→ Boiler plate Comparison Codes:

```cpp
class Mint {
public:
    Mint() = default;
    Mint(int x) :m_val{x} {}

    friend bool operator<(const Mint& m1, const Mint& m2)
    {
        return m1.m_val < m2.m_val;
    }

    friend bool operator==(const Mint& m1, const Mint& m2)
    {
        return m1.m_val == m2.m_val;
    }

private:
    int m_val{};
};

inline bool operator>(const Mint& m1, const Mint& m2)
{
    return m2 < m1;
}

inline bool operator>=(const Mint& m1, const Mint& m2)
{
    return !(m1 < m2);
}
```

Bu sadece == ve <=> için geçerli

→ Derleyiciye default ettirebiliriz. Böylelikle rewrite yapabiliyor oluruz. m==5 → 5==m (cpp 20)

```cpp
class Mint {
public:
    Mint() = default;
    Mint(int x) :m_val{x} {}
    bool operator==(const Mint&)const = default;

private:
    int m_val{};
};
```

Artık bozan durum olmadığı sürece, constexpr + noexcept

⇒ Ayrıca default ettirince != 'de geldi!

```cpp
class Mint {
public:
    Mint() = default;
    Mint(int x) :m_val{x} {}
    bool operator==(const Mint&)const = default;

private:
    int m_val{};
};

int main()
{
    Mint m1{35}, m2{34};
    auto b = (m1 == m2);

    std::cout << "b = " << b << "\n";
}
```

yazılma sırasına göre karşılaştırma yapan bir fonksiyon yazdı bizim için İki nesnenin m_val'i

→ false

```cpp
struct Data {
    int a{}, b{}, c{};
    bool operator==(const Data&)const = default;
};

int main()
{
    Data x, y;

    auto b = x != y;
    //auto b = !(x == y);
}
```

Derleyici default ettirdiğimiz ifadeyi kullanarak, != 'i çıkarttı ⇒ rewrite etti.

```cpp
nclude <iostream>
nclude <compare>
nclude "nutility.h"


truct Data {
    Data() = default;
    Data(int x) : a{x}, b{x}, c{x} {}
    int a{}, b{}, c{};
    bool operator==(const Data&)const = default;
};


int main()
{
    using namespace std;

    Data x(5);

    cout << boolalpha << (5 == x) << "\n";

}
```

→ argüman olarak Data alsa bile, yani kurallar gereği, 5==x yazımı hala değil.

legal olmadığını anladı

5==x'i x==5 gibi

- Const olabilir edilmez rewrite etti. Ayrıca implicit conversion oldu 5, obje türüne dönüştü.

- Ayrıca != 'de geçerli

→ Spaceship Operator:

→ rewriteable and defaultable

→ Threeway comparison operatörü 3 farklı türde değer üretebilir.

→ Strong Ordering
→ Weak Ordering
→ Partial Ordering

=> bunlar hem primitive hem de user defined türler için geçerli

- Strong ordering: - ya büyük - ya küçük - ya da eşit

- Weak ordering: - ya büyük - ya küçük - ya eşit - ya da equivalence söz konusu

```cpp
int main()
{
    using namespace std;

    string  x{"zeynep"}, y{"volkan"};


    auto b = x <=> y;

    if (b == strong_ordering::equal) {
        std::cout << x << " esit " << y << "\n";
    }
    else if (b == strong_ordering::greater) {
        std::cout << x << " buyuk " << y << "\n";;
    }

    else if (b == strong_ordering::less) {
        std::cout << x << " kucuk " << y << "\n";
    }
```

→ lexicographical compare → zeynep büyük volkan

→ Fakat comparison işlemini bu şekilde uzun uzun yapmaktansa (x<=>y) > 0 gibi yazabiliriz!

strcmp mantığında

→ spaceship'in default edilmesi:

```cpp
class Point {


public:
    auto operator<=>(const Point&)const = default;
private:
    int mx{};
    int my{};
    int mz{};
};

int main()
{
    Point p1, p2;

    //p1 < p2
    (p1 <=> p2) < 0
}
```

auto'yu kullanma nedenimiz, ordering sırası, türe göre değişir ve tür belli değil

hepsi int olduğu için auto = strong_ordering ama double olsaydı partial_ordering

← yerine atttıklarımız gibi

→ Şimdi artık, tüm operatörlere sahibiz,
→ Ayrıca gerekirse yine rewrite edilebilir.

```cpp
class Point {


public:
    std::strong_ordering  operator<=>(const Point& other)const
    {
        auto result = mx <=> other.mx;
        if (result != std::strong_ordering::equal)
            return result;

        result = my <=> other.my;
        if (result != std::strong_ordering::equal)
            return result;

        return mz <=> other.mz;
    }
private:
    int mx{};
    int my{};
    int mz{};
};


int main()
{
    Point p1, p2;

    p1 < p2;
    p1 <= p2;
    p1 > p2;
    p1 >= p2;


}
```

eğer default etmek yerine bir yöntem

bu ifadeler legal olur

Ancak == 'i kullanmalıyız.

=='i implement etmez

```cpp
class Nec {
public:
    constexpr Nec(long i) noexcept
        : m_val{ i } {
    }

    // == ve != için
    [[nodiscard]] bool operator==(const Nec& other)const
    {
        return m_val == other.m_val;
    }

    // < <= > >= için
    auto operator<=> (const Nec& rhs) const
    {
        return m_val <=> rhs.m_val; // defines ordering (<, <=, >, and >=)
    }
private:
    int m_val;
    //...
};
```

eğer siz

== operatorunu default ederseniz
!=

< <= > >=

<=> default edersek → tüm operatörler var!

<=> kendimiz tanımlarsak

<
<=
>

*Örnek:

```cpp
int main()
{
    int x = 12;
    int y = 20;

    std::cout << std::boolalpha;
    std::cout << (std::strong_ordering::greater > 0) << '\n';     //TRUE
    std::cout << (std::partial_ordering::unordered > 0) << '\n'; //FALSE
    std::cout << (std::partial_ordering::unordered < 0) << '\n'; //FALSE

    std::cout << (std::strong_ordering::less > 0) << '\n';     //FALSE
    std::cout << (std::strong_ordering::less == 0) << '\n';    //FALSE
    std::cout << (std::strong_ordering::less != 0) << '\n';    //TRUE
    std::cout << (std::strong_ordering::less >= 0) << '\n';    //FALSE
}
```

unordered'ın 0'ile karşılaştırması! → compile yanlış!

*lexicographical_compare function: → algorithm başlık dosyasında

```cpp
#include <iostream>
#include <compare>
#include <algorithm>
#include <vector>
#include <list>

int main()
{
    using namespace std;

    vector<int> ivec{ 3, 7, 9, 12 };
    list<int> ilist{ 3, 7, 9, 5 };

    //ivec < ilist → bu sırada hatalı
    lexicographical_compare(ivec.begin(), ivec.end(), ilist.begin(), ilist.end())
}
```

→ Böylelikle 2 farklı container'ın içerisine lexicographical olarak bakabiliyoruz.

*Regex Kütüphanesi:

→ regular expression'ın kısaltması

→ yazılıda ilgili işlemlerin (tokenize, find gibi) bir kısmını standard kütüphane ile yapabilirken, bazı işlemler standart kütüphane ile yapmak ya çok karmaşık, ya da time efficient değil.

→ Aslında bir string üzerinde kural belirliyoruz. String bir email mi?, String bir posta kodu mu? ... + komut satırı, C, C++ fonksiyonun tanımları gibi...
String bir date mi?

*regex grammar: regex string'in kuralı nasıl belirlenir / belirlenir?

```
regex string

1. validation / matching

2. search

3. tokenizing

4. replace
```

→ regex string ile yapılabilen işlemler.

* Regex Grammar: → Noktasını öğrenmek için regex101.com

→ Meta characters:

.	Any character except newline
[...]	One of the characters ... (may contain ranges)
[^...]	None of the characters ... (may contain ranges)
[[:*charclass*:]]	A character of the specified character class *charclass* (see Table 14.4)
\n, \t, \f, \r, \v	A newline, tabulator, form feed, carriage return, or vertical tab
\x*hh*, \u*hhh*	A hexadecimal or Unicode character
\d, \D, \s, \S, \w, \W	A shortcut for a character of a character class (see Table 14.4)
*	The previous character or group any times
?	The previous character or group optional (none or one times)
+	The previous character or group at least one time
{*n*}	The previous character or group *n* times
{*n*,}	The previous character or group at least *n* times
{*n*,*m*}	The previous character or group at least *n* and at most *m* times
...|...	The pattern before or the pattern after |
(...)	Grouping
\1, \2, \3, ...	The *n*th group (first group has index 1)
\b	A positive word boundary (beginning or end of a word)
\B	A negative word boundary (no beginning or end of a word)
^	The beginning of a line (includes beginning of all characters)
$	The end of a line (includes end of all characters)