|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | **작성자** | **2014132002 구태균** | **팀명** | - |
| **주차** | **14 주차** | **기간** | **2020.04.13 ~ 2020.04.19** | **지도교수** | **정 내 훈** (서명) |
| **이번주 한일** | * Enable\_shared\_from\_this 분석 | | | | |

**[std::Enable\_shared\_from\_this]**

template<class \_Ux, enable\_if\_t<conjunction\_v<conditional\_t<is\_array\_v<\_Ty>,

\_Can\_array\_delete<\_Ux>, \_Can\_scalar\_delete<\_Ux>>, \_SP\_convertible<\_Ux, \_Ty>>, int> = 0>

explicit shared\_ptr(\_Ux \* \_Px)

{ // construct shared\_ptr object that owns \_Px

\_Setp(\_Px, is\_array<\_Ty>{});

}

template<class \_Ux>

void \_Setp(\_Ux \* \_Px, false\_type)

{ // take ownership of \_Px

\_TRY\_BEGIN // allocate control block and set

\_Set\_ptr\_rep\_and\_enable\_shared(\_Px, new \_Ref\_count<\_Ux>(\_Px));

\_CATCH\_ALL // allocation failed, delete resource

delete \_Px;

\_RERAISE;

\_CATCH\_END

}

template<class \_Ux>

void \_Set\_ptr\_rep\_and\_enable\_shared(\_Ux \* \_Px, \_Ref\_count\_base \* \_Rx)

{ // take ownership of \_Px

this->\_Set\_ptr\_rep(\_Px, \_Rx);

\_Enable\_shared\_from\_this(\*this, \_Px);

}

void \_Set\_ptr\_rep(element\_type \* \_Other\_ptr, \_Ref\_count\_base \* \_Other\_rep)

{ // take new resource

\_Ptr = \_Other\_ptr;

\_Rep = \_Other\_rep;

}

template<class \_Other,class \_Yty>

void \_Enable\_shared\_from\_this(const shared\_ptr<\_Other>& \_This, \_Yty \* \_Ptr)

{ // possibly enable shared\_from\_this

\_Enable\_shared\_from\_this1(\_This, \_Ptr, bool\_constant<conjunction\_v<

negation<is\_array<\_Other>>,

negation<is\_volatile<\_Yty>>,

\_Can\_enable\_shared<\_Yty>>>{});

}

template<class \_Other,class \_Yty>

void \_Enable\_shared\_from\_this1(const shared\_ptr<\_Other>& \_This, \_Yty \* \_Ptr, true\_type)

{ // enable shared\_from\_this

if (\_Ptr && \_Ptr->\_Wptr.expired())

\_Ptr->\_Wptr = shared\_ptr<remove\_cv\_t<\_Yty>>(\_This, const\_cast<remove\_cv\_t<\_Yty> \*>(\_Ptr));

}

**[lfsp::Enable\_shared\_from\_this]**

template<class \_Ux,

enable\_if\_t<conjunction\_v<conditional\_t< is\_array\_v<\_Ty>,\_Can\_array\_delete<\_Ux>, \_Can\_scalar\_delete<\_Ux> >, \_SP\_convertible<\_Ux, \_Ty>>, int> = 0>

explicit shared\_ptr(\_Ux \* \_Px)

{ // construct shared\_ptr object that owns \_Px

\_Setp(\_Px);

}

**< enable\_if\_t >**

template <bool \_Test, typename \_Ty = void>

struct enable\_if { };

 // \_Ty에 대한 type aliasing을 제공하지 않는다.

template <typename \_Ty>

struct enable\_if<true, \_Ty> { using type = \_Ty ; };

// \_Ty에 대한 type aliasing을 제공.

template<bool \_Test, typename \_Ty = void>

using enable\_if\_t = typename enable\_if<\_Test, \_Ty>::type;

enable\_if<true, \_Ty> : \_Ty에 대한 type aliasing 제공

enable\_if<false, \_Ty> : 어떤 타입도 제공하지 않음 ( 컴파일 에러 )

**< conjunction\_v >**

: 유형의 특성을 논리적으로 연결해, 시퀀스에서 논리적 AND를 수행.

template<class... B>  
inline constexpr bool conjunction\_v = conjunction<B...>::value;

**< conditional\_t >**

template <bool BoolType, typename TrueType, typename FalseType>

struct **conditional** { using type = FalseType; };

// BoolType이 false라면, type은 FalseType

template <typename TrueType, typename FalseType>

struct **conditional**<true, TrueType, FalseType> { using type = TrueType; };

// BoolType이 true라면, type은 TrueType

template <bool BoolType, typename TrueType, typename FalseType>

using **conditional\_t**= typename conditional<BoolType, TrueType, FalseType>::type;

<컴파일 타임>

BoolType의 평가가 true이면, std::condition::type은 TrueType 형식

BoolType의 평가가 false이면, std::condition::type은 FalseType 형식

**< is\_array\_v >**

: 유형이 array일 경우 true, 이외일 경우 false

template< class T >  
struct is\_array;

template< class T >  
constexpr bool is\_array\_v = is\_array<T>::value;

\_Can\_array\_delete<\_Ux>, \_Can\_scalar\_delete<\_Ux> ------- ??

-> False\_type으로 대체

\_SP\_convertible<\_Ux, \_Ty> ------- ??

-> type?

template<class \_Ux>

void \_Setp(\_Ux \* \_Px)

{ // take ownership of \_Px

\_Set\_ptr\_rep\_and\_enable\_shared(\_Px, new \_Ref\_count<\_Ux>(\_Px)); // control\_block 생성

}

template<class \_Ux>

void \_Set\_ptr\_rep\_and\_enable\_shared(\_Ux \* \_Px, \_Ref\_count\_base \* \_Rx)

{ // take ownership of \_Px

this->\_Set\_ptr\_rep(\_Px, \_Rx); // control\_block 초기화 : ptr, ctr 초기화

\_Enable\_shared\_from\_this(\*this, \_Px); // enable\_shared\_from\_this 초기화

}

void \_Set\_ptr\_rep(element\_type \* \_Other\_ptr, \_Ref\_count\_base \* \_Other\_rep)

{ // take new resource

\_Ptr = \_Other\_ptr;

\_Rep = \_Other\_rep;

}

template<class \_Other,class \_Yty>

void \_Enable\_shared\_from\_this(const shared\_ptr<\_Other>& \_This, \_Yty \* \_Ptr)

{ // possibly enable shared\_from\_this

\_Enable\_shared\_from\_this1(\_This, \_Ptr,

bool\_constant< conjunction\_v< negation< is\_array< \_Other > >, negation< is\_volatile< \_Yty > >, \_Can\_enable\_shared< \_Yty > > >{});

// enable\_shared\_from\_this Wptr 초기화 구분

* 만약 enable\_shared\_from\_this를 상속받았다면, Wptr을 초기화

}

template<class \_Other,class \_Yty>

void \_Enable\_shared\_from\_this1(const shared\_ptr<\_Other>& \_This, \_Yty \* \_Ptr, true\_type)

{ // enable shared\_from\_this

if (\_Ptr && \_Ptr->\_Wptr.expired())

\_Ptr->\_Wptr = shared\_ptr<\_Yty>(\_This, const\_cast<\_Yty\*>(\_Ptr));

// enable\_shared\_fron\_this Wptr 초기화

}

template<class \_Other, class \_Yty>

void \_Enable\_shared\_from\_this1(const shared\_ptr<\_Other>&, \_Yty \*, false\_type)

{ // don't enable shared\_from\_this

}

**[atomic\_load()]**

**[atomic\_store()]**

|  |  |  |  |
| --- | --- | --- | --- |
| **문제점 정리** |  | **해결 방안** |  |
| **다음 주차** | **16 주차** | **다음 기간** | **2020.04.27 ~ 2020.05.03** |
| **다음주 할 일** |  | | |
| **지도교수**  **Comment** |  | | |