0;
}
```

모던 C++

Concurrency

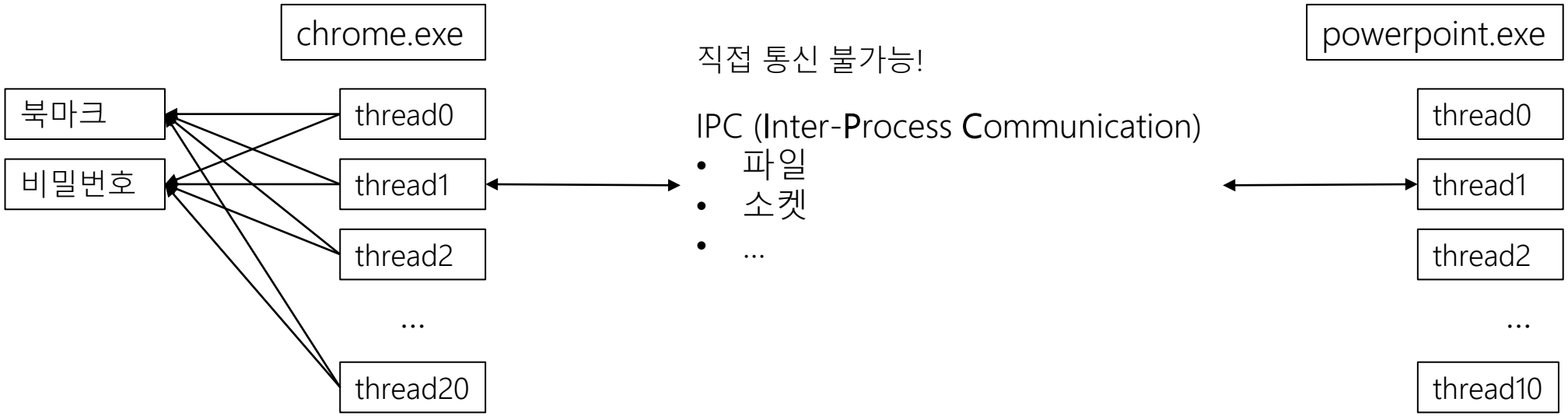
스레드 (thread)

- 독립된 최소한의 실행 흐름
- 하나의 프로세스는 여러 스레드를 가질 수 있음



스레드와 프로세스

- 프로세스는 서로 자원을 공유하지 않음
- (동일한 프로세스의) 스레드는 서로 자원을 공유함

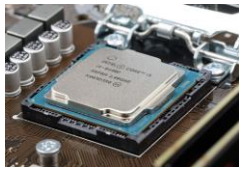


하드웨어 스레드와 소프트웨어 스레드

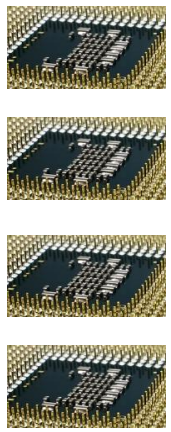
- 하드웨어 스레드는 실제 스레드가 실행되는 물리적 실체(CPU / Core)

- 소프트웨어 스레드는 시분할(time-slicing) 되어 하드웨어 스레드에서 실행 됨

Concurrency

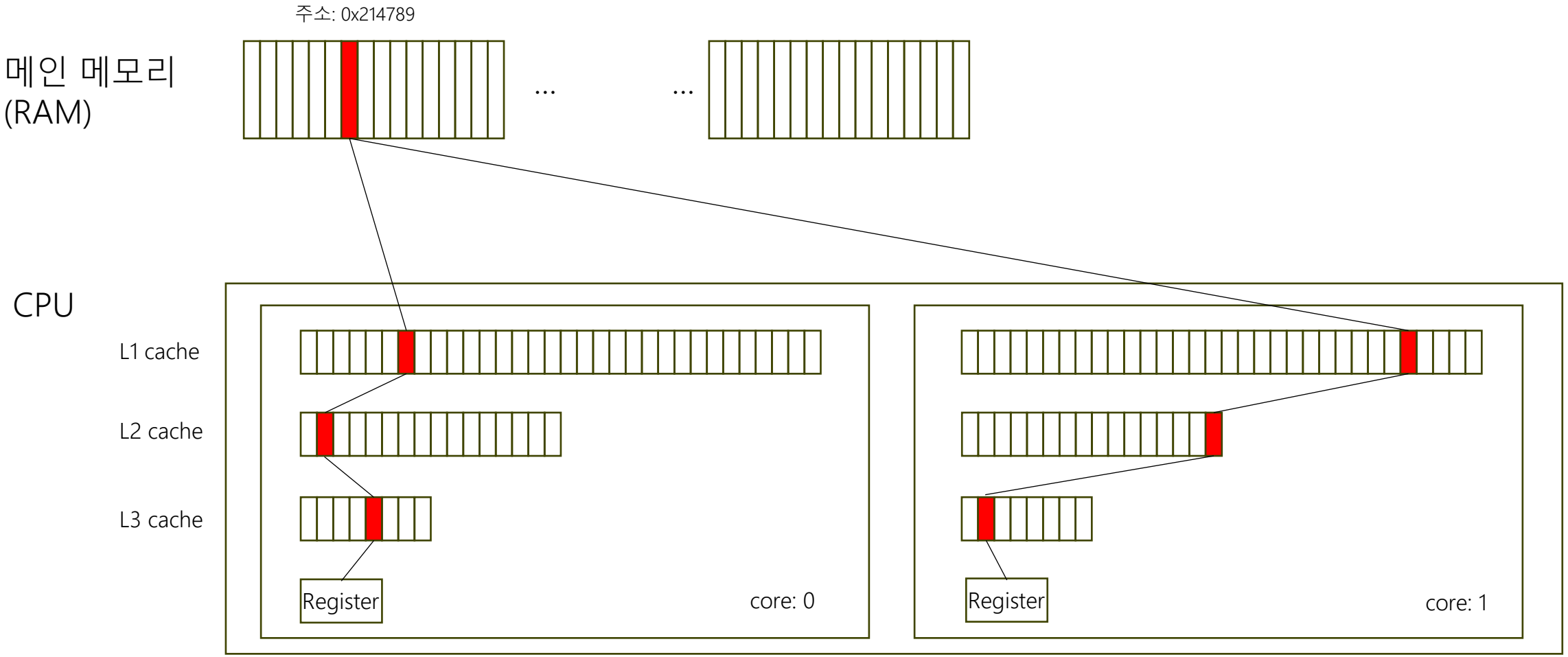


1	2	3	4	...	100



임계 구역(critical section)
-> 동기화(synchronization) 필요!

원자성 (atomic)


주소: 0x214789
메인 메모리
(RAM)
CPU
L1 cache
L2 cache
L3 cache
Register
core: 0
Register
core: 1

원자성 (atomic)

쓰기(write) = 읽기(read) + 수정(modify) + 쓰기(write)

원자적 연산: 다른 스레드는 중간 과정을 관측할 수 없음

원자성 (atomic)

atomic

- integral types
- pointer types
- floating point types (C++ 20)
- Defined in header `<atomic>`

- **`load`**
- **`store`**
- **`add`**
- **`sub`**
- **`exchange`**
- **`compare_exchange`**

Memory Order

컴파일러는 컴파일 시 우리가 작성한 소스코드를 순서대로 옮기지 않음!

```
int a = 1;
int b = 2;
int c = a + 1;
int d = b + 1;
```

```
int a = 1;
int c = a + 1;
int b = 2;
int d = b + 1;
```

c는 a에 의존, d 는 b에 의존
-> 서로 의존성이 없는 코드는 순서가 보장되지 않습니다

Memory Order

`memory_order_relaxed`	순서를 전혀 보장하지 않음
`memory_order_consume`	이후에 적힌 연산 중 atomic 객체와 종속성이 있는 연산은 해당 기준 이전으로 재배치 될 수 없음
`memory_order_acquire`	이후에 적힌 연산은 해당 기준 이전으로 재배치 될 수 없음
`memory_order_release`	이전에 적힌 연산은 해당 기준 이후로 재배치 될 수 없음
`memory_order_acq_rel`	
`memory_order_seq_cst`	소스코드와 같은 순서를 보장함

```
std::atomic_thread_fence(std::memory_order order)
```

개별 연산이나 전체 연산에 대해 memory order를 설정할 수 있음

thread

- 함수 하나를 비동기로 실행할 수 있음.
  - 객체 생성과 동시에 함수가 실행되고, 함수 실행이 완료되지 않아도 객체는 생성 됨
- 프로세스 종료 전, 실행중인 모든 스레드들을 정상적으로 종료 해 주어야 함

```
#include <thread>

auto thread = std::thread([&]() {
  for (int i = 0; i < 100'000; ++i) {
    // do something
  }
});

// wait for unfinished threads
if (thread.joinable()) {
  thread.join();
}
```

this_thread

- 현재 스레드에게 일정 시간 대기하라고 요청

```
// 특정 시간 동안 대기
std::this_thread::sleep_for(std::chrono::seconds(10));

// 특정 시각 까지 대기
std::this_thread::sleep_until(std::chrono::system_clock::now() + std::chrono::seconds(10));
```

- 현재 스레드의 식별자 반환

```
std::cout << std::this_thread::get_id() << '\n';
```

mutex

- 상호 배제(Mutual Exclusion) 알고리즘 기반
- 임계 구역에 접근하는 스레드를 제한시킴

➡ 해당 구간(연산) 을 원자적으로 만듦

- 크기가 1인 세마포어(semaphore)

➡ 세마포어는 임계 구역에 동시에 접근가능한 스레드를 제한함

mutex

```
int data = 1;
std::mutex m;

std::thread t1([&](){
  std::lock_guard<std::mutex> lck(m);
  data += 1;
});

std::thread t2([&]() {
  std::lock_guard<std::mutex> lck(m);
  data += 1;
});

// data = 3
std::cout << data << '\n';
```

1. lock을 획득 하기 전 까지 대기

2-1. lock을 획득 했으면 대기 상태를 벗어나 코드 실행

-> Scope 를 벗어나면 **mutex**의 소유권을 반환

2-2. 다른 스레드가 먼저 **mutex**의 소유권을 획득 한 경우

-> 1번으로 돌아감

lock_guard

가장 단순한 종류

생성함과 동시에 mutex의 소유권 획득을 시도하거나 소유권을 이양 받음

mutex의 소유권을 반환하려면 객체가 파괴되어야 함(scope 벗어남)

```
std::mutex m;

{
  // 소유권 획득 시도
  std::lock_guard<std::mutex> lck(m);
  // 소유권 획득
}
// 스코프 벗어남; 소유권 반환
```

unique_lock

생성함시에 mutex의 소유권 획득 시도 여부를 결정할 수 있음(획득, 연기)

소유권을 반환하고 획득하는 시도를 계속 할 수 있음

```
std::mutex m;

{
  // 객체 생성하지만 소유권은 아직 가지지 않음
  std::unique_lock<std::mutex> lck(m, std::defer_lock);

  // 소유권 획득 시도
  lck.lock();

  // 소유권 획득

  // 소유권 반환
  lck.unlock();

  // 소유권 획득 시도
  lck.lock();

  // 소유권 획득
}
// 스코프 벗어남; 소유권 반환
```

conditional_variable

lock은 `mutex`의 소유권 획득을 시도하면 소유권을 획득할 때 까지 무한 대기하게 됩니다
이는 스레드가 실제 하드웨어 스레드에서 실행될 때 아무것도 하지 않게 되기 때문에 프로그램의 성능이 하락할 수 있습니다

`conditional_variable` 을 이용해서 명시적인 알림(notification)을 받기 전 까지 다른 작업을 하라고 지시할 수 있습니다

conditional_variable

```
std::mutex m;
std::condition_variable cv;
bool wakeup = false;

std::thread th1([&]() {
  std::unique_lock<std::mutex> lck(m);

  // 스레드가 실행되었지만 wakeup의 초깃값이 false 이기 때문에 대기 상태로 들어감
  cv.wait(lck, [&]() {
    // wakeu이 true 일 때 까지 대기
    return wakeup;
  });
});


std::thread th2([&]() {
  // wakeup 는 다른 스레드에서도 관측하고 있으므로 mutex를 이용해 값을 수정
  std::unique_lock<std::mutex> lck(m);
  wakeup = true;
  lck.unlock();

  // cv 를 이용해 대기하고 있는 스레드 한개에게 작업 진행 알림을 보냄
  cv.notify_one();
});
```

async

- 추상화 된 비동기 함수 객체
  - 객체 생성과 동시에 함수를 실행하거나, 실행하지 않고 나중에 실행할 수 있음
- 프로세스 종료 전, 실행중인 모든 함수들을 종료 해 주어야 함

```
auto a = std::async([](int begin, int end) {
  for (int i = begin; i < end; ++i) {
    // do something
  }
}, 0, 100);

auto b = std::async(std::launch::deferred, /* function */);

auto c = std::async(std::launch::async, /* function */);

c.get();
b.wait();
```