|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | **작성자** | **2014132002 구태균** | **팀명** | - |
| **주차** | **4 주차** | **기간** | **2020.02.03~2020.02.09** | **지도교수** | **정 내 훈** (서명) |
| **이번주 한일** | Weak\_ptr | | | | |

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

**[\_\_weak\_count]**

template<\_Lock\_policy \_Lp>

class \_\_weak\_count

{

public:

constexpr \_\_weak\_count() noexcept

: \_M\_pi(nullptr)

{ }

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

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

{

if (\_M\_pi != nullptr) \_M\_pi->\_M\_weak\_add\_ref();

}

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

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

{

if (\_M\_pi != nullptr) \_M\_pi->\_M\_weak\_add\_ref();

}

\_\_weak\_count(\_\_weak\_count&& \_\_r) noexcept

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

{

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

}

~\_\_weak\_count() noexcept

{

if (\_M\_pi != nullptr) \_M\_pi->\_M\_weak\_release();

}

\_\_weak\_count& operator=(const \_\_shared\_count<\_Lp>& \_\_r) noexcept

{

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

if (\_\_tmp != nullptr) \_\_tmp->\_M\_weak\_add\_ref();

if (\_M\_pi != nullptr) \_M\_pi->\_M\_weak\_release();

\_M\_pi = \_\_tmp;

return \*this;

}

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

{

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

if (\_\_tmp != nullptr) \_\_tmp->\_M\_weak\_add\_ref();

if (\_M\_pi != nullptr) \_M\_pi->\_M\_weak\_release();

\_M\_pi = \_\_tmp;

return \*this;

}

\_\_weak\_count& operator=(\_\_weak\_count&& \_\_r) noexcept

{

if (\_M\_pi != nullptr) \_M\_pi->\_M\_weak\_release();

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

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

return \*this;

}

void \_M\_swap(\_\_weak\_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 != nullptr ? \_M\_pi->\_M\_get\_use\_count() : 0;

}

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

{

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

}

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

{

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

}

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

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

{

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

}

private:

friend class \_\_shared\_count<\_Lp>;

\_Sp\_counted\_base<\_Lp>\* \_M\_pi;

};

**[\_\_weak\_ptr]**

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

class \_\_weak\_ptr

{

template<typename \_Yp, typename \_Res = void>

using \_Compatible = typename enable\_if<\_\_sp\_compatible\_with<\_Yp\*, \_Tp\*>::value, \_Res>::type;

// weak\_ptr 구성가능 여부

template<typename \_Yp>

using \_Assignable = \_Compatible<\_Yp, \_\_weak\_ptr&>;

// weak\_ptr 할당가능 여부

public:

using element\_type = typename remove\_extent<\_Tp>::type;

// (using a = b) == (typedef b a) template에서 사용하기 편리.

// remove\_extent<T>::type : 원소의 타입을 리턴 (배열이여도 원래 원소타입 리턴, 템플릿::멤버이름)

// \_Tp로 배열이 들어와도 원래의 타입으로 element\_type

constexpr \_\_weak\_ptr() noexcept

: \_M\_ptr(nullptr), \_M\_refcount()

{ }

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

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

// The "obvious" converting constructor implementation:

//

// template<typename \_Tp1>

// \_\_weak\_ptr(const \_\_weak\_ptr<\_Tp1, \_Lp>& \_\_r)

// : \_M\_ptr(\_\_r.\_M\_ptr), \_M\_refcount(\_\_r.\_M\_refcount) // never throws

// { }

//

// has a serious problem.

//

// \_\_r.\_M\_ptr may already have been invalidated. The \_M\_ptr(\_\_r.\_M\_ptr)

// conversion may require access to \*\_\_r.\_M\_ptr (virtual inheritance).

//

// It is not possible to avoid spurious access violations since

// in multithreaded programs \_\_r.\_M\_ptr may be invalidated at any point.

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

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

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

{

\_M\_ptr = \_\_r.lock().get();

}

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

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

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

{ }

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

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

{

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

}

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

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

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

{

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

}

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

template<typename \_Yp>

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

{

\_M\_ptr = \_\_r.lock().get();

\_M\_refcount = \_\_r.\_M\_refcount;

return \*this;

}

template<typename \_Yp>

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

{

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

\_M\_refcount = \_\_r.\_M\_refcount;

return \*this;

}

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

{

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

\_M\_refcount = std::move(\_\_r.\_M\_refcount);

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

return \*this;

}

template<typename \_Yp>

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

{

\_M\_ptr = \_\_r.lock().get();

\_M\_refcount = std::move(\_\_r.\_M\_refcount);

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

return \*this;

}

\_\_shared\_ptr<\_Tp, \_Lp> lock() const noexcept

{

return \_\_shared\_ptr<element\_type, \_Lp>(\*this, std::nothrow);

}

long use\_count() const noexcept

{

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

}

bool expired() const noexcept

{

return \_M\_refcount.\_M\_get\_use\_count() == 0;

}

template<typename \_Tp1>

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

{

return \_M\_refcount.\_M\_less(\_\_rhs.\_M\_refcount);

}

template<typename \_Tp1>

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

{

return \_M\_refcount.\_M\_less(\_\_rhs.\_M\_refcount);

}

void reset() noexcept

{

\_\_weak\_ptr().swap(\*this);

}

void swap(\_\_weak\_ptr& \_\_s) noexcept

{

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

\_M\_refcount.\_M\_swap(\_\_s.\_M\_refcount);

}

private:

// Used by \_\_enable\_shared\_from\_this.

void \_M\_assign(\_Tp\* \_\_ptr, const \_\_shared\_count<\_Lp>& \_\_refcount) noexcept

{

if (use\_count() == 0) {

\_M\_ptr = \_\_ptr;

\_M\_refcount = \_\_refcount;

}

}

template<typename \_Tp1, \_Lock\_policy \_Lp1> friend class \_\_shared\_ptr;

template<typename \_Tp1, \_Lock\_policy \_Lp1> friend class \_\_weak\_ptr;

friend class \_\_enable\_shared\_from\_this<\_Tp, \_Lp>;

friend class enable\_shared\_from\_this<\_Tp>;

element\_type\* \_M\_ptr; // Contained pointer.

\_\_weak\_count<\_Lp> \_M\_refcount; // Reference counter.

};

// 20.7.2.3.6 weak\_ptr specialized algorithms.

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

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

{

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

}

|  |  |  |  |
| --- | --- | --- | --- |
| **문제점 정리** | \_Sp\_counted\_deleter  \_Sp\_counted\_ptr\_inplace  \_\_shared\_ptr\_access | **해결 방안** |  |
| **다음 주차** | **5 주차** | **다음 기간** | **2020.02.10~2020.02.16** |
| **다음주 할 일** |  | | |
| **지도교수**  **Comment** |  | | |