C++ Korea 5th Seminar

C++20 Key Features Summary

옥찬호

Nexon Korea, Microsoft MVP

utilForever@gmail.com

소개

옥찬호 (Chris Ohk)

- 넥슨 코리아 게임 프로그래머
- Microsoft Developer Technologies MVP
- 페이스북 그룹 C++ Korea 대표
- IT 전문서 집필 및 번역 다수
  - 게임샐러드로 코드 한 줄 없이 게임 만들기 (2013)
  - 유니티 Shader와 Effect 제작 (2014)
  - 2D 게임 프로그래밍 (2014)
  - 러스트 핵심 노트 (2017)
  - 모던 C++ 입문 (2017)
  - C++ 최적화 (2019)

시작 전에 잠깐 광고

<하스스톤> 강화학습 환경 개발기 – 0티어 덱을 만들기 위해 떠나는 모험

옥찬호 – 넥슨코리아 / NEXON KOREA

판교 GBI타워 B1 발표장 | 13:30 ~ 14:20

발표분야 : 프로그래밍

발표내용의난이도 : 사전지식 불필요 : 튜토리얼이나 개요 수준에서의 설명

수강권장대상 : 게임 프로그래머, 카드 게임 개발이나 강화학습 환경 구축에 관심이 많은 프로그래머

학생참관여부 : 학생 참관 가능

공개수준 : L1–Public Open

키워드 : 하스스톤 / TCG / 강화학습

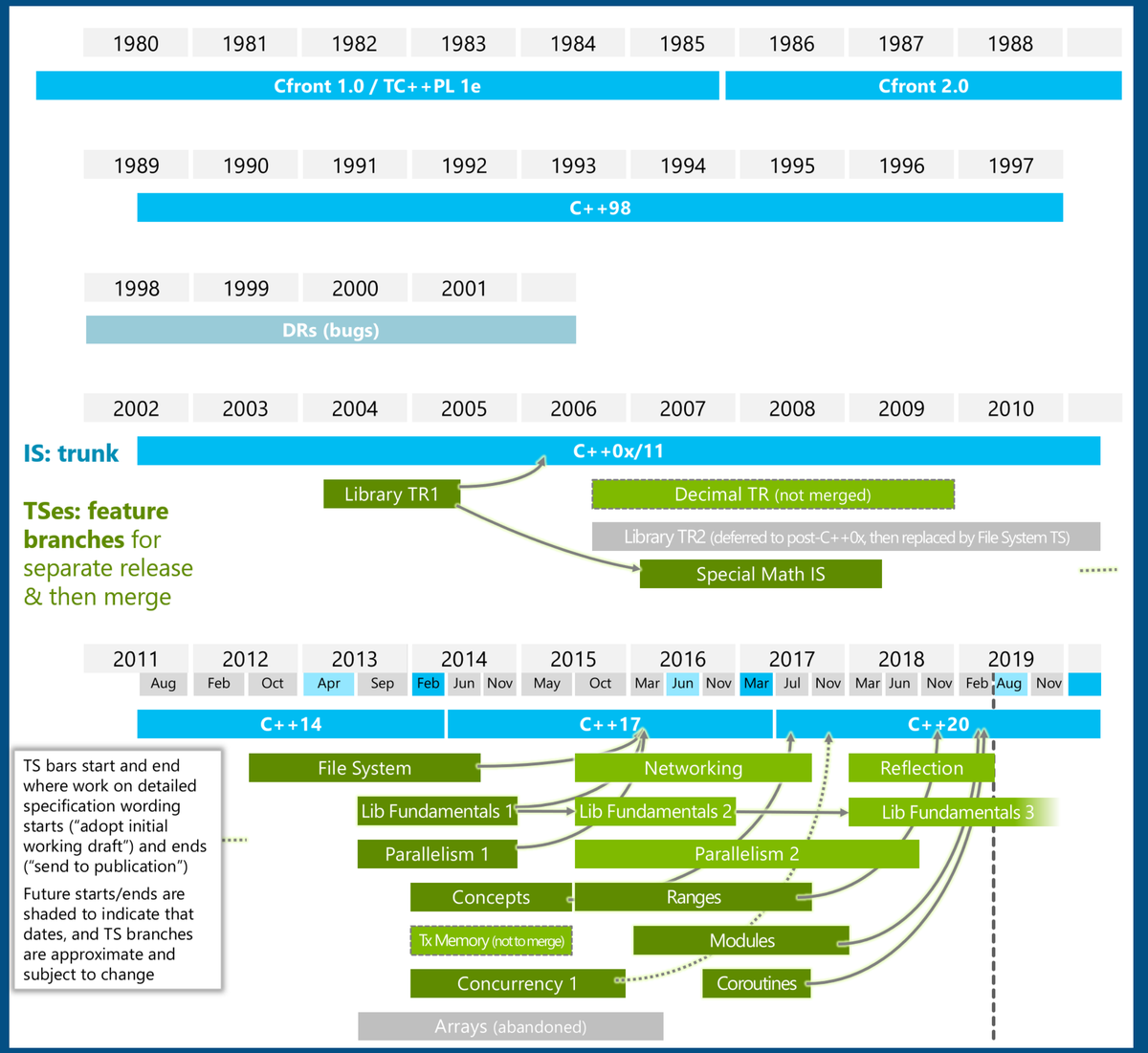
발표자 소개

현재 넥슨 코리아에서 마영전 게임 클라이언트 개발을 담당하고 있습니다. C++와 게임 개발, 컴퓨터 그래픽스, 강화학습, 오픈 소스에 관심이 많습니다. 페이스북 C++ Korea 그룹을 운영하며 스터디 활동과 함께 다양한 책을 번역/집필 중이며, 남는 시간엔 학생들과 함께 다양한 오픈 소스 프로젝트를 진행하며 보다 나은 미래를 만들기 위해 노력하고 있습니다. 옮긴 책으로는 『게임샐러드로 코드 한 줄 없이 게임 만들기』 (에이콘출판사, 2013), 『유니티 Shader와 Effect 제작』 (에이콘출판사, 2014), 『2D 게임 프로그래밍』 (에이콘출판사, 2014), 『러스트 핵심 노트』 (한빛미디어, 2017), 『모던 C++ 입문』 (길벗, 2017)이 있습니다.

시작하기 전에...

- C++20에 추가될 주요 기능들을 설명합니다.
- C++20에 추가될 기능 중 C++11/14/17 지식이 필요한 경우,
  이해를 돕기 위해 사전 지식을 먼저 설명합니다.
- 현재 컴파일러에서 사용할 수 있는 기능 위주로 설명합니다.
  (실제 예제 코드가 어떻게 동작하는지 설명하기 위해)
- 모든 예제 코드는 발표 이후 Github를 통해 제공됩니다.
  (https://github.com/utilForever/ModernCpp)

현재 표준 C++ 진행 상황


1980 1981 1982 1983 1984 1985 1986 1987 1988
Cfront 1.0 / TC++PL 1e
Cfront 2.0
1989 1990 1991 1992 1993 1994 1995 1996 1997
C++98
1998 1999 2000 2001
DRs (bugs)
2002 2003 2004 2005 2006 2007 2008 2009 2010
IS: trunk
C++0x/11
Library TR1
Decimal TR (not merged)
Library TR2 (deferred to post-C++0x, then replaced by File System TS)
Special Math IS
TSes: feature branches for separate release & then merge
2011 2012 2013 2014 2015 2016 2017 2018 2019
Aug Feb Oct Apr Sep Feb Jun Nov May Oct Mar Jun Nov Mar Jul Nov Mar Jun Nov Feb Aug Nov
C++14
C++17
C++20
TS bars start and end where work on detailed specification wording starts ("adopt initial working draft") and ends ("send to publication")
Future starts/ends are shaded to indicate that dates, and TS branches are approximate and subject to change
File System
Networking
Reflection
Lib Fundamentals 1
Lib Fundamentals 2
Lib Fundamentals 3
Parallelism 1
Parallelism 2
Concepts
Ranges
Tx Memory (not to merge)
Modules
Concurrency 1
Coroutines
Arrays (abandoned)

현재 표준 C++ 진행 상황

https://twitter.com/GorNishanov

현재 표준 C++ 진행 상황

https://twitter.com/GorNishanov

가벼운 퀴즈

C++17 quiz: Structured bindings

Given a Point p and structured binding [a,b], what is the value of **p.x** and **p.y**?

```
struct Point {
    Point() : x(0),y(0){}
    int x;
    int y;
};
```

```
int main(){
   Point p;
   auto [a,b] = p;
   a = 1;
   b = 2;
   std::cout << p.x << " "
             << p.y << '\n';
}
```

A

0,0

B

1,2

@walletfox

https://twitter.com/walletfox

C++20 주요 기능

C++ Korea 5th Seminar
C++20 Key Features Summary

- Concepts
- Contracts
- Ranges
- Module
- Coroutines

Concepts

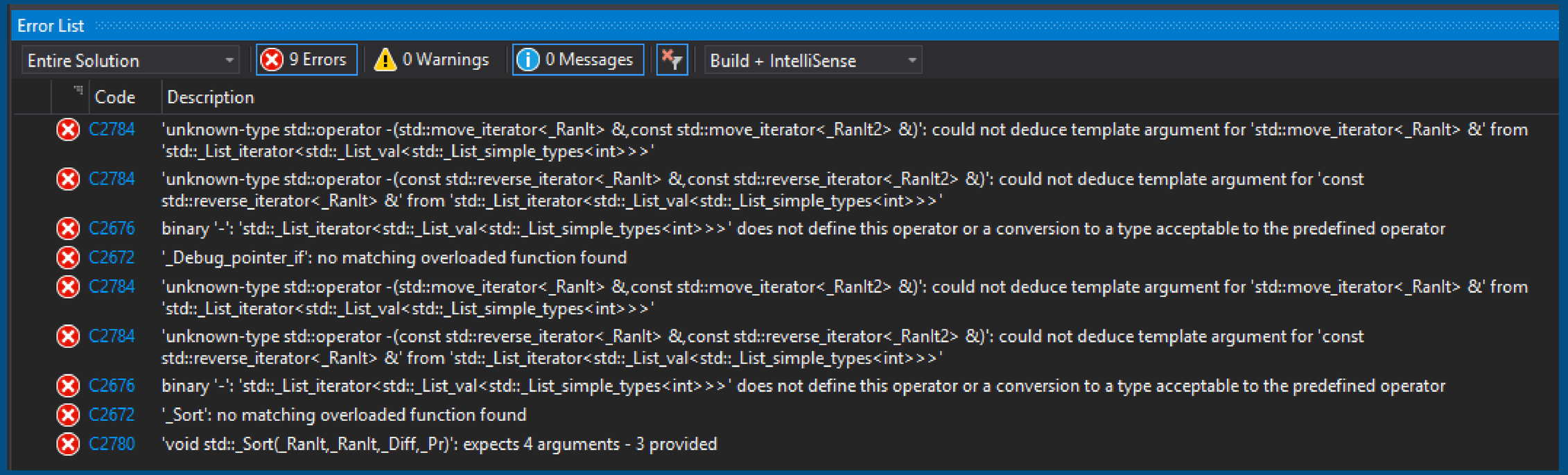
- 다음 코드의 출력 결과는?

```
#include <algorithm>
#include <list>

int main()
{
    std::list<int> l = { 2, 1, 3 };
    std::sort(l.begin(), l.end());
}
```

Concepts

- 다음 코드의 출력 결과는?



Concepts

- 왜 오류가 발생할까요?
  - ```
    template <class RandomIt>
    void sort(RandomIt first, RandomIt last);
    ```
  - 여기서 **RandomIt**는 **ValueSwappable**과 **RandomAccessIterator**의 제약 사항을 만족해야 합니다. 하지만 `int`에는 반복자(Iterator)가 존재하지 않습니다.
    ```

Concepts

- C#에서는 제약 사항을 위해 다양한 클래스와 인터페이스를 제공합니다.
  - `Icloneable`
  - `Icomparable`
  - `Iconvertible`
  - `Iformattable`
  - `Nullable`
  - `...`
- C++에는 지금까지 이러한 기능을 지원하지 않았습니다.
  Concept이 등장하기 전까지는 말이죠.

Concepts

- Concept을 사용해 템플릿 매개 변수에 제약을 걸 수 있습니다.
  - 리턴 타입 앞에 concept 키워드를 추가합니다.
    변수일 경우 타입이 bool이어야 하고, 함수일 경우 리턴 타입이 bool이어야 합니다.
  - requires 키워드를 통해 제약 조건을 설정합니다.
    리턴 타입은 constexpr bool 또는 concept bool이어야 합니다.

```
template <class T>
concept bool EqualityComparable()
{
    return requires(T a, T b) {
        {a == b}->Boolean; // Boolean is the concept defining
        {a != b}->Boolean; // a type usable in boolean context
    };
}
```

Concepts

- 사용하는 방법은 기존 템플릿과 같습니다.

```
void f(const EqualityComparable&);

// OK, std::string satisfies EqualityComparable
f("abc"s);
// Error: not EqualityComparable
f(std::use_facet<std::ctype<char>>(std::locale{}));
```

Concepts

- Concept을 사용했을 때의 가장 큰 장점은 가독성이라고 할 수 있습니다.

```
In instantiation of 'void std::__sort(_RandomAccessIterator,
_RandomAccessIterator, _Compare) [with _RandomAccessIterator
= std::_List_iterator<int>;
_Compare = __gnu_cxx::__ops::_Iter_less_iter]':
error: no match for 'operator-' (operand types are
'std::_List_iterator<int>' and 'std::_List_iterator<int>')
std::__lg(__last - __first) * 2,
```

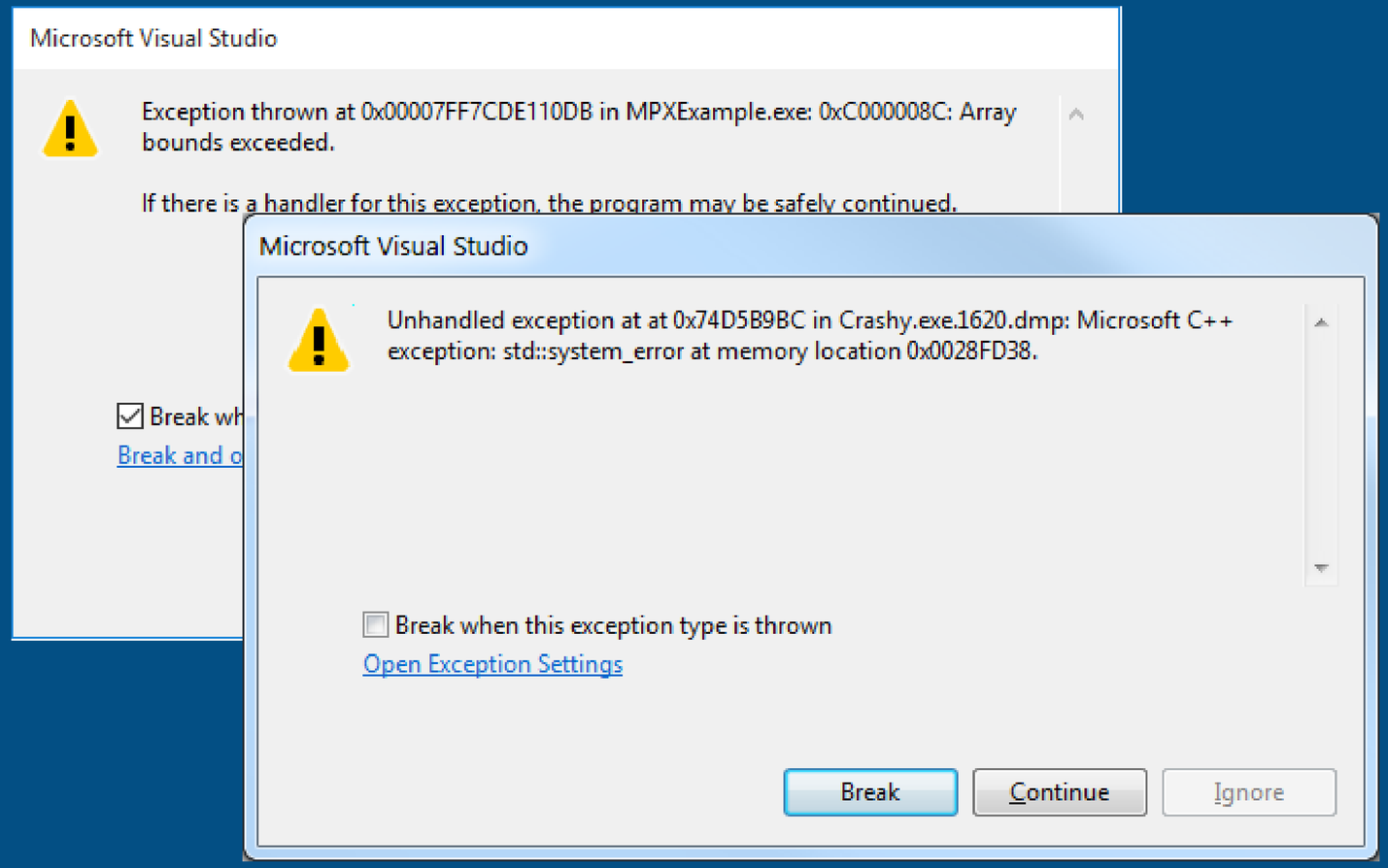
```
error: cannot call function 'void std::sort(_RAIter, _RAIter)
[with _RAIter = std::_List_iterator<int>]'
note:   concept 'RandomAccessIterator()' was not satisfied
```

Concepts

- Concept 리스트는 다음과 같습니다.
  - Basic : `DefaultConstructible, MoveConstructible, CopyConstructible, MoveAssignable, CopyAssignable, Destructible`
  - Library-wide : `EqualityComparable, LessThanComparable, Swappable, ValueSwappable, NullablePointer, Hash, Allocator, FunctionObject, Callable, Predicate`
  - Iterator : `InputIterator, OutputIterator, ForwardIterator, BidirectionalIterator, RandomAccessIterator, ContiguousIterator`

Contract

- 우리는 항상 버그와 함께합니다. (maybe bugs be with you)



Contract

- 우리는 발생할 수 있는 버그를 미리 파악하기 위해 assert나 static_assert를 사용합니다.

```
void print_number(int* myInt)
{
    assert(myInt != nullptr);
    std::cout << *myInt << "\n";
}

int main()
{
    int a = 10;
    int *b = nullptr, *c = nullptr;
    b = &a;

    print_number(b); print_number(c);
}
```

Contract

- Contract는 함수 앞에 사전 조건과 사후 조건을 추가해 assert()처럼 특정 조건을 만족하는지 검사할 수 있는 기능입니다.
  - expects : 사전 조건(Pre-condition)을 위한 키워드
  - ensures : 사후 조건(Post-condition)을 위한 키워드

```
template<typename T>
class ArrayView
{
    T& operator[](size_t i)[[expects:i < size()]];

    ArrayView(const vector<T>& v)[[ensures:data() == v.data()]];
};
```

Contract

- 사용할 때 주의할 점은 상속 관계에서 함수를 오버라이드할 때 사전 조건이나 사후 조건이 강해지거나 약해지면 안됩니다.

```
struct A {
    bool f() const; bool g() const;
    virtual std::string bhar()[[expects:f() && g()]];
    virtual int hash()[[ensures:g()]];
    virtual void gash()[[expects:g()]];
    virtual double fash(int i) const[[expects:i > 0]];
};

struct B : A {
    std::string bhar() override[[expects:f()]]; // ERROR: weakening.
    int hash() override[[ensures:f() && g()]];  // ERROR: strengthening.
    void gash() override[[expects:g()]];        // OK: repeat from base.
    double fash(int) override;                  // OK: inherited from base.
};
```

Ranges

- Ranges는 범위를 다루는 제너릭 라이브러리입니다.
- Ranges의 장점
  - 편의성 (Convenience) : 짧아지는 코드
  - 느긋한 계산 (Lazy Evaluation) : 무한 범위 표현 가능, 불필요한 계산 회피
  - 조합성 (Composability) : 직관적인 표현 가능, 코드 생산성 향상
- Ranges에 대해 좀 더 깊이 있는 지식을 알고 싶다면, 이전 세미나에서 발표했던 “Ranges for C++ Standard Library”를 참고하세요.
  (https://www.slideshare.net/seao/c-korea-2nd-seminar-ranges-for-the-cpp-standard-library)

Ranges

- [0 ~ 10)을 채워봅시다.

```
int arr[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

```
std::vector<int> v(10);
for (int i = 0; i < 10; ++i)
    v[i] = i;
```

```
std::vector<int> v = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

Ranges

- [0 ~ 10)을 채워봅시다.

```
#include <numeric>
std::vector<int> v(10);
std::iota(v.begin(), v.end(), 0);
```

```
std::vector<int> v(10);
ranges::iota(v, 0);
```

Ranges

- [0 ~ 15)에서 5개씩 그룹을 만들어 각 그룹의 합을 구하는 예제

```
vector<int> v =
    view::iota(0, 15) |
    view::group_by(quotient) |
    view::transform(sum);
for (auto e : v)
    cout << e << " "; cout << endl;

auto quotient = [](int a, int b) {
    return a / 5 == b / 5;
};

auto sum = [](auto rng) {
    return ranges::accumulaate(rng, 0);
};
```

Ranges

- $a^2 + b^2 = c^2$ 을 만족하는 세 자연수 쌍 (a, b, c) 100개 구하는 예제

```
auto triples = view::ints(1)         >>= [] (int z) { return
               view::ints(1, z + 1) >>= [=](int x) { return
               view::ints(x, z + 1) >>= [=](int y) { return
         yield_if(x*x + y*y == z*z,
               std::make_tuple(x, y, z));
}; }; };

auto triangles = triples | view::take(100);
```

Module

- C++에서는 #include를 통해 헤더 파일을 포함합니다.
- 문제는 헤더 파일 안에 있는 모든 부분을 포함한다는 점입니다.
  (이로 인해 포함하고 싶지 않은 부분도 가져오게 됩니다.)
- 다른 언어는 필요한 부분만 가져올 수 있는 기능을 제공하고 있습니다.
  - C# : `using System.Linq;`
  - Java : `import java.util.Scanner;`
  - Python : `from path import pi`
- 이제 C++에도 Module이 추가되어 빛을 볼 수 있게 되었습니다.

Module

- 여러분이 Module에서 기억해야 할 키워드는 3가지입니다.
  - `module` : 모듈로 정의할 이름을 지정합니다.
    예) `module M1` : 정의할 모듈의 이름은 `M1`이다.
  - `import` : 가져올 모듈의 이름을 지정합니다.
    예) `import M1` : 가져올 모듈의 이름은 `M1`이다.
  - `export` : 내보낼 함수의 인터페이스를 지정합니다.
    예) `export int f(int, int)`
    내보낼 함수의 이름은 `f`이고 리턴 타입은 `int`, 매개 변수는 `(int, int)`다.

Module

- 두 cpp 파일에서 모듈을 정의해 다른 cpp 파일에서 사용하는 예제

```
// m1.cpp
module M1;
export int f(int, int);
int g(int x) { return x * x; }
int f(int x, int y) { return g(x) + g(y); }

// m2.cpp
module M2;
export int g(int, int);
import std.core;
int f(int x, int y) { return x + y; }
int g(int x, int y)
{
    return f(abs(x), abs(y));
}
```

```
// main.cpp
import M1;
import M2;

int main()
{
    printf("%d\n",
           f(3, 4) + g(3, 4));
}
```

Coroutine

- “C++ Coroutine 알아보기: 접근법, 컴파일러, 그리고 이슈들” 세션에서 Coroutine에 대한 내용을 발표할 예정입니다.
  (발표 자료를 올릴 때는 내용을 추가할 예정이니 참고 부탁드립니다.)

C++20 주요 기능

- Designated initializers
- `operator<=>`
- Relation operator/comparison
- Nested inline namespaces
- `consteval` fuctions
- Feature test macros

C++20 주요 기능

- `std::span`
- `std::endian`
- `std::bit_cast`
- Calender and timezone library

Designated initializers

- 다음과 같은 구조체가 있다고 합시다.

```
struct A { int x; int y; int z; };
```

- 구조체 타입 변수를 선언해 봅시다.

```
A a1{3, 4};
A a2{3, 4, 5};
```

- 만약 x, z에만 값을 대입하고 싶다면 어떻게 해야 할까요?

  아마도 이렇게 해야 할 것입니다.

```
A a1;
a1.x = 3;
a1.z = 5;
```

Designated initializers

- C++20에서는 좀 더 편한 방법으로 대입할 수 있습니다.

```
A a1{.x = 3, .z = 5};
```

- 단, 초기화 순서는 변수의 선언 순서와 일치해야 합니다.

```
A a1{.x = 3, .y = 4};    // OK
A a2{.y = 4, .x = 3};    // Error
```

- 공용체에도 사용할 수 있습니다.

```
union u { int a; const char* b; };
u f = { .b = "asdf" };          // OK, active member of the union is b
u g = { .a = 1,.b = "asdf" };  // Error, only one initializer may be provided
```

operator<=>

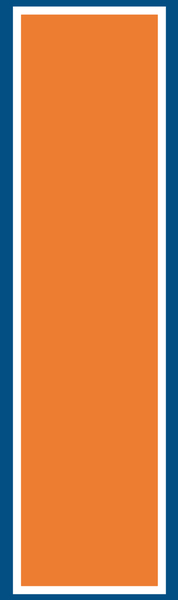
- strcmp와 유사한 동작을 하며 3가지의 값을 반환하는 연산자입니다.
  (Spaceship operator, 3-way comparison이라고도 합니다.)
  - a가 b보다 크면 1을 반환합니다.
  - a가 b보다 작으면 -1을 반환합니다.
  - a가 b와 '**같거나(equal)**' '**동등하면(equivalent)**' 0을 반환합니다.
    (참고 : '같다'와 '동등하다'는 다른 개념입니다.)

```
(a <=> b) < 0       // true if a < b
(a <=> b) < 0       // true if a > b
(a <=> b) == 0      // true if a is equal/equivalent to b
```

Relation operator/comparison

- 다음 두 직사각형이 있습니다.

가로가 4m, 세로가 1m인 직사각형

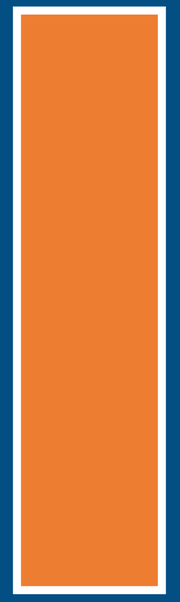


가로가 1m, 세로가 4m인 직사각형

Relation operator/comparison

- 두 직사각형은 분명히 다릅니다. 하지만 넓이는 같습니다.
  넓이가 큰 순서대로 정렬한다면 무엇이 앞에 정렬될까요?

가로가 4m, 세로가 1m인 직사각형

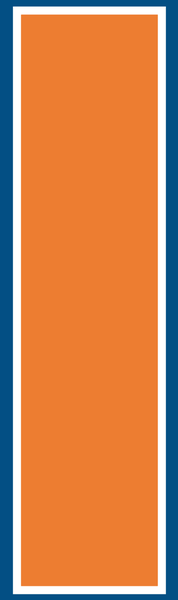


가로가 1m, 세로가 4m인 직사각형

Relation operator/comparison

- 이 때 두 직사각형을 ‘동등하다’고 합니다.
  다른 말로 weak ordering이라고 합니다.

가로가 4m, 세로가 1m인 직사각형



가로가 1m, 세로가 4m인 직사각형

Relation operator/comparison

- 다음 두 정사각형이 있습니다.

가로가 4m, 세로가 4m인 정사각형

가로가 4m, 세로가 4m인 정사각형

Relation operator/comparison

- 두 정사각형은 같습니다.
  가로/세로 길이도 같고 넓이도 같습니다.

가로가 4m, 세로가 4m인 정사각형

가로가 4m, 세로가 4m인 정사각형

Relation operator/comparison

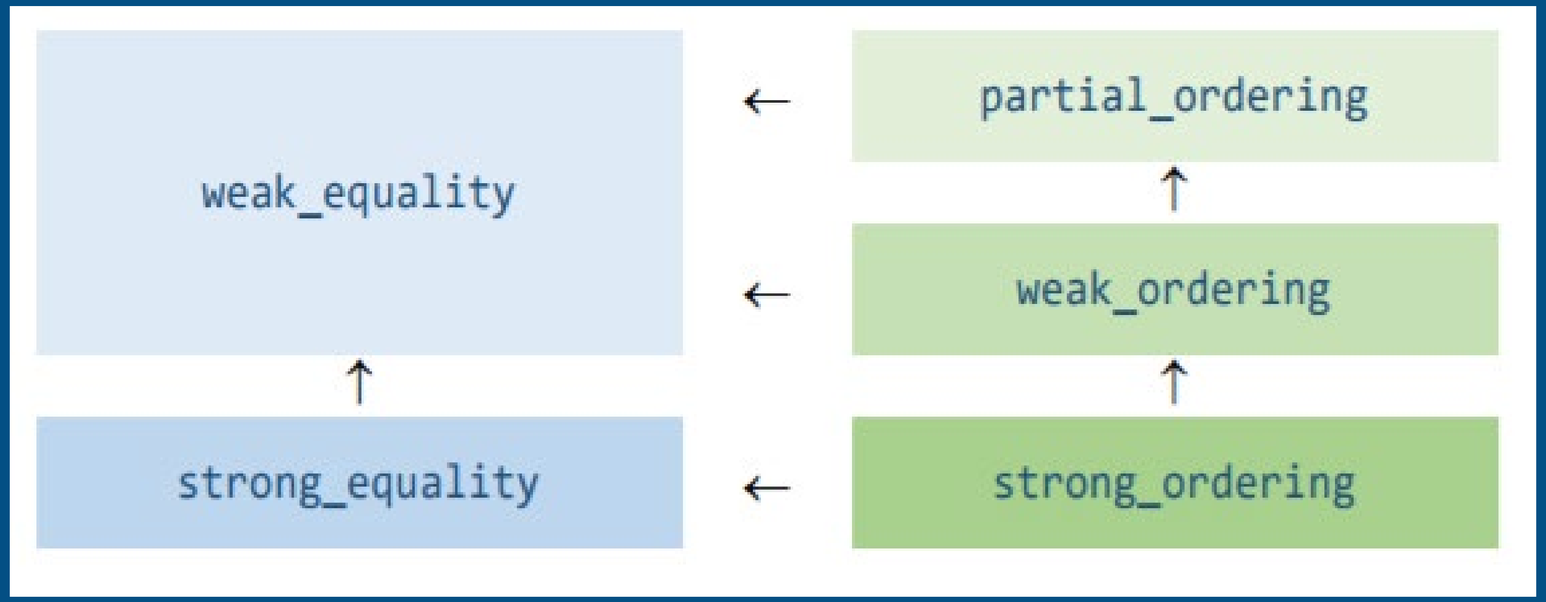
- 이때 두 정사각형을 ‘같다’라고 합니다.
  다른 말로 strong ordering이라고 합니다.

가로가 4m, 세로가 4m인 정사각형

가로가 4m, 세로가 4m인 정사각형

Relation operator/comparison

- operator<=>의 리턴 타입은 다음 다섯 타입 중 하나가 됩니다.



https://medium.com/@barryrevzin/implementing-the-spaceship-operator-for-optional-4de89fc6d5ec

Relation operator/comparison

- 다섯 타입의 값이 의미하는 바는 다음과 같습니다.

	Numeric values			
Category	-1	0	+1	Non-numeric values
strong_ordering	less	equal	greater	
weak_ordering	less	equivalent	greater	
partial_ordering	less	equivalent	greater	unordered
strong_equality		equal	nonequal	
weak_equality		equivalent	nonequivalent	

https://medium.com/@barryrevzin/implementing-the-spaceship-operator-for-optional-4de89fc6d5ec

Relation operator/comparison

- operator<=> + relation operator를 사용한 예제

```
class Rectangle {
public:
    Rectangle(int x, int y) : m_x(x), m_y(y) { }

    int GetArea() const { return m_x * m_y; }

    std::weak_ordering operator<=>(const Rectangle& r) const {
        //if (auto cmp = m_x <=> r.m_x; cmp != 0) return cmp;
        //if (auto cmp = m_y <=> r.m_y; cmp != 0) return cmp;
        return GetArea() <=> r.GetArea();
    }

private:
    int m_x, m_y;
};
```

consteval functions

- constexpr 함수
  - 컴파일 타임에 계산 가능하면 컴파일 타임에 계산합니다.
  - 컴파일 타임에 계산이 불가능하면 런타임에 계산합니다.
- consteval 함수
  - 컴파일 타임에 계산 가능하면 컴파일 타임에 계산합니다.
  - 컴파일 타임에 계산이 불가능하면 컴파일 오류가 발생합니다.

consteval functions

- 함수를 호출할 때마다 컴파일 타임에 상수 표현식을 생성합니다.
- consteval을 사용한 예제

```
consteval int sqr(int n) { return n * n; }
constexpr int r = sqr(100);  // OK

consteval int sqrsqr(int n)
{
  return sqr(sqr(n)); // OK
}

constexpr int dblsqr(int n)
{
  return 2 * sqr(n); // Error
}
```

Nested inline namespaces

- 중첩 네임스페이스 + 인라인

```
namespace A::B::inline C
{
    ...
}
```

=

```
namespace A::B
{
    inline namespace C
    {
        ...
    }
}
```

Nested inline namespaces

- 중첩 네임스페이스 + 인라인

```
namespace A::inline B::C
{
    ...
}
```

=

```
namespace A
{
    inline namespace B
    {
        namespace C
        {
            ...
        }
    }
}
```

Feature test macros

- 세상에는 다양한 C++ 컴파일러가 있습니다.
  - GCC
  - Clang
  - MSVC
  - EDG eccp
  - Intel C++
  - Portland Group (PGI)
  - …

Feature test macros

- 컴파일러마다 버전도 다양합니다.

```
MSC     1.0   _MSC_VER == 100
MSC     2.0   _MSC_VER == 200
MSC     3.0   _MSC_VER == 300
MSC     4.0   _MSC_VER == 400
MSC     5.0   _MSC_VER == 500
MSC     6.0   _MSC_VER == 600
MSC     7.0   _MSC_VER == 700
MSVC++ 1.0   _MSC_VER == 800
MSVC++ 2.0   _MSC_VER == 900
MSVC++ 4.0   _MSC_VER == 1000 (Developer Studio 4.0)
MSVC++ 4.2   _MSC_VER == 1020 (Developer Studio 4.2)
MSVC++ 5.0   _MSC_VER == 1100 (Visual Studio 97 version 5.0)
MSVC++ 6.0   _MSC_VER == 1200 (Visual Studio 6.0 version 6.0)
MSVC++ 7.0   _MSC_VER == 1300 (Visual Studio .NET 2002 version 7.0)
MSVC++ 7.1   _MSC_VER == 1310 (Visual Studio .NET 2003 version 7.1)
MSVC++ 8.0   _MSC_VER == 1400 (Visual Studio 2005 version 8.0)
MSVC++ 9.0   _MSC_VER == 1500 (Visual Studio 2008 version 9.0)
MSVC++ 10.0  _MSC_VER == 1600 (Visual Studio 2010 version 10.0)
MSVC++ 11.0  _MSC_VER == 1700 (Visual Studio 2012 version 11.0)
MSVC++ 12.0  _MSC_VER == 1800 (Visual Studio 2013 version 12.0)
MSVC++ 14.0  _MSC_VER == 1900 (Visual Studio 2015 version 14.0)
MSVC++ 14.1  _MSC_VER == 1910 (Visual Studio 2017 version 15.0)
MSVC++ 14.11 _MSC_VER == 1911 (Visual Studio 2017 version 15.3)
MSVC++ 14.12 _MSC_VER == 1912 (Visual Studio 2017 version 15.5)
MSVC++ 14.13 _MSC_VER == 1913 (Visual Studio 2017 version 15.6)
MSVC++ 14.14 _MSC_VER == 1914 (Visual Studio 2017 version 15.7)
MSVC++ 14.15 _MSC_VER == 1915 (Visual Studio 2017 version 15.8)
MSVC++ 14.16 _MSC_VER == 1916 (Visual Studio 2017 version 15.9)
```

Feature test macros

- 컴파일러 종류 및 버전에 따라 표준 지원 여부가 다릅니다.

  (apple-clang 손절하세요... 모두들 따라 쳐봅시다. “brew install clang”)

```
#if defined(ROSETTASTONE_WINDOWS)
#include <filesystem>
#elif defined(ROSETTASTONE_LINUX)
#include <experimental/filesystem>
#elif defined(ROSETTASTONE_MACOSX)
#include <sys/stat.h>
#endif

#if defined(ROSETTASTONE_WINDOWS)
namespace filesystem = std::filesystem;
#elif defined(ROSETTASTONE_LINUX)
namespace filesystem = std::experimental::filesystem;
#endif
```

Feature test macros

- 정교하게 지원 여부를 확인하다 보면 전처리 지옥에 빠지게 됩니다.

```
#ifdef __has_include
#  if __has_include(<optional>)
#    include <optional>
#  elif __has_include(<experimental/optional>)
#    include <experimental/optional>
#  elif __has_include(<boost/optional.hpp>)
#    include <boost/optional.hpp>
#  else
#     error "Missing <optional>"
#  endif
#endif
```

Feature test macros

- C++20에는 기능 테스트 매크로라는 기능이 추가되었는데,
  매크로를 사용해 C++의 주요 기능을 버전별로 테스트할 수 있습니다.

```
#if __cpp_constexpr >= 201304
#  define CONSTEXPR constexpr
#else
#  define CONSTEXPR inline
#endif

CONSTEXPR int bar(unsigned i)
{
    ...
}
```

Feature test macros

- C++20에는 기능 테스트 매크로라는 기능이 추가되었는데,
  매크로를 사용해 C++의 주요 기능을 버전별로 테스트할 수 있습니다.

```
CONSTEXPR int bar(unsigned i)
{
#if __cpp_binary_literals
    unsigned mask1 = 0b11000000;
    unsigned mask2 = 0b00000111;
#else
    unsigned mask1 = 0xC0;
    unsigned mask2 = 0x07;
#endif
    if ( i & mask1 ) return 1;
    if ( i & mask2 ) return 2;
    return 0;
}
```

Feature test macros

- 그리고 C++20에 새로 추가된 __has_cpp_attribute를 사용하면 특성(attribute) 지원 여부를 확인할 수 있습니다.

```
#ifdef __has_cpp_attribute
#  if __has_cpp_attribute(deprecated)
#    define DEPRECATED(msg) [[deprecated(msg)]]
#  endif
#endif
#ifndef DEPRECATED
#    define DEPRECATED(msg)
#endif

DEPRECATED("foo() has been deprecated") void foo();
```

std::span

- std::array_view의 이름이 바뀌었습니다.
- std::string <-> std::string_view와 같은 관계입니다.
- 연속 공간을 갖는 모든 컨테이너에 사용 가능합니다.
  - std::vector
  - std::string
  - T[]
  - T* + size

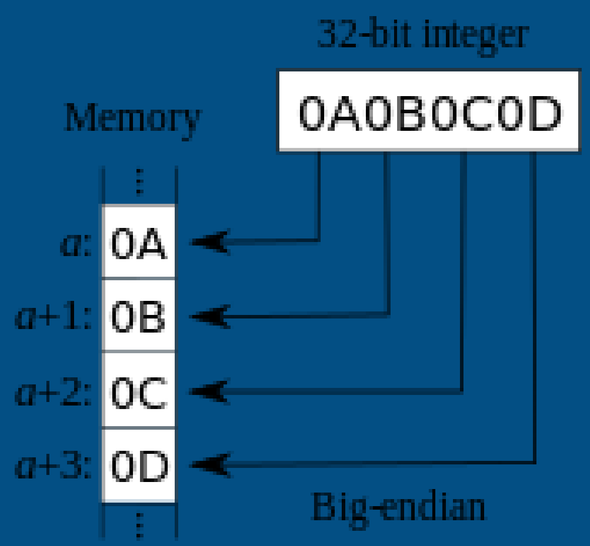
std::span

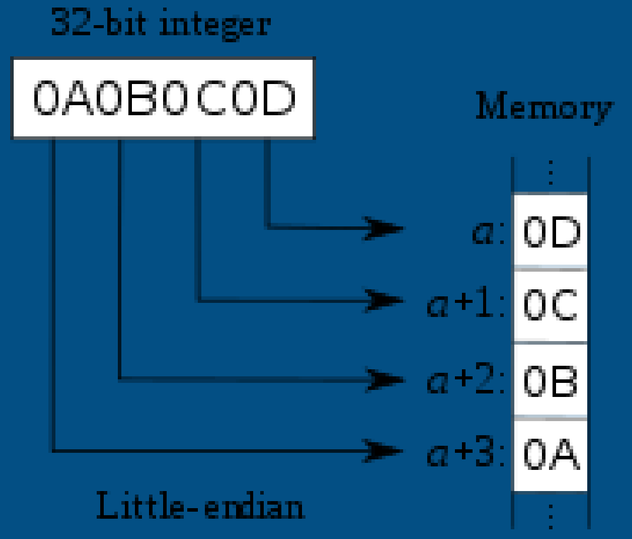
- std::span을 사용한 예제

```
std::vector<std::string> greeting = { "hello", "world" };
std::span<std::string> span(greeting);
for (auto&& s : span)
{
    s[0] = std::toupper(s[0]);
}
for (const auto& word : greeting)
{
    std::cout << word << ' ';
}
```

std::endian

- 아키텍처에 따라 데이터를 메모리에 저장하는 순서가 다를 수 있습니다.
  예를 들어 숫자 '0x12345678'을 메모리에 저장할 때,
  - 빅 엔디언 (Big Endian) : 12 34 56 78
  - 리틀 엔디언 (Little Endian) : 78 56 34 12





std::endian

- C++에서 엔디언 변환을 하려면 함수를 직접 구현해야 합니다.

```
//! Byte swap unsigned int
uint32_t swap_uint32(uint32_t val)
{
    val = ((val << 8) & 0xFF00FF00) | ((val >> 8) & 0xFF00FF);
    return (val << 16) | (val >> 16);
}

//! Byte swap int
int32_t swap_int32(int32_t val)
{
    val = ((val << 8) & 0xFF00FF00) | ((val >> 8) & 0xFF00FF);
    return (val << 16) | ((val >> 16) & 0xFFFF);
}
```

https://stackoverflow.com/questions/2182002/convert-big-endian-to-little-endian-in-c-without-using-provided-func

std::endian

- 하지만 아키텍처마다 어떤 엔디언을 사용하는지 알아야 합니다.

```
#if defined(__BYTE_ORDER) && __BYTE_ORDER == __BIG_ENDIAN ||                \
    defined(__BIG_ENDIAN__) || defined(__ARMEB__) || defined(__THUMBEB__) || \
    defined(__AARCH64EB__) || defined(_MIBSEB) || defined(__MIBSEB) ||      \
    defined(__MIBSEB__)
// It's a big-endian target architecture
#elif defined(__BYTE_ORDER) && __BYTE_ORDER == __LITTLE_ENDIAN ||           \
    defined(__LITTLE_ENDIAN__) || defined(__ARMEL__) ||                     \
    defined(__THUMBEL__) || defined(__AARCH64EL__) || defined(_MIPSEL) || \
    defined(__MIPSEL) || defined(__MIPSEL__)
// It's a little-endian target architecture
#else
#error "I don't know what architecture this is!"
#endif
```

https://stackoverflow.com/questions/2182002/convert-big-endian-to-little-endian-in-c-without-using-provided-func

std::endian

- std::endian을 사용하면 문제없습니다.

```
template <typename T, std::endian endianess = std::endian::native>
std::string HEX__1(const T & value, size_t value_size = sizeof(T)) {
    using namespace std;
    uint8_t* buffer = (uint8_t*)(&value);
    char converted[value_size * 2 + 1];
    if (endianess == std::endian::big)
        for (size_t i = 0; i < value_size; ++i) {
            sprintf(&converted[i * 2], "%02X", buffer[i]);
        }
    else
        for (size_t i = 0; i < value_size; ++i) {
            sprintf(&converted[i * 2], "%02X", buffer[value_size - 1 - i]);
        }
    return converted;
}
```

https://gist.github.com/ivanstepanovftw/9159efe123140c45d9df02e3d0454152

std::bit_cast

- 어떤 값을 비트로 캐스팅합니다.

```
constexpr double f64v = 19880124.0;
constexpr auto u64v = std::bit_cast<std::uint64_t>(f64v);
constexpr std::uint64_t u64v2 = 0x3fe9000000000000ull;
constexpr auto f64v2 = std::bit_cast<double>(u64v2);

int main() {
    std::cout
        << std::fixed << f64v << "f64.to_bits() == 0x"
        << std::hex << u64v << "u64\n";

    std::cout
        << "f64::from_bits(0x" << std::hex << u64v2 << "u64) == "
        << std::fixed << f64v2 << "f64\n";
}
```

https://en.cppreference.com/w/cpp/numeric/bit_cast

Calender and timezone library

- C++11 표준에 <chrono> 라이브러리가 추가되었습니다.
  하지만 시간을 측정할 때 이외에는 딱히 사용하지 않았습니다.
- C++20의 <chrono> 라이브러리에는 많은 기능이 추가됩니다.
  - 시간 관련 클래스 추가
  - 캘린더 관련 클래스 추가
  - 표준 시간대 관련 클래스 추가

Calender and timezone library

- 시간 관련 클래스
  - UTC(협정 시계시) 시간 : `utc_clock`
    - `system_time <-> utc_time`을 위해 `from_sys/to_sys` 함수 지원
  - TAI(국제 원자시) 시간 : `tai_clock`
    - `utc_time <-> tai_time`을 위해 `from_utc/to_utc` 함수 지원
  - GPS 시간 : `gps_clock`
    - `utc_time <-> utc_time`을 위해 `from_utc/to_utc` 함수 지원
  - 시간 변환 함수 : `clock_cast`

Calender and timezone library

- 시간 관련 클래스 예제

```
using sys_days = std::chrono::time_point<
                 std::chrono::system_clock, std::chrono::days>;
sys_days st(1997_y / dec / 12);
auto ut = utc_clock::from_sys(st);
auto tt = tai_clock::from_utc(ut);
auto gt = gps_clock::from_utc(ut);

assert(clock_cast<sys_clock>(st) == st);
assert(clock_cast<utc_clock>(ut) == ut);
assert(clock_cast<tai_clock>(tt) == tt);
assert(clock_cast<gps_clock>(gt) == gt);
```

Calender and timezone library

- 캘린더 관련 클래스
  - 년/월/일 표현 : `year_month_day`
    - 다음 세 순서로 사용할 수 있습니다 (년/월/일, 월/일/년, 일/월/년)
  - 캘린더 편의 기능 지원
    - `year_month_day_last` : 어떤 년/월의 마지막 일을 저장하는 클래스
    - `month_day_last` : 어떤 월의 마지막 일을 저장하는 클래스
    - weekday_last : 어떤 월의 마지막 평일을 저장하는 클래스
    - weekday_indexed : 어떤 월의 n번째 평일을 저장하는 클래스

Calender and timezone library

- 캘린더 관련 클래스 예제

```
auto d1 = 2018_y / mar / 27;
auto d2 = 27_d / mar / 2018;
auto d3 = mar / 27 / 2018;
year_month_day today = floor<days>(system_clock::now());

assert(d1 == d2);
assert(d2 == d3);
assert(d3 == today);
```

https://mariusbancila.ro/blog/2018/03/27/cpp20-calendars-and-time-zones/

Calender and timezone library

- 캘린더 관련 클래스 예제

```
auto today = year_month_day{
floor<days>(system_clock::now()) };
auto ymdl = year_month_day_last(today.year(),
month_day_last{ month{ 2 } });
auto last_day_feb = year_month_day{ ymdl };

assert(last_day_feb == 2018_y / feb / 28);  // for 2018
```

https://mariusbancila.ro/blog/2018/03/27/cpp20-calendars-and-time-zones/

Calender and timezone library

- 캘린더 관련 클래스 예제

```
inline int number_of_days(sys_days const& first,
                          sys_days const& last)
{
    return (last - first).count();
}

auto days = number_of_days(2018_y / apr / 1, 2018_y / dec / 25);

assert(days == 268);
```

https://mariusbancila.ro/blog/2018/03/27/cpp20-calendars-and-time-zones/

Calender and timezone library

- 표준 시간대 관련 클래스
  - 주요 함수
    - `locate_zone` : 이름에 기반해 표준 시간대를 찾는 함수
    - `current_zone` : 현재 로컬 컴퓨터의 표준 시간대를 찾는 함수
  - 기타 구조체 및 함수
    - `sys_info, local_info`
    - `zoned_traits, zoned_time`
    - `choose, leap, link`
    - `nonexistent_local_time`
    - `ambiguous_local_time`

Calender and timezone library

- 표준 시간대 관련 클래스 예제

```
auto time = floor<std::chrono::milliseconds>(system_clock::now());
std::cout << std::left << std::setw(25) << std::setfill(' ')
          << "Time"
          << time << std::endl;

auto localtime =
zoned_time<std::chrono::milliseconds>(date::current_zone(), time);
std::cout << std::left << std::setw(25) << std::setfill(' ')
          << "Local"
          << localtime << std::endl;
```

https://mariusbancila.ro/blog/2018/03/27/cpp20-calendars-and-time-zones/

Calender and timezone library

- 표준 시간대 관련 클래스 예제

```
auto zone_names = {
    "Asia/Tokyo",
    "Asia/Hong_Kong",
    "Europe/Bucharest",
    "Europe/Berlin",
    "Europe/London",
    "America/New_York",
    "Pacific/Honolulu",
};

for(auto const& name : zone_names)
    std::cout << std::left << std::setw(25) << std::setfill(' ')
              << name
              << zoned_time<std::chrono::milliseconds>(name, localtime)
              << std::endl;
```

https://mariusbancila.ro/blog/2018/03/27/cpp20-calendars-and-time-zones/

다루지 않은 C++20의 새 기능

- Synchronized output
- `char8_t`
- `[[likely]]`, `[[unlikely]]` attribute
- `[[no_unique_address]]` attribute
- `constexpr`
  - `virtual` functions
  - `dynamic_cast`
  - `typeid`
  - `try-catch`

다루지 않은 C++20의 새 기능

- std::remove_cvref
- std::atomic_ref
- std::atomic_shared_ptr
- std::atomic_weak_ptr
- std::destroying_delete

C++20에 추가될 수도 있는 기능

- Expansion statements
- `constexpr std::vector`
- `std::format`
- `std::source_location`
- `std::flatmap, std::flatset`
- `std::byteswap`
- `using enum`

정리

- C++20 표준화 작업이 막바지 단계를 향해 가고 있습니다.
- C++20이 도입되면 C++11 또는 그 이상의 변화가 예상됩니다.
- 소개한 기능 외에도 다양한 추가 / 변경 사항들이 있습니다.
- 새 기능이 추가되었다고 무조건 다 알아야 할 필요는 없습니다.
- 모던 C++의 기능을 무조건 사용하기보다는,
  어떤 기능인지 파악하고 필요한 곳에 적절히 사용합시다.

감사합니다

http://github.com/utilForever

질문 환영합니다!