|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | **작성자** | **2014132002 구태균** | **팀명** | - |
| **주차** | **3 주차** | **기간** | **2020.01.27~2020.02.02** | **지도교수** | **정 내 훈** (서명) |
| **이번주 한일** | shared\_ptr 상속 클래스 함수 분석 | | | | |

**---------------------------<shared\_ptr\_base.h>---------------------------**

**[\_\_shared\_ptr]**

template<typename \_Tp, \_Lock\_policy \_Lp>

class \_\_shared\_ptr : public \_\_shared\_ptr\_access<\_Tp, \_Lp>

{

public:

constexpr \_\_shared\_ptr() noexcept

: \_M\_ptr(0), [\_M\_refcount()](#shared_count_)

{

// [element\_type\*] \_M\_ptr = 0

// [\_\_shared\_count.\_Sp\_counted\_base] \_M\_refcount.\_M\_pi = 0

}

template<typename \_Yp, typename = \_SafeConv<\_Yp>>

explicit \_\_shared\_ptr(\_Yp\* \_\_p)

: \_M\_ptr(\_\_p), [\_M\_refcount](#shared_count_ptr_arraytype)(\_\_p, typename is\_array<\_Tp>::type())

{

static\_assert(!is\_void<\_Yp>::value, "incomplete type");

// 설명 : \_Yp 형식이 void형식이라면 컴파일 실패

// static\_assert : <https://docs.microsoft.com/ko-kr/cpp/cpp/static-assert?view=vs-2019>

// is\_void : <https://en.cppreference.com/w/cpp/types/is_void>

static\_assert(sizeof(\_Yp) > 0, "incomplete type");

// \_Yp 크기가 0이라면 컴파일 실패

[\_M\_enable\_shared\_from\_this\_with](#M_enable_shared_from_this_with)(\_\_p);

// [element\_type\*] \_M\_ptr = \_\_p

// [\_\_shared\_count.\_Sp\_counted\_base] \_M\_refcount.\_M\_pi = 0

}

template<typename \_Yp, typename \_Deleter, typename = \_SafeConv<\_Yp>>

\_\_shared\_ptr(\_Yp\* \_\_p, \_Deleter \_\_d)

: \_M\_ptr(\_\_p), [\_M\_refcount](#shared_count_ptr_deleter)(\_\_p, std::move(\_\_d))

{

static\_assert(\_\_is\_invocable<\_Deleter&, \_Yp\*&>::value,

"deleter expression d(p) is well-formed");

// 설명 : deleter로 호출할 수 있다면 컴파일 실패

// \_\_is\_invocable : <https://en.cppreference.com/w/cpp/types/is_invocable>

[\_M\_enable\_shared\_from\_this\_with](#M_enable_shared_from_this_with)(\_\_p);

}

template<typename \_Yp, typename \_Deleter, typename \_Alloc,typename = \_SafeConv<\_Yp>>

\_\_shared\_ptr(\_Yp\* \_\_p, \_Deleter \_\_d, \_Alloc \_\_a)

: \_M\_ptr(\_\_p), [\_M\_refcount](#shared_count_ptr_deleter_allocator)(\_\_p, std::move(\_\_d), std::move(\_\_a))

{

static\_assert(\_\_is\_invocable<\_Deleter&, \_Yp\*&>::value,

"deleter expression d(p) is well-formed");

[\_M\_enable\_shared\_from\_this\_with](#M_enable_shared_from_this_with)(\_\_p);

}

// 파라미터 == nullptr

template<typename \_Deleter>

\_\_shared\_ptr(nullptr\_t \_\_p, \_Deleter \_\_d)

: \_M\_ptr(0), [\_M\_refcount](#shared_count_ptr_deleter)(\_\_p, std::move(\_\_d))

{ }

template<typename \_Deleter, typename \_Alloc>

\_\_shared\_ptr(nullptr\_t \_\_p, \_Deleter \_\_d, \_Alloc \_\_a)

: \_M\_ptr(0), [\_M\_refcount](#shared_count_ptr_deleter_allocator)(\_\_p, std::move(\_\_d), std::move(\_\_a))

{ }

// Aliasing constructor

template<typename \_Yp>

\_\_shared\_ptr(const \_\_shared\_ptr<\_Yp, \_Lp>& \_\_r, element\_type\* \_\_p) noexcept

: \_M\_ptr(\_\_p), [\_M\_refcount](#shared_count_const_ref_r)(\_\_r.\_M\_refcount) // never throws

{ }

// 복사 : refcount =+ 1

// Aliasing constructor

template<typename \_Yp>

\_\_shared\_ptr(\_\_shared\_ptr<\_Yp, \_Lp>&& \_\_r, element\_type\* \_\_p) noexcept

: \_M\_ptr(\_\_p), [\_M\_refcount](#shared_count_)()

{

\_M\_refcount.[\_M\_swap](#shared_count_func__M_swap)(\_\_r.\_M\_refcount);

\_\_r.\_M\_ptr = 0;

}

\_\_shared\_ptr(const \_\_shared\_ptr&) noexcept = default;

\_\_shared\_ptr& operator=(const \_\_shared\_ptr&) noexcept = default;

~\_\_shared\_ptr() = default;

template<typename \_Yp, typename = \_Compatible<\_Yp>>

\_\_shared\_ptr(const \_\_shared\_ptr<\_Yp, \_Lp>& \_\_r) noexcept

: \_M\_ptr(\_\_r.\_M\_ptr), [\_M\_refcount](#shared_count_const_ref_r)(\_\_r.\_M\_refcount)

{ }

// 복사 : refcount =+ 1

\_\_shared\_ptr(\_\_shared\_ptr&& \_\_r) noexcept

: \_M\_ptr(\_\_r.\_M\_ptr), \_M\_refcount()

{

\_M\_refcount[.\_M\_swap](#shared_count_func__M_swap)(\_\_r.\_M\_refcount);

\_\_r.\_M\_ptr = 0;

}

// 이동 : refcount 그대로

template<typename \_Yp, typename = \_Compatible<\_Yp>>

\_\_shared\_ptr(\_\_shared\_ptr<\_Yp, \_Lp>&& \_\_r) noexcept

: \_M\_ptr(\_\_r.\_M\_ptr), \_M\_refcount()

{

\_M\_refcount.[\_M\_swap](#shared_count_func__M_swap)(\_\_r.\_M\_refcount);

\_\_r.\_M\_ptr = 0;

}

// 이동 : refcount 그대로

template<typename \_Yp, typename = \_Compatible<\_Yp>>

explicit \_\_shared\_ptr(const \_\_weak\_ptr<\_Yp, \_Lp>& \_\_r)

: \_M\_refcount(\_\_r.\_M\_refcount) // may throw

{

// It is now safe to copy \_\_r.\_M\_ptr, as

// \_M\_refcount(\_\_r.\_M\_refcount) did not throw.

\_M\_ptr = \_\_r.\_M\_ptr;

}

// If an exception is thrown this constructor has no effect.

template<typename \_Yp, typename \_Del, typename = \_UniqCompatible<\_Yp, \_Del>>

\_\_shared\_ptr(unique\_ptr<\_Yp, \_Del>&& \_\_r)

: \_M\_ptr(\_\_r.get()), \_M\_refcount()

{

auto \_\_raw = \_\_to\_address(\_\_r.get());

\_M\_refcount = \_\_shared\_count<\_Lp>(std::move(\_\_r));

\_M\_enable\_shared\_from\_this\_with(\_\_raw);

}

constexpr \_\_shared\_ptr(nullptr\_t) noexcept : [\_\_shared\_ptr](#shared_ptr_)() { }

template<typename \_Yp>

\_Assignable<\_Yp> operator=(const \_\_shared\_ptr<\_Yp, \_Lp>& \_\_r) noexcept

{

\_M\_ptr = \_\_r.\_M\_ptr;

\_M\_refcount = \_\_r.\_M\_refcount; // \_\_shared\_count::op= doesn't throw

return \*this;

}

\_\_shared\_ptr& operator=(\_\_shared\_ptr&& \_\_r) noexcept

{

[\_\_shared\_ptr(std::move(\_\_r))](#shared_ptr_move_r).[swap](#shared_ptr_func_swap)(\*this);

return \*this;

}

template<class \_Yp>

\_Assignable<\_Yp> operator=(\_\_shared\_ptr<\_Yp, \_Lp>&& \_\_r) noexcept

{

[\_\_shared\_ptr(std::move(\_\_r))](#shared_ptr_move_r).[swap](#shared_ptr_func_swap)(\*this);

return \*this;

}

// unique\_ptr

template<typename \_Yp, typename \_Del>

\_UniqAssignable<\_Yp, \_Del> operator=(unique\_ptr<\_Yp, \_Del>&& \_\_r)

{

[\_\_shared\_ptr(std::move(\_\_r))](#shared_ptr_move_r).[swap](#shared_ptr_func_swap)(\*this);

return \*this;

}

void reset() noexcept

{

[\_\_shared\_ptr()](#shared_ptr_).[swap](#shared_ptr_func_swap)(\*this);

}

template<typename \_Yp>

\_SafeConv<\_Yp> reset(\_Yp\* \_\_p) // \_Yp must be complete.

{

// Catch self-reset errors.

\_\_glibcxx\_assert(\_\_p == 0 || \_\_p != \_M\_ptr);

[\_\_shared\_ptr(\_\_p)](#shared_ptr_ptr).[swap](#shared_ptr_func_swap)(\*this);

}

template<typename \_Yp, typename \_Deleter>

\_SafeConv<\_Yp> reset(\_Yp\* \_\_p, \_Deleter \_\_d)

{

[\_\_shared\_ptr(\_\_p, std::move(\_\_d))](#shared_ptr_ptr_deleter).[swap](#shared_ptr_func_swap)(\*this);

}

template<typename \_Yp, typename \_Deleter, typename \_Alloc>

\_SafeConv<\_Yp> reset(\_Yp\* \_\_p, \_Deleter \_\_d, \_Alloc \_\_a)

{

[\_\_shared\_ptr(\_\_p, std::move(\_\_d), std::move(\_\_a))](#shared_ptr_ptr_deleter_allocator).[swap](#shared_ptr_func_swap)(\*this);

}

/// Return the stored pointer.

element\_type\* get() const noexcept

{

return \_M\_ptr;

// get [element\_type\*]

}

/// Return true if the stored pointer is not null.

explicit operator bool() const // never throws

{

return \_M\_ptr == 0 ? false : true;

// [element\_type\*] != null 이면, true

// [element\_type\*] == null 이면, false

}

/// Return true if use\_count() == 1.

bool unique() const noexcept

{

return \_M\_refcount.[\_M\_unique](#shared_count__M_unique)();

// \_M\_refcount.\_M\_use\_count == 1 이면, true

}

/// If \*this owns a pointer, return the number of owners, otherwise zero.

long use\_count() const noexcept

{

return \_M\_refcount.\_M\_get\_use\_count();

}

/// Exchange both the owned pointer and the stored pointer.

void swap(\_\_shared\_ptr<\_Tp, \_Lp>& \_\_other) noexcept

{

std::swap(\_M\_ptr, \_\_other.\_M\_ptr);

\_M\_refcount.[\_M\_swap](#shared_count_func__M_swap)(\_\_other.\_M\_refcount);

}

/\*\* @brief Define an ordering based on ownership.

\*

\* This function defines a strict weak ordering between two shared\_ptr

\* or weak\_ptr objects, such that one object is less than the other

\* unless they share ownership of the same pointer, or are both empty.

\* @{

\*/

// less : <https://ko.cppreference.com/w/cpp/utility/functional/less>

template<typename \_Tp1>

bool owner\_before(\_\_shared\_ptr<\_Tp1, \_Lp> const& \_\_rhs) const noexcept

{

return \_M\_refcount.[\_M\_less](#shared_count__M_less_shared_count)(\_\_rhs.\_M\_refcount);

}

template<typename \_Tp1>

bool owner\_before(\_\_weak\_ptr<\_Tp1, \_Lp> const& \_\_rhs) const noexcept

{

return \_M\_refcount.[\_M\_less](#shared_count__M_less_weak_count)(\_\_rhs.\_M\_refcount);

}

protected:

// This constructor is non-standard, it is used by allocate\_shared.

template<typename \_Alloc, typename... \_Args>

\_\_shared\_ptr(\_Sp\_alloc\_shared\_tag<\_Alloc> \_\_tag, \_Args&&... \_\_args)

: \_M\_ptr(), \_M\_refcount(\_M\_ptr, \_\_tag, std::forward<\_Args>(\_\_args)...)

// forward : <https://github.com/jwvg0425/ModernCppStudy/wiki/item-23>

{

[\_M\_enable\_shared\_from\_this\_with](#M_enable_shared_from_this_with)(\_M\_ptr);

}

template<typename \_Tp1, \_Lock\_policy \_Lp1, typename \_Alloc, typename... \_Args>

friend \_\_shared\_ptr<\_Tp1, \_Lp1>

\_\_allocate\_shared(const \_Alloc& \_\_a, \_Args&&... \_\_args);

// This constructor is used by \_\_weak\_ptr::lock() and

// shared\_ptr::shared\_ptr(const weak\_ptr&, std::nothrow\_t).

\_\_shared\_ptr(const \_\_weak\_ptr<\_Tp, \_Lp>& \_\_r, std::nothrow\_t)

: \_M\_refcount(\_\_r.\_M\_refcount, std::nothrow)

{

\_M\_ptr = \_M\_refcount.\_M\_get\_use\_count() ? \_\_r.\_M\_ptr : nullptr;

}

private:

template<typename \_Yp>

using \_\_esft\_base\_t =

decltype(\_\_enable\_shared\_from\_this\_base(std::declval<const \_\_shared\_count<\_Lp>&>(),

std::declval<\_Yp\*>()));

// declval : rvalue reference를 반환

// <http://egloos.zum.com/Lusain/v/3182023>

// Detect an accessible and unambiguous enable\_shared\_from\_this base.

template<typename \_Yp, typename = void>

struct \_\_has\_esft\_base : false\_type { };

template<typename \_Yp>

struct \_\_has\_esft\_base<\_Yp, \_\_void\_t<\_\_esft\_base\_t<\_Yp>>> : \_\_not\_<is\_array<\_Tp>> { };

// No enable shared\_from\_this for arrays

// \_M\_enable\_shared\_from\_this\_with()

template<typename \_Yp, typename \_Yp2 = typename remove\_cv<\_Yp>::type>

typename enable\_if<\_\_has\_esft\_base<\_Yp2>::value>::type

\_M\_enable\_shared\_from\_this\_with(\_Yp\* \_\_p) noexcept

{

if (auto \_\_base = \_\_enable\_shared\_from\_this\_base(\_M\_refcount, \_\_p))

\_\_base->\_M\_weak\_assign(const\_cast<\_Yp2\*>(\_\_p), \_M\_refcount);

}

template<typename \_Yp, typename \_Yp2 = typename remove\_cv<\_Yp>::type>

typename enable\_if<!\_\_has\_esft\_base<\_Yp2>::value>::type

\_M\_enable\_shared\_from\_this\_with(\_Yp\*) noexcept

{ }

void\* \_M\_get\_deleter(const std::type\_info& \_\_ti) const noexcept

{

return \_M\_refcount.[\_M\_get\_deleter](#shared_count__M_get_deleter)(\_\_ti);

}

template<typename \_Del, typename \_Tp1, \_Lock\_policy \_Lp1>

friend \_Del\* get\_deleter(const \_\_shared\_ptr<\_Tp1, \_Lp1>&) noexcept;

template<typename \_Del, typename \_Tp1>

friend \_Del\* get\_deleter(const shared\_ptr<\_Tp1>&) noexcept;

};

template<typename \_Tp1, typename \_Tp2, \_Lock\_policy \_Lp>

inline bool operator==(const \_\_shared\_ptr<\_Tp1, \_Lp>& \_\_a, const \_\_shared\_ptr<\_Tp2, \_Lp>& \_\_b) noexcept

{

return \_\_a.get() == \_\_b.get();

// \_\_a.\_M\_ptr == \_\_b.\_M\_ptr

}

template<typename \_Tp, \_Lock\_policy \_Lp>

inline bool operator==(const \_\_shared\_ptr<\_Tp, \_Lp>& \_\_a, nullptr\_t) noexcept

{

return !\_\_a;

// \_\_a.\_M\_ptr == nullptr

}

template<typename \_Tp, \_Lock\_policy \_Lp>

inline bool operator==(nullptr\_t, const \_\_shared\_ptr<\_Tp, \_Lp>& \_\_a) noexcept

{

return !\_\_a;

// \_\_a.\_M\_ptr == nullptr

}

template<typename \_Tp1, typename \_Tp2, \_Lock\_policy \_Lp>

inline bool operator!=(const \_\_shared\_ptr<\_Tp1, \_Lp>& \_\_a, const \_\_shared\_ptr<\_Tp2, \_Lp>& \_\_b) noexcept

{

return \_\_a.get() != \_\_b.get();

// \_\_a.\_M\_ptr != \_\_b.\_M\_ptr

}

template<typename \_Tp, \_Lock\_policy \_Lp>

inline bool operator!=(const \_\_shared\_ptr<\_Tp, \_Lp>& \_\_a, nullptr\_t) noexcept

{

return (bool)\_\_a;

// \_\_a.\_M\_ptr != nullptr

}

template<typename \_Tp, \_Lock\_policy \_Lp>

inline bool operator!=(nullptr\_t, const \_\_shared\_ptr<\_Tp, \_Lp>& \_\_a) noexcept

{

return (bool)\_\_a;

// \_\_a.\_M\_ptr != nullptr

}

template<typename \_Tp, typename \_Up, \_Lock\_policy \_Lp>

inline bool operator<(const \_\_shared\_ptr<\_Tp, \_Lp>& \_\_a, const \_\_shared\_ptr<\_Up, \_Lp>& \_\_b) noexcept

{

using \_Tp\_elt = typename \_\_shared\_ptr<\_Tp, \_Lp>::element\_type;

using \_Up\_elt = typename \_\_shared\_ptr<\_Up, \_Lp>::element\_type;

using \_Vp = typename common\_type<\_Tp\_elt\*, \_Up\_elt\*>::type;

// common\_type : 동일한 타입이면 true

// <https://docs.microsoft.com/ko-kr/cpp/standard-library/common-type-class?view=vs-2019>

return less<\_Vp>()(\_\_a.get(), \_\_b.get());

}

template<typename \_Tp, \_Lock\_policy \_Lp>

inline bool operator<(const \_\_shared\_ptr<\_Tp, \_Lp>& \_\_a, nullptr\_t) noexcept

{

using \_Tp\_elt = typename \_\_shared\_ptr<\_Tp, \_Lp>::element\_type;

return less<\_Tp\_elt\*>()(\_\_a.get(), nullptr);

}

template<typename \_Tp, \_Lock\_policy \_Lp>

inline bool operator<(nullptr\_t, const \_\_shared\_ptr<\_Tp, \_Lp>& \_\_a) noexcept

{

using \_Tp\_elt = typename \_\_shared\_ptr<\_Tp, \_Lp>::element\_type;

return less<\_Tp\_elt\*>()(nullptr, \_\_a.get());

}

template<typename \_Tp1, typename \_Tp2, \_Lock\_policy \_Lp>

inline bool operator<=(const \_\_shared\_ptr<\_Tp1, \_Lp>& \_\_a, const \_\_shared\_ptr<\_Tp2, \_Lp>& \_\_b) noexcept

{

return !(\_\_b < \_\_a);

}

templat<typename \_Tp, \_Lock\_policy \_Lp>

inline bool operator<=(const \_\_shared\_ptr<\_Tp, \_Lp>& \_\_a, nullptr\_t) noexcept

{

return !(nullptr < \_\_a);

}

template<typename \_Tp, \_Lock\_policy \_Lp>

inline bool operator<=(nullptr\_t, const \_\_shared\_ptr<\_Tp, \_Lp>& \_\_a) noexcept

{

return !(\_\_a < nullptr);

}

template<typename \_Tp1, typename \_Tp2, \_Lock\_policy \_Lp>

inline bool operator>(const \_\_shared\_ptr<\_Tp1, \_Lp>& \_\_a, const \_\_shared\_ptr<\_Tp2, \_Lp>& \_\_b) noexcept

{

return (\_\_b < \_\_a);

}

template<typename \_Tp, \_Lock\_policy \_Lp>

inline bool operator>(const \_\_shared\_ptr<\_Tp, \_Lp>& \_\_a, nullptr\_t) noexcept

{

return nullptr < \_\_a;

}

template<typename \_Tp, \_Lock\_policy \_Lp>

inline bool operator>(nullptr\_t, const \_\_shared\_ptr<\_Tp, \_Lp>& \_\_a) noexcept

{

return \_\_a < nullptr;

}

template<typename \_Tp1, typename \_Tp2, \_Lock\_policy \_Lp>

inline bool operator>=(const \_\_shared\_ptr<\_Tp1, \_Lp>& \_\_a, const \_\_shared\_ptr<\_Tp2, \_Lp>& \_\_b) noexcept

{

return !(\_\_a < \_\_b);

}

template<typename \_Tp, \_Lock\_policy \_Lp>

inline bool operator>=(const \_\_shared\_ptr<\_Tp, \_Lp>& \_\_a, nullptr\_t) noexcept

{

return !(\_\_a < nullptr);

}

template<typename \_Tp, \_Lock\_policy \_Lp>

inline bool operator>=(nullptr\_t, const \_\_shared\_ptr<\_Tp, \_Lp>& \_\_a) noexcept

{

return !(nullptr < \_\_a);

}

// 20.7.2.2.8 shared\_ptr specialized algorithms.

template<typename \_Tp, \_Lock\_policy \_Lp>

inline void swap(\_\_shared\_ptr<\_Tp, \_Lp>& \_\_a, \_\_shared\_ptr<\_Tp, \_Lp>& \_\_b) noexcept

{

\_\_a.[swap](#shared_ptr_func_swap)(\_\_b);

}

// 20.7.2.2.9 shared\_ptr casts

// The seemingly equivalent code:

// shared\_ptr<\_Tp, \_Lp>(static\_cast<\_Tp\*>(\_\_r.get()))

// will eventually result in undefined behaviour, attempting to

// delete the same object twice.

/// static\_pointer\_cast

template<typename \_Tp, typename \_Tp1, \_Lock\_policy \_Lp>

inline \_\_shared\_ptr<\_Tp, \_Lp> static\_pointer\_cast(const \_\_shared\_ptr<\_Tp1, \_Lp>& \_\_r) noexcept

{

using \_Sp = \_\_shared\_ptr<\_Tp, \_Lp>;

return \_Sp(\_\_r, static\_cast<typename \_Sp::element\_type\*>(\_\_r.get()));

}

// The seemingly equivalent code:

// shared\_ptr<\_Tp, \_Lp>(const\_cast<\_Tp\*>(\_\_r.get()))

// will eventually result in undefined behaviour, attempting to

// delete the same object twice.

/// const\_pointer\_cast

template<typename \_Tp, typename \_Tp1, \_Lock\_policy \_Lp>

inline \_\_shared\_ptr<\_Tp, \_Lp> const\_pointer\_cast(const \_\_shared\_ptr<\_Tp1, \_Lp>& \_\_r) noexcept

{

using \_Sp = \_\_shared\_ptr<\_Tp, \_Lp>;

return \_Sp(\_\_r, const\_cast<typename \_Sp::element\_type\*>(\_\_r.get()));

}

// The seemingly equivalent code:

// shared\_ptr<\_Tp, \_Lp>(dynamic\_cast<\_Tp\*>(\_\_r.get()))

// will eventually result in undefined behaviour, attempting to

// delete the same object twice.

/// dynamic\_pointer\_cast

template<typename \_Tp, typename \_Tp1, \_Lock\_policy \_Lp>

inline \_\_shared\_ptr<\_Tp, \_Lp> dynamic\_pointer\_cast(const \_\_shared\_ptr<\_Tp1, \_Lp>& \_\_r) noexcept

{

using \_Sp = \_\_shared\_ptr<\_Tp, \_Lp>;

if (auto\* \_\_p = dynamic\_cast<typename \_Sp::element\_type\*>(\_\_r.get()))

return \_Sp(\_\_r, \_\_p);

return \_Sp();

}

**[\_\_shared\_count]**

template<\_Lock\_policy \_Lp>

class \_\_shared\_count

{

public:

constexpr \_\_shared\_count() noexcept : \_M\_pi(0)

{ }

// [\_sp\_counter\_base\*] \_M\_pi = 0

template<typename \_Ptr>

explicit \_\_shared\_count(\_Ptr \_\_p) : \_M\_pi(0)

{

\_\_try {

\_M\_pi = new \_Sp\_counted\_ptr<\_Ptr, \_Lp>(\_\_p);

}

\_\_catch(...) {

delete \_\_p;

\_\_throw\_exception\_again;

}

}

// 성공 : [\_Sp\_counted\_base\*] \_M\_pi == \_\_p

// 실패 : [\_Sp\_counted\_base\*] \_M\_pi == 0

// \_\_shared\_count(\_Ptr \_\_p, array-type)

// 두번째 인자 : 배열의 형태 (false\_type : 배열x / true\_type : 배열o)

template<typename \_Ptr>

\_\_shared\_count(\_Ptr \_\_p, false\_type)

: [\_\_shared\_count](#shared_count_ptr)(\_\_p)

{ }

// 성공 : [\_Sp\_counted\_base\*] \_M\_pi == \_\_p

// 실패 : [\_Sp\_counted\_base\*] \_M\_pi == 0

template<typename \_Ptr>

\_\_shared\_count(\_Ptr \_\_p, true\_type)

: [\_\_shared\_count](#shared_count_ptr_deleter_allocator)(\_\_p, \_\_sp\_array\_delete{}, allocator<void>())

{ }

template<typename \_Ptr, typename \_Deleter, typename = typename \_not\_alloc\_shared\_tag<\_Deleter>::type>

\_\_shared\_count(\_Ptr \_\_p, \_Deleter \_\_d)

: [\_\_shared\_count](#shared_count_ptr_deleter_allocator)(\_\_p, std::move(\_\_d), allocator<void>())

{ }

template<typename \_Ptr, typename \_Deleter,

typename \_Alloc, typename = typename \_\_not\_alloc\_shared\_tag<\_Deleter>::type>

\_\_shared\_count(\_Ptr \_\_p, \_Deleter \_\_d, \_Alloc \_\_a) : \_M\_pi(0)

{

typedef \_Sp\_counted\_deleter<\_Ptr, \_Deleter, \_Alloc, \_Lp> \_Sp\_cd\_type;

\_\_try {

typename \_Sp\_cd\_type::\_\_allocator\_type \_\_a2(\_\_a);

auto \_\_guard = std::\_\_allocate\_guarded(\_\_a2);

\_Sp\_cd\_type\* \_\_mem = \_\_guard.get();

::new (\_\_mem) \_Sp\_cd\_type(\_\_p, std::move(\_\_d), std::move(\_\_a));

\_M\_pi = \_\_mem;

\_\_guard = nullptr;

}

\_\_catch(...)

{

\_\_d(\_\_p); // Call \_Deleter on \_\_p.

\_\_throw\_exception\_again;

}

}

// 성공 : [\_Sp\_counted\_base\*] \_M\_pi == ???

// 실패 : [\_Sp\_counted\_base\*] \_M\_pi == 0

template<typename \_Tp, typename \_Alloc, typename... \_Args>

\_\_shared\_count(\_Tp\*& \_\_p, \_Sp\_alloc\_shared\_tag<\_Alloc> \_\_a, \_Args&&... \_\_args)

{

typedef \_Sp\_counted\_ptr\_inplace<\_Tp, \_Alloc, \_Lp> \_Sp\_cp\_type;

typename \_Sp\_cp\_type::\_\_allocator\_type \_\_a2(\_\_a.\_M\_a);

auto \_\_guard = std::\_\_allocate\_guarded(\_\_a2);

\_Sp\_cp\_type\* \_\_mem = \_\_guard.get();

auto \_\_pi = ::new (\_\_mem)

\_Sp\_cp\_type(\_\_a.\_M\_a, std::forward<\_Args>(\_\_args)...);

\_\_guard = nullptr;

\_M\_pi = \_\_pi;

\_\_p = \_\_pi->\_M\_ptr();

}

// unique\_ptr

template<typename \_Tp, typename \_Del>

explicit \_\_shared\_count(std::unique\_ptr<\_Tp, \_Del>&& \_\_r) : \_M\_pi(0)

{

// \_GLIBCXX\_RESOLVE\_LIB\_DEFECTS

// 2415. Inconsistency between unique\_ptr and shared\_ptr

if (\_\_r.get() == nullptr)

return;

using \_Ptr = typename unique\_ptr<\_Tp, \_Del>::pointer;

using \_Del2 = typename conditional<is\_reference<\_Del>::value,

reference\_wrapper<typename remove\_reference<\_Del>::type>,

\_Del>::type;

using \_Sp\_cd\_type

= \_Sp\_counted\_deleter<\_Ptr, \_Del2, allocator<void>, \_Lp>;

using \_Alloc = allocator<\_Sp\_cd\_type>;

using \_Alloc\_traits = allocator\_traits<\_Alloc>;

\_Alloc \_\_a;

\_Sp\_cd\_type\* \_\_mem = \_Alloc\_traits::allocate(\_\_a, 1);

\_Alloc\_traits::construct(\_\_a, \_\_mem, \_\_r.release(),

\_\_r.get\_deleter()); // non-throwing

\_M\_pi = \_\_mem;

}

// Throw bad\_weak\_ptr when \_\_r.\_M\_get\_use\_count() == 0.

explicit \_\_shared\_count(const \_\_weak\_count<\_Lp>& \_\_r);

// Does not throw if \_\_r.\_M\_get\_use\_count() == 0, caller must check.

explicit \_\_shared\_count(const \_\_weak\_count<\_Lp>& \_\_r, std::nothrow\_t);

~\_\_shared\_count() noexcept

{

if (\_M\_pi != nullptr)

\_M\_pi->\_M\_release();

}

\_\_shared\_count(const \_\_shared\_count& \_\_r) noexcept

: \_M\_pi(\_\_r.\_M\_pi)

{

if (\_M\_pi != 0)

\_M\_pi->\_M\_add\_ref\_copy();

}

\_\_shared\_count& operator=(const \_\_shared\_count& \_\_r) noexcept

{

\_Sp\_counted\_base<\_Lp>\* \_\_tmp = \_\_r.\_M\_pi;

if (\_\_tmp != \_M\_pi) {

if (\_\_tmp != 0)

\_\_tmp->\_M\_add\_ref\_copy();

if (\_M\_pi != 0)

\_M\_pi->\_M\_release();

\_M\_pi = \_\_tmp;

}

return \*this;

}

void \_M\_swap(\_\_shared\_count& \_\_r) noexcept

{

\_Sp\_counted\_base<\_Lp>\* \_\_tmp = \_\_r.\_M\_pi;

\_\_r.\_M\_pi = \_M\_pi;

\_M\_pi = \_\_tmp;

}

long \_M\_get\_use\_count() const noexcept

{

return \_M\_pi != 0 ? \_M\_pi->\_M\_get\_use\_count() : 0;

}

// unique\_ptr 이면 return true

bool \_M\_unique() const noexcept

{

return this->\_M\_get\_use\_count() == 1;

}

// \_M\_pi가 nullptr이 아니면 return deleter

void\* \_M\_get\_deleter(const std::type\_info& \_\_ti) const noexcept

{

return \_M\_pi ? \_M\_pi->[\_M\_get\_deleter(\_\_ti)](#Sp_counted_base__M_get_deleter) : nullptr;

}

// less : <https://ko.cppreference.com/w/cpp/utility/functional/less>

bool \_M\_less(const \_\_shared\_count& \_\_rhs) const noexcept

{

return std::less<\_Sp\_counted\_base<\_Lp>\*>()(this->\_M\_pi, \_\_rhs.\_M\_pi);

}

bool \_M\_less(const \_\_weak\_count<\_Lp>& \_\_rhs) const noexcept

{

return std::less<\_Sp\_counted\_base<\_Lp>\*>()(this->\_M\_pi, \_\_rhs.\_M\_pi);

}

// 동일한 shared\_count라면 return true

// Friend function injected into enclosing namespace and found by ADL

friend inline bool operator==(const \_\_shared\_count& \_\_a, const \_\_shared\_count& \_\_b) noexcept

{

return \_\_a.\_M\_pi == \_\_b.\_M\_pi;

}

};

**[\_\_Sp\_counted\_base]**

template<\_Lock\_policy \_Lp = \_\_default\_lock\_policy>

class \_Sp\_counted\_base : public \_Mutex\_base<\_Lp>

{

public:

\_Sp\_counted\_base() noexcept : \_M\_use\_count(1), \_M\_weak\_count(1) { }

// 생성 초기값 \_M\_use\_count = 1 && \_M\_weak\_count = 1

virtual ~\_Sp\_counted\_base() noexcept { }

// 소멸자

virtual void \_M\_dispose() noexcept = 0;

virtual void \_M\_destroy() noexcept { delete this; }

virtual void\* \_M\_get\_deleter(const std::type\_info&) noexcept = 0;

void \_M\_add\_ref\_copy() // single은 따로 정의

{

// 복사를 위해서 \_M\_use\_cont 증가

\_\_gnu\_cxx::\_\_atomic\_add\_dispatch(&\_M\_use\_count, 1);

// \_\_atomic\_add\_dispatch : 두번째 인수의 값을 첫번째 인수에 더함.

// 설명 <https://gcc.gnu.org/onlinedocs/libstdc++/manual/ext_concurrency.html>

}

void \_M\_add\_ref\_lock(); // 전역으로 정의 (single, mutex, atomic)

bool \_M\_add\_ref\_lock\_nothrow(); // 전역으로 정의 (single, mutex, atomic)

void \_M\_release() noexcept // single은 따로 정의

{

\_GLIBCXX\_SYNCHRONIZATION\_HAPPENS\_BEFORE(&\_M\_use\_count);

if (\_\_gnu\_cxx::\_\_exchange\_and\_add\_dispatch(&\_M\_use\_count, -1) == 1) {

// 반환값이 1일 때, (이전의 값을 반환)

\_GLIBCXX\_SYNCHRONIZATION\_HAPPENS\_AFTER(&\_M\_use\_count);

\_M\_dispose();

// destroy ()를 실행하는 스레드에서 dispose ()의 영향을 확인하려면,

// dispose ()와 destroy () 사이에 메모리 장벽이 있어야함.

if (\_Mutex\_base<\_Lp>::\_S\_need\_barriers)

\_\_atomic\_thread\_fence(\_\_ATOMIC\_ACQ\_REL);

\_GLIBCXX\_SYNCHRONIZATION\_HAPPENS\_BEFORE(&\_M\_weak\_count);

if (\_\_gnu\_cxx::\_\_exchange\_and\_add\_dispatch(&\_M\_weak\_count, -1) == 1) {

\_GLIBCXX\_SYNCHRONIZATION\_HAPPENS\_AFTER(&\_M\_weak\_count);

\_M\_destroy();

}

}

}

void \_M\_weak\_add\_ref() noexcept // single은 따로 정의

{

\_\_gnu\_cxx::\_\_atomic\_add\_dispatch(&\_M\_weak\_count, 1);

}

void \_M\_weak\_release() noexcept // single은 따로 정의

{

\_GLIBCXX\_SYNCHRONIZATION\_HAPPENS\_BEFORE(&\_M\_weak\_count);

if (\_\_gnu\_cxx::\_\_exchange\_and\_add\_dispatch(&\_M\_weak\_count, -1) == 1) {

\_GLIBCXX\_SYNCHRONIZATION\_HAPPENS\_AFTER(&\_M\_weak\_count);

if (\_Mutex\_base<\_Lp>::\_S\_need\_barriers)

\_\_atomic\_thread\_fence(\_\_ATOMIC\_ACQ\_REL);

\_M\_destroy();

}

}

long \_M\_get\_use\_count() const noexcept // single은 따로 정의

{

// No memory barrier is used here so there is no synchronization

// with other threads.

return \_\_atomic\_load\_n(&\_M\_use\_count, \_\_ATOMIC\_RELAXED);

// 동기화하지 않고 ref 리턴.

}

};

template<>

inline void \_Sp\_counted\_base<\_S\_single>:: M\_add\_ref\_lock()

{

if (\_M\_use\_count == 0)

\_\_throw\_bad\_weak\_ptr();

++\_M\_use\_count;

}

template<>

inline void \_Sp\_counted\_base<\_S\_mutex>:: \_M\_add\_ref\_lock()

{

\_\_gnu\_cxx::\_\_scoped\_lock sentry(\*this);

if (\_\_gnu\_cxx::\_\_exchange\_and\_add\_dispatch(&\_M\_use\_count, 1) == 0)

{

\_M\_use\_count = 0;

\_\_throw\_bad\_weak\_ptr();

}

}

template<>

inline void \_Sp\_counted\_base<\_S\_atomic>:: \_M\_add\_ref\_lock()

{

\_Atomic\_word \_\_count = \_M\_get\_use\_count();

do {

if (\_\_count == 0) \_\_throw\_bad\_weak\_ptr();

} while (!\_\_atomic\_compare\_exchange\_n(&\_M\_use\_count, &\_\_count, \_\_count + 1,

true, \_\_ATOMIC\_ACQ\_REL, \_\_ATOMIC\_RELAXED));

}

template<>

inline bool \_Sp\_counted\_base<\_S\_single>:: \_M\_add\_ref\_lock\_nothrow()

{

if (\_M\_use\_count == 0) return false;

++\_M\_use\_count;

return true;

}

template<>

inline bool \_Sp\_counted\_base<\_S\_mutex>:: \_M\_add\_ref\_lock\_nothrow()

{

\_\_gnu\_cxx::\_\_scoped\_lock sentry(\*this);

if (\_\_gnu\_cxx::\_\_exchange\_and\_add\_dispatch(&\_M\_use\_count, 1) == 0) {

\_M\_use\_count = 0;

return false;

}

return true;

}

template<>

inline bool \_Sp\_counted\_base<\_S\_atomic>:: \_M\_add\_ref\_lock\_nothrow()

{

\_Atomic\_word \_\_count = \_M\_get\_use\_count();

do {

if (\_\_count == 0) return false;

} while (!\_\_atomic\_compare\_exchange\_n(&\_M\_use\_count, &\_\_count, \_\_count + 1,

true, \_\_ATOMIC\_ACQ\_REL, \_\_ATOMIC\_RELAXED));

return true;

}

template<>

inline void \_Sp\_counted\_base<\_S\_single>::\_M\_add\_ref\_copy() {

++\_M\_use\_count;

}

template<>

inline void \_Sp\_counted\_base<\_S\_single>::\_M\_release() noexcept

{

if (--\_M\_use\_count == 0) {

\_M\_dispose();

if (--\_M\_weak\_count == 0)

\_M\_destroy();

}

}

template<>

inline void \_Sp\_counted\_base<\_S\_single>::\_M\_weak\_add\_ref() noexcept

{

++\_M\_weak\_count;

}

template<>

inline void \_Sp\_counted\_base<\_S\_single>::\_M\_weak\_release() noexcept

{

if (--\_M\_weak\_count == 0)

\_M\_destroy();

}

template<>

inline long \_Sp\_counted\_base<\_S\_single>::\_M\_get\_use\_count() const noexcept

{

return \_M\_use\_count;

}

**[\_Mutex\_base]**

template<\_Lock\_policy \_Lp>

class \_Mutex\_base

{

protected:

// The atomic policy uses fully-fenced builtins, single doesn't care.

enum { \_S\_need\_barriers = 0 };

};

**[\_Mutex\_base<\_S\_mutex>]**

template<>

class \_Mutex\_base<\_S\_mutex> : public \_\_gnu\_cxx::\_\_mutex

{

protected:

// This policy is used when atomic builtins are not available.

// The replacement atomic operations might not have the necessary

// memory barriers.

enum { \_S\_need\_barriers = 1 };

};

**---------------------------<shared\_ptr.h>---------------------------**

**[shared\_ptr]**

template<typename \_Tp>

class shared\_ptr : public \_\_shared\_ptr<\_Tp>

{

public:

/\*\*

\* @brief Construct an empty %shared\_ptr.

\* @post use\_count()==0 && get()==0

\*/

constexpr shared\_ptr() noexcept

: [\_\_shared\_ptr<\_Tp>](#shared_ptr_)()

{ }

shared\_ptr(const shared\_ptr&) noexcept = default; ///< Copy constructor

/\*\*

\* @brief Construct a %shared\_ptr that owns the pointer @a \_\_p.

\* @param \_\_p A pointer that is convertible to element\_type\*.

\* @post use\_count() == 1 && get() == \_\_p

\* @throw std::bad\_alloc, in which case @c delete @a \_\_p is called.

\*/

template<typename \_Yp, typename = \_Constructible<\_Yp\*>>

explicit shared\_ptr(\_Yp\* \_\_p)

: [\_\_shared\_ptr<\_Tp>](#shared_ptr_ptr)(\_\_p)

{ }

/\*\*

\* @brief Construct a %shared\_ptr that owns the pointer @a \_\_p

\* and the deleter @a \_\_d.

\* @param \_\_p A pointer.

\* @param \_\_d A deleter.

\* @post use\_count() == 1 && get() == \_\_p

\* @throw std::bad\_alloc, in which case @a \_\_d(\_\_p) is called.

\*

\* Requirements: \_Deleter's copy constructor and destructor must

\* not throw

\*

\* \_\_shared\_ptr will release \_\_p by calling \_\_d(\_\_p)

\*/

template<typename \_Yp, typename \_Deleter, typename = \_Constructible<\_Yp\*, \_Deleter>>

shared\_ptr(\_Yp\* \_\_p, \_Deleter \_\_d)

: [\_\_shared\_ptr<\_Tp>](#shared_ptr_ptr_deleter)(\_\_p, std::move(\_\_d))

{ }

/\*\*

\* @brief Construct a %shared\_ptr that owns a null pointer

\* and the deleter @a \_\_d.

\* @param \_\_p A null pointer constant.

\* @param \_\_d A deleter.

\* @post use\_count() == 1 && get() == \_\_p

\* @throw std::bad\_alloc, in which case @a \_\_d(\_\_p) is called.

\*

\* Requirements: \_Deleter's copy constructor and destructor must

\* not throw

\*

\* The last owner will call \_\_d(\_\_p)

\*/

template<typename \_Deleter>

shared\_ptr(nullptr\_t \_\_p, \_Deleter \_\_d)

: [\_\_shared\_ptr<\_Tp>](#shared_ptr_ptr_deleter)(\_\_p, std::move(\_\_d))

{ }

/\*\*

\* @brief Construct a %shared\_ptr that owns the pointer @a \_\_p

\* and the deleter @a \_\_d.

\* @param \_\_p A pointer.

\* @param \_\_d A deleter.

\* @param \_\_a An allocator.

\* @post use\_count() == 1 && get() == \_\_p

\* @throw std::bad\_alloc, in which case @a \_\_d(\_\_p) is called.

\*

\* Requirements: \_Deleter's copy constructor and destructor must

\* not throw \_Alloc's copy constructor and destructor must not

\* throw.

\*

\* \_\_shared\_ptr will release \_\_p by calling \_\_d(\_\_p)

\*/

template<typename \_Yp, typename \_Deleter, typename \_Alloc,

typename = \_Constructible<\_Yp\*, \_Deleter, \_Alloc>>

shared\_ptr(\_Yp\* \_\_p, \_Deleter \_\_d, \_Alloc \_\_a)

: [\_\_shared\_ptr<\_Tp>](#shared_ptr_ptr_deleter_allocator)(\_\_p, std::move(\_\_d), std::move(\_\_a))

{ }

/\*\*

\* @brief Construct a %shared\_ptr that owns a null pointer

\* and the deleter @a \_\_d.

\* @param \_\_p A null pointer constant.

\* @param \_\_d A deleter.

\* @param \_\_a An allocator.

\* @post use\_count() == 1 && get() == \_\_p

\* @throw std::bad\_alloc, in which case @a \_\_d(\_\_p) is called.

\*

\* Requirements: \_Deleter's copy constructor and destructor must

\* not throw \_Alloc's copy constructor and destructor must not

\* throw.

\*

\* The last owner will call \_\_d(\_\_p)

\*/

template<typename \_Deleter, typename \_Alloc>

shared\_ptr(nullptr\_t \_\_p, \_Deleter \_\_d, \_Alloc \_\_a)

: [\_\_shared\_ptr<\_Tp>](#shared_ptr_ptr_deleter_allocator)(\_\_p, std::move(\_\_d), std::move(\_\_a))

{ }

// Aliasing constructor

/\*\*

\* @brief Constructs a `shared\_ptr` instance that stores `\_\_p`

\* and shares ownership with `\_\_r`.

\* @param \_\_r A `shared\_ptr`.

\* @param \_\_p A pointer that will remain valid while `\*\_\_r` is valid.

\* @post `get() == \_\_p && use\_count() == \_\_r.use\_count()`

\*

\* This can be used to construct a `shared\_ptr` to a sub-object

\* of an object managed by an existing `shared\_ptr`. The complete

\* object will remain valid while any `shared\_ptr` owns it, even

\* if they don't store a pointer to the complete object.

\*

\* @code

\* shared\_ptr<pair<int,int>> pii(new pair<int,int>());

\* shared\_ptr<int> pi(pii, &pii->first);

\* assert(pii.use\_count() == 2);

\* @endcode

\*/

template<typename \_Yp>

shared\_ptr(const shared\_ptr<\_Yp>& \_\_r, element\_type\* \_\_p) noexcept

: [\_\_shared\_ptr<\_Tp>](#shared_ptr_ptr_elementtype)(\_\_r, \_\_p)

{ }

/\*\*

\* @brief If @a \_\_r is empty, constructs an empty %shared\_ptr;

\* otherwise construct a %shared\_ptr that shares ownership

\* with @a \_\_r.

\* @param \_\_r A %shared\_ptr.

\* @post get() == \_\_r.get() && use\_count() == \_\_r.use\_count()

\*/

template<typename \_Yp, typename = \_Constructible<const shared\_ptr<\_Yp>&>>

shared\_ptr(const shared\_ptr<\_Yp>& \_\_r) noexcept

: [\_\_shared\_ptr<\_Tp>](#shared_ptr_ptr)(\_\_r)

{ }

/\*\*

\* @brief Move-constructs a %shared\_ptr instance from @a \_\_r.

\* @param \_\_r A %shared\_ptr rvalue.

\* @post \*this contains the old value of @a \_\_r, @a \_\_r is empty.

\*/

shared\_ptr(shared\_ptr&& \_\_r) noexcept

: [\_\_shared\_ptr<\_Tp>](#shared_ptr_move_r)(std::move(\_\_r))

{ }

/\*\*

\* @brief Move-constructs a %shared\_ptr instance from @a \_\_r.

\* @param \_\_r A %shared\_ptr rvalue.

\* @post \*this contains the old value of @a \_\_r, @a \_\_r is empty.

\*/

template<typename \_Yp, typename = \_Constructible<shared\_ptr<\_Yp>>>

shared\_ptr(shared\_ptr<\_Yp>&& \_\_r) noexcept

: [\_\_shared\_ptr<\_Tp>](#shared_ptr_move_r)(std::move(\_\_r))

{ }

/\*\*

\* @brief Constructs a %shared\_ptr that shares ownership with @a \_\_r

\* and stores a copy of the pointer stored in @a \_\_r.

\* @param \_\_r A weak\_ptr.

\* @post use\_count() == \_\_r.use\_count()

\* @throw bad\_weak\_ptr when \_\_r.expired(),

\* in which case the constructor has no effect.

\*/

template<typename \_Yp, typename = \_Constructible<const weak\_ptr<\_Yp>&>>

explicit shared\_ptr(const weak\_ptr<\_Yp>& \_\_r)

: [\_\_shared\_ptr<\_Tp>](#shared_ptr_move_r)(\_\_r)

{ }

template<typename \_Yp, typename \_Del, typename = \_Constructible<unique\_ptr<\_Yp, \_Del>>>

shared\_ptr(unique\_ptr<\_Yp, \_Del>&& \_\_r)

: [\_\_shared\_ptr<\_Tp>](#shared_ptr_move_r)(std::move(\_\_r))

{ }

/\*\*

\* @brief Construct an empty %shared\_ptr.

\* @post use\_count() == 0 && get() == nullptr

\*/

constexpr shared\_ptr(nullptr\_t) noexcept

: [shared\_ptr](#s_shared_ptr_)()

{ }

shared\_ptr& operator=(const shared\_ptr&) noexcept = default;

template<typename \_Yp>

\_Assignable<const shared\_ptr<\_Yp>&> operator=(const shared\_ptr<\_Yp>& \_\_r) noexcept

{

this->\_\_shared\_ptr<\_Tp>::operator=(\_\_r);

return \*this;

}

shared\_ptr& operator=(shared\_ptr&& \_\_r) noexcept

{

this->\_\_shared\_ptr<\_Tp>::operator=(std::move(\_\_r));

return \*this;

}

template<class \_Yp>

\_Assignable<shared\_ptr<\_Yp>> operator=(shared\_ptr<\_Yp>&& \_\_r) noexcept

{

this->\_\_shared\_ptr<\_Tp>::operator=(std::move(\_\_r));

return \*this;

}

template<typename \_Yp, typename \_Del>

\_Assignable<unique\_ptr<\_Yp, \_Del>> operator=(unique\_ptr<\_Yp, \_Del>&& \_\_r)

{

this->\_\_shared\_ptr<\_Tp>::operator=(std::move(\_\_r));

return \*this;

}

private:

// This constructor is non-standard, it is used by allocate\_shared.

template<typename \_Alloc, typename... \_Args>

shared\_ptr(\_Sp\_alloc\_shared\_tag<\_Alloc> \_\_tag, \_Args&&... \_\_args)

: \_\_shared\_ptr<\_Tp>(\_\_tag, std::forward<\_Args>(\_\_args)...)

{ }

template<typename \_Yp, typename \_Alloc, typename... \_Args>

friend shared\_ptr<\_Yp>

allocate\_shared(const \_Alloc& \_\_a, \_Args&&... \_\_args);

};

/// Equality operator for shared\_ptr objects, compares the stored pointers

template<typename \_Tp, typename \_Up>

\_GLIBCXX\_NODISCARD inline bool operator==(const shared\_ptr<\_Tp>& \_\_a, const shared\_ptr<\_Up>& \_\_b) noexcept

{

return \_\_a.get() == \_\_b.get();

}

/// shared\_ptr comparison with nullptr

template<typename \_Tp>

\_GLIBCXX\_NODISCARD inline bool operator==(const shared\_ptr<\_Tp>& \_\_a, nullptr\_t) noexcept

{

return !\_\_a;

}

/// shared\_ptr comparison with nullptr

template<typename \_Tp>

\_GLIBCXX\_NODISCARD inline bool operator==(nullptr\_t, const shared\_ptr<\_Tp>& \_\_a) noexcept

{

return !\_\_a;

}

/// Inequality operator for shared\_ptr objects, compares the stored pointers

template<typename \_Tp, typename \_Up>

\_GLIBCXX\_NODISCARD inline bool operator!=(const shared\_ptr<\_Tp>& \_\_a, const shared\_ptr<\_Up>& \_\_b) noexcept

{

return \_\_a.get() != \_\_b.get();

}

/// shared\_ptr comparison with nullptr

template<typename \_Tp>

\_GLIBCXX\_NODISCARD inline bool operator!=(const shared\_ptr<\_Tp>& \_\_a, nullptr\_t) noexcept

{

return (bool)\_\_a;

}

/// shared\_ptr comparison with nullptr

template<typename \_Tp>

\_GLIBCXX\_NODISCARD inline bool operator!=(nullptr\_t, const shared\_ptr<\_Tp>& \_\_a) noexcept

{

return (bool)\_\_a;

}

/// Relational operator for shared\_ptr objects, compares the stored pointers

template<typename \_Tp, typename \_Up>

\_GLIBCXX\_NODISCARD inline bool operator<(const shared\_ptr<\_Tp>& \_\_a, const shared\_ptr<\_Up>& \_\_b) noexcept

{

using \_Tp\_elt = typename shared\_ptr<\_Tp>::element\_type;

using \_Up\_elt = typename shared\_ptr<\_Up>::element\_type;

using \_Vp = typename common\_type<\_Tp\_elt\*, \_Up\_elt\*>::type;

return less<\_Vp>()(\_\_a.get(), \_\_b.get());

}

/// shared\_ptr comparison with nullptr

template<typename \_Tp>

\_GLIBCXX\_NODISCARD inline bool operator<(const shared\_ptr<\_Tp>& \_\_a, nullptr\_t) noexcept

{

using \_Tp\_elt = typename shared\_ptr<\_Tp>::element\_type;

return less<\_Tp\_elt\*>()(\_\_a.get(), nullptr);

}

/// shared\_ptr comparison with nullptr

template<typename \_Tp>

\_GLIBCXX\_NODISCARD inline bool operator<(nullptr\_t, const shared\_ptr<\_Tp>& \_\_a) noexcept

{

using \_Tp\_elt = typename shared\_ptr<\_Tp>::element\_type;

return less<\_Tp\_elt\*>()(nullptr, \_\_a.get());

}

/// Relational operator for shared\_ptr objects, compares the stored pointers

template<typename \_Tp, typename \_Up>

\_GLIBCXX\_NODISCARD inline bool operator<=(const shared\_ptr<\_Tp>& \_\_a, const shared\_ptr<\_Up>& \_\_b) noexcept

{

return !(\_\_b < \_\_a);

}

/// shared\_ptr comparison with nullptr

template<typename \_Tp>

\_GLIBCXX\_NODISCARD inline bool operator<=(const shared\_ptr<\_Tp>& \_\_a, nullptr\_t) noexcept

{

return !(nullptr < \_\_a);

}

/// shared\_ptr comparison with nullptr

template<typename \_Tp>

\_GLIBCXX\_NODISCARD inline bool operator<=(nullptr\_t, const shared\_ptr<\_Tp>& \_\_a) noexcept

{

return !(\_\_a < nullptr);

}

/// Relational operator for shared\_ptr objects, compares the stored pointers

template<typename \_Tp, typename \_Up>

\_GLIBCXX\_NODISCARD inline bool operator>(const shared\_ptr<\_Tp>& \_\_a, const shared\_ptr<\_Up>& \_\_b) noexcept

{

return (\_\_b < \_\_a);

}

/// shared\_ptr comparison with nullptr

template<typename \_Tp>

\_GLIBCXX\_NODISCARD inline bool operator>(const shared\_ptr<\_Tp>& \_\_a, nullptr\_t) noexcept

{

return nullptr < \_\_a;

}

/// shared\_ptr comparison with nullptr

template<typename \_Tp>

\_GLIBCXX\_NODISCARD inline bool operator>(nullptr\_t, const shared\_ptr<\_Tp>& \_\_a) noexcept

{

return \_\_a < nullptr;

}

/// Relational operator for shared\_ptr objects, compares the stored pointers

template<typename \_Tp, typename \_Up>

\_GLIBCXX\_NODISCARD inline bool operator>=(const shared\_ptr<\_Tp>& \_\_a, const shared\_ptr<\_Up>& \_\_b) noexcept

{

return !(\_\_a < \_\_b);

}

/// shared\_ptr comparison with nullptr

template<typename \_Tp>

\_GLIBCXX\_NODISCARD inline bool operator>=(const shared\_ptr<\_Tp>& \_\_a, nullptr\_t) noexcept

{

return !(\_\_a < nullptr);

}

/// shared\_ptr comparison with nullptr

template<typename \_Tp>

\_GLIBCXX\_NODISCARD inline bool operator>=(nullptr\_t, const shared\_ptr<\_Tp>& \_\_a) noexcept

{

return !(nullptr < \_\_a);

}

/// Swap overload for shared\_ptr

template<typename \_Tp>

inline void swap(shared\_ptr<\_Tp>& \_\_a, shared\_ptr<\_Tp>& \_\_b) noexcept

{

\_\_a.swap(\_\_b);

}

/// Convert type of `shared\_ptr`, via `static\_cast`

template<typename \_Tp, typename \_Up>

inline shared\_ptr<\_Tp> static\_pointer\_cast(const shared\_ptr<\_Up>& \_\_r) noexcept

{

using \_Sp = shared\_ptr<\_Tp>;

return \_Sp(\_\_r, static\_cast<typename \_Sp::element\_type\*>(\_\_r.get()));

}

/// Convert type of `shared\_ptr`, via `const\_cast`

template<typename \_Tp, typename \_Up>

inline shared\_ptr<\_Tp> const\_pointer\_cast(const shared\_ptr<\_Up>& \_\_r) noexcept

{

using \_Sp = shared\_ptr<\_Tp>;

return \_Sp(\_\_r, const\_cast<typename \_Sp::element\_type\*>(\_\_r.get()));

}

/// Convert type of `shared\_ptr`, via `dynamic\_cast`

template<typename \_Tp, typename \_Up>

inline shared\_ptr<\_Tp> dynamic\_pointer\_cast(const shared\_ptr<\_Up>& \_\_r) noexcept

{

using \_Sp = shared\_ptr<\_Tp>;

if (auto\* \_\_p = dynamic\_cast<typename \_Sp::element\_type\*>(\_\_r.get()))

return \_Sp(\_\_r, \_\_p);

return \_Sp();

}

|  |  |  |  |
| --- | --- | --- | --- |
| **문제점 정리** |  | **해결 방안** |  |
| **다음 주차** | **4 주차** | **다음 기간** | **2020.02.03~2020.02.09** |
| **다음주 할 일** | Weak\_ptr | | |
| **지도교수**  **Comment** |  | | |