|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | **작성자** | **2014132002 구태균** | **팀명** | - |
| **주차** | **5 주차** | **기간** | **2020.02.10~2020.02.16** | **지도교수** | **정 내 훈** (서명) |
| **이번주 한일** | GCC 코드 변경   * 확장성 & 호환성 제외 * 예외처리 제외 | | | | |

**[shared\_ptr\_base.h]**

#include <typeinfo>

#include <bits/allocated\_ptr.h>

#include <bits/refwrap.h>

#include <bits/stl\_function.h>

#include <ext/aligned\_buffer.h>

* **호환성 제외**

--------------------------------------------

namespace std \_GLIBCXX\_VISIBILITY(default)

{

\_GLIBCXX\_BEGIN\_NAMESPACE\_VERSION

…

* **호환성**

**namespace std**

**{**

**…**

--------------------------------------------

#if \_GLIBCXX\_USE\_DEPRECATED

#pragma GCC diagnostic push

#pragma GCC diagnostic ignored "-Wdeprecated-declarations"

template<typename> class auto\_ptr;

#pragma GCC diagnostic pop

#endif

* **호환성 제외**

--------------------------------------------

class bad\_weak\_ptr

inline void \_\_throw\_bad\_weak\_ptr()

* **확장성 제외**

--------------------------------------------

using \_\_gnu\_cxx::\_Lock\_policy;

using \_\_gnu\_cxx::\_\_default\_lock\_policy;

using \_\_gnu\_cxx::\_S\_single;

using \_\_gnu\_cxx::\_S\_mutex;

using \_\_gnu\_cxx::\_S\_atomic;

* **호환성 확장성 제외**
* **\_S\_atomic 이용**

--------------------------------------------

class \_Mutex\_base

* **확장성 제외**

--------------------------------------------

[ class \_Sp\_counted\_base ]

void \_M\_add\_ref\_copy()

{

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

* **호환성**

**\_M\_use\_count++;**

}

-------------------

void \_M\_release() noexcept

{

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

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

{

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

\_M\_dispose();

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();

}

}

* **호환성**

**if ( 1 == \_M\_use\_count--) {**

**\_M\_dispose();**

**atomic\_thread\_fence(memory\_order\_acquire);**

**if (1 == \_M\_weak\_count--) {**

**\_M\_destroy();**

**}**

**}**

}

-------------------

void \_M\_weak\_add\_ref() noexcept

{

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

* **호환성**

**\_M\_weak\_count++;**

}

-------------------

void \_M\_weak\_release() noexcept

{

\_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();

}

* **호환성**

**if ( 1 == \_M\_use\_count--)**

**{**

**\_M\_dispose();**

**atomic\_thread\_fence(memory\_order\_acquire);**

**if (1 == \_M\_weak\_count--)**

**{**

**\_M\_destroy();**

**}**

**}**

}

-------------------

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

{

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

* **호환성**

**return \_M\_use\_count.load(memory\_order\_relaxed);**

}

-------------------

\_Atomic\_word \_M\_use\_count;

\_Atomic\_word \_M\_weak\_count;

* **호환성**

**#include <atomic>**

**atomic\_int \_M\_use\_count;**

**atomic\_int \_M\_weak\_count;**

--------------------------------------------

template<>

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

template<>

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

template<>

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

template<>

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

template<>

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

template<>

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

template<>

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

template<>

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

template<>

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

* **확장성 제외**

--------------------------------------------

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));

* **호환성**

**int \_\_count = \_M\_get\_use\_count();**

**do {**

**if (\_\_count == 0)**

**bad\_weak\_ptr\_exception();**

**} while (!atomic\_compare\_exchange\_strong(&\_M\_use\_count, &\_\_count, \_\_count + 1));**

}

--------------------------------------------

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;

* **호환성**

**int \_\_count = \_M\_get\_use\_count();**

**do {**

**if (\_\_count == 0)**

**return false;**

**} while (!atomic\_compare\_exchange\_strong(&\_M\_use\_count, &\_\_count, \_\_count + 1));**

**return true;**

}

--------------------------------------------

template<>

inline void \_Sp\_counted\_ptr<nullptr\_t, \_S\_single>::\_M\_dispose() noexcept { }

template<>

inline void \_Sp\_counted\_ptr<nullptr\_t, \_S\_mutex>::\_M\_dispose() noexcept { }

template<>

inline void \_Sp\_counted\_ptr<nullptr\_t, \_S\_atomic>::\_M\_dispose() noexcept { }

* **확장성**

**template<>**

**inline void \_Sp\_counted\_ptr<nullptr\_t>::\_M\_dispose() noexcept { }**

--------------------------------------------

template<int \_Nm, typename \_Tp, bool \_\_use\_ebo = !\_\_is\_final(\_Tp) && \_\_is\_empty(\_Tp)>

struct \_Sp\_ebo\_helper

template<int \_Nm, typename \_Tp>

struct \_Sp\_ebo\_helper<\_Nm, \_Tp, true> : private \_Tp

template<int \_Nm, typename \_Tp>

struct \_Sp\_ebo\_helper<\_Nm, \_Tp, false>

template<typename \_Ptr, typename \_Deleter, typename \_Alloc, \_Lock\_policy \_Lp>

class \_Sp\_counted\_deleter

struct \_Sp\_make\_shared\_tag

template<typename \_Alloc>

struct \_Sp\_alloc\_shared\_tag

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

class \_Sp\_counted\_ptr\_inplace

struct \_\_sp\_array\_delete

* **확장성 제외**

--------------------------------------------

[ class \_shared\_count ]

template<typename \_Tp>

struct \_\_not\_alloc\_shared\_tag { using type = void; };

template<typename \_Tp>

struct \_\_not\_alloc\_shared\_tag<\_Sp\_alloc\_shared\_tag<\_Tp>> { };

* **확장성 제외**

-------------------

template<typename \_Ptr>

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

template<typename \_Ptr>

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

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

\_\_shared\_count(\_Ptr \_\_p)

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

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

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

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

* **확장성 제외**

-------------------

#if \_GLIBCXX\_USE\_DEPRECATED

#pragma GCC diagnostic push

#pragma GCC diagnostic ignored "-Wdeprecated-declarations"

template<typename \_Tp>

explicit \_\_shared\_count(std::auto\_ptr<\_Tp>&& \_\_r);

#pragma GCC diagnostic pop

#endif

* **확장성 & 호환성 제외**

-------------------

template<typename \_Tp, typename \_Del>

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

* **확장성 제외**

-------------------

bool \_M\_unique() const noexcept

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

* **확장성 제외**

--------------------------------------------

template<typename \_Tp, bool = is\_array<\_Tp>::value, bool = is\_void<\_Tp>::value>

class \_\_shared\_ptr\_access

template<typename \_Tp>

class \_\_shared\_ptr\_access<\_Tp, false, true>

template<typename \_Tp>

class \_\_shared\_ptr\_access<\_Tp, true, false>

* **확장성 제외**
* **Shared\_ptr 변경**

template<typename \_Tp>

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

**template<typename \_Tp>**

**class \_\_shared\_ptr**

* **Shared\_ptr 멤버함수 추가 : operator\*() / operator->() / \_M\_get()**

**private:**

**element\_type\* \_M\_get() const noexcept**

**{**

**return static\_cast<const \_\_shared\_ptr<\_Tp>\*>(this)->get();**

**}**

**public:**

**element\_type& operator\*() const noexcept**

**{**

**return \*\_M\_get();**

**}**

**element\_type\* operator->() const noexcept**

**{**

**return \_M\_get();**

**}**

--------------------------------------------

[ class \_shared\_ptr ]

template<typename \_Yp, typename \_Del, typename \_Res = void, typename \_Ptr = typename unique\_ptr<\_Yp, \_Del>::pointer>

using \_UniqCompatible = typename enable\_if<\_\_and\_<\_\_sp\_compatible\_with<\_Yp\*, \_Tp\*>, is\_convertible<\_Ptr, element\_type\*>>::value, \_Res>::type;

template<typename \_Yp, typename \_Del>

using \_UniqAssignable = \_UniqCompatible<\_Yp, \_Del, \_\_shared\_ptr&>;

* **확장성 제외**

-------------------

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

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

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

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

* **확장성 제외**

-------------------

template<typename \_Deleter>

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

template<typename \_Deleter, typename \_Alloc>

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

* **확장성 제외**

-------------------

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

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

* **확장성 제외**

-------------------

template<typename \_Yp, typename \_Del>

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

* **확장성 제외**

-------------------

template<typename \_Yp, typename \_Deleter>

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

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

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

* **확장성 제외**

-------------------

bool unique() const noexcept

* **확장성 제외**

-------------------

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

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

template<typename \_Tp1, typename \_Alloc, typename... \_Args>

friend \_\_shared\_ptr<\_Tp1>

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

* **확장성 제외**

-------------------

#if \_\_cplusplus > 201402L

template<typename \_Tp, typename \_Tp1>

inline \_\_shared\_ptr<\_Tp>

reinterpret\_pointer\_cast(const \_\_shared\_ptr<\_Tp1>& \_\_r) noexcept

{

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

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

}

#endif

* **확장성 제외**

--------------------------------------------

|  |  |  |  |
| --- | --- | --- | --- |
| **문제점 정리** | Shared\_ptr\_base.h   * Void\_t * \_\_shared\_ptr<TP>   Private:  using \_\_esft\_base\_t   * struct \_Sp\_owner\_less | **해결 방안** |  |
| **다음 주차** | **6 주차** | **다음 기간** | **2020.02.17~2020.02.23** |
| **다음주 할 일** |  | | |
| **지도교수**  **Comment** |  | | |