\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

saat 21:00-22.00

->iki string arasında +, += çalışıyo ama -, -= çalışmıyo

->string "[]" (stringde overloadlanmış subscript operatörü) boundary check yapmıyor(stringin uzunluğunu aştın mı uyarı vermiyor):

string x = "alaz"; x[15] = "a" gibi bir işlem yapmamaya dikkat etmem lazım

x.at(15); yazınca out of range exception atıyor(programi crashliyor ve hatayi yazıyor)

-> yani string.at() operatörü boundary check yapıyor.

->c++ yeni operatör oluşturmaya izin vermiyor, fakat var olan operatörleri overloadlamaya izin veriyor

->operatör overloading otomatik değil, kendim yapmam lazım

->classlarda overloaded operatör function kullanmak için bu overloaded operatörleri definelamam lazım, şu 3üne dikkat:

**operator=** ----> "=" i overloadlerken "="in sağındaki objenin datalarını "="in solundaki objenin içine kopyalarız. fakat eğer classta pointer varsa, bu kopyalama tehlikeli olabilir(shallow copy). yani kopyalarken sadece sağ objedeki pointerin işaret ettiği adres sol objenin pointerine kopyalanırsa, bir objenin pointeri üzerinde yaptığımız işlem diğer objede de uygulanmış olur. kısaca dinamik bellek yönetiminde problem oluşturur.

**operator(&)** ----> objeyi point eden bir pointer returnler // address operator

**operator(,)** ----> virgül operatörü önce virgülün solundaki ifadeyi değerlendirir, sonrasında sağı değerlendirir ve sağdaki ifadenin sonucunu geri döndürür:

*int x = (5,10); // output: x = 10* a

-> overloadlanamayan operatörler:

“.” “.\*”(pointer to member) “::” “ ?:”

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Saat 22.00-23.00

-> “+” operatörünü overloadlayarak iki int sayıyı toplamak yerine çıkartma işlemi yapmasını sağlayamam. Operatör aşırı yükleme, yalnızca kullanıcı tanımlı türler (class, struct) veya temel türler(int, string) ile kullanıcı tanımlı türlerin karışımları için geçerlidir.

**-> Bazı operatörler sadece class member fonksiyonu olarak overloadlanabilir:** Eğer **(), [], -> veya herhangi bir atama operatörü (= gibi)** overloadlanacaksa **bu operatörler classın member fonksiyonu olarak tanımlanmak zorundadır**. Diğer overloadlanabilir operatörler hem class member hem de global(non-member) fonksiyonlar olarak tanımlanabilir.

-> **Operatörün aldığı operand sayısı değiştirilemez:** overloadlanan bir operatör, aldığı operand sayısını değiştiremez. Yani, tekil (unary) operatörler tekil kalır, ikili (binary) operatörler ikili kalır. &, \*, +, ve - operatörleri hem unary hem de binary versiyonları olduğu için her iki versiyon da ayrı ayrı overloadlanabilir:

***&, \*, +, ve – operatörlerinin binary ve unary overloadlanmış helleri “mainPCcalisma” projesinde var.***

Unary (&): adres döndür, **myClass\* operator&()** şeklinde yazılır.

Binary (&): ne yaptığını bilmiyorum

Unary (\*): dereference operatörü, **int operator\*()** şeklinde yazılabilir fakat fundamental typesi int olmak zorunda değil, classtaki dataya göre değişebilir. Bu operator display fonksiyonunun farklı bir versiyonu gibi de düşünülebilir.

Binary (\*): çarpma işlemi, classın dataları içinde çarpma işlemi yapar. **myClass operator\*(const myClass& obj)** şeklinde yazılabilir.

Unary (+): değer neyse aynen döner, bir işlem yapmaz. **myClass operator+()** şeklinde yazılabilir. (implementasyonunda sadece **return \*this;** yazıyor.)

Binary (+): normal toplama işlemi.

Unary (-): girilen değer neyse negatife döndürür**. myClass operator-()** şeklinde yazılır. Implementationunda **return myClass(-value);** yazar.

Binary (+): normal çıkarma işlemi.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Saat 23.00-00.00

**-> Member Function Olarak Overloadlanan Binary Operatörler:**

Bir String **(kullanıcı tarafından oluşturulmuş bir class)** classı tanımladığınızı ve iki String nesnesini < operatörü ile karşılaştırmak istediğinizi düşünün. < operatörünü, String sınıfı içinde bir member fonksiyon olarak overloadladığınızda, y < z ifadesi, aslında y.operator<(z) olarak değerlendirilir. Yani sol operand (y) String classına ait bir obje olduğu için bu fonksiyon, z objesini argüman olarak alır ve < operatörünü uygular.

Bu durumda, **binary operatörler member fonksiyonlar olarak sadece sol operand sınıfın bir nesnesi olduğunda** tanımlanabilir. Yani y < z ifadesinde sol taraftaki y objesi mutlaka String classına ait olmalıdır.

**bool operator< (const String& str)**

**-> Global/Non-member Fonksiyonlar Olarak Overloadlayan Binary Operatörler:**

Binary operatör < bir non-member/global fonksiyon olarak tanımlandığında, iki argüman almalıdır. Bu argümanlardan biri mutlaka String classının bir objesi (veya referansı) olmalıdır. Yani, y < z ifadesi, programda aslında operator<(y, z) şeklinde değerlendirilir ve iki String objesini veya referanslarını karşılaştıran bir fonksiyon çağrısı yapılır.

**bool operator<(const String &y, const String &z);**

-> **Stream insertion(<<) ve stream extraction(>>) operatörlerinin yazımı:**

**ostream& operator<<( ostream& os, const myClass& obj) {  
 os << obj.value;  
 return os;  
}**  
  
**istream& operator>>( istream& is, myClass& obj) {  
 is >> obj.value;  
 return is;  
}**

***Main’e yazılanlar:***

**Compiler’in gördüğü: cout << obj;**

**Compiler’in çağırdığı: operator<<( cout, obj );**

**Compiler’in gördüğü: cin >> obj;**

**Compiler’in çağırdığı: operator>>( cin, obj );**

>> operatörü, istream referansı (örneğin, cin) döndürür. Bu, myClass nesneleri üzerinde giriş işlemlerinin diğer myClass nesneleri veya başka veri tipleriyle birlikte cascadelenmesine(zincirlenmesine) olanak tanır. Örneğin, bir program iki myClass nesnesine tek bir ifadeyle şu şekilde veri girişi yapabilir:

cin >> obj1 >> obj2;

İlk olarak, cin >> obj1 ifadesi çalışır ve bu, aşağıdaki gibi bir fonksiyon çağrısı yapar:

operator>>( cin, obj1 );

Bu çağrı, cin'i referans olarak döndürür. Böylece cin >> obj1 ifadesi cin'i döndüğü için, ifadenin kalan kısmı sadece cin >> obj2 olarak değerlendirilir. Bu, şu şekilde başka bir üye olmayan fonksiyon çağrısı yapar:

operator>>( cin, obj2 );

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Saat 00.40-01.20

<< operatörünü overloadlamak için bir ostream referansı ve bir const myClass referansı alıp ostream referansı döndüren bir fonksiyon tanımlanır. Bu, myClass türündeki nesnelerin ekrana yazdırılmasını sağlar.

Örneğin, cout << obj ifadesi kullanıldığında, derleyici aşağıdaki gibi bir üye olmayan fonksiyon çağrısı oluşturur:

operator<<(cout, obj);

Bu fonksiyon, obj nesnesinin valuesini ekrana yazdırır.

operator<< fonksiyonunun parametresinde myClass referansı const olarak tanımlanır çünkü myClass nesnesi sadece ekrana yazdırılacak ve nesneyi değiştirmeyecek.

operator>> fonksiyonunun parametresinde myClass referansı non-const olarak tanımlanır çünkü myClass nesnesi, kullanıcıdan alınan girdi değerini saklamak için değiştirilmelidir.

**-> Stream Insertion(<<) ve Extraction(>>) Operatörleri Neden Non-member fonksiyon biçiminde overloadlanır?**

Overloadlanmış stream insertion operatorü(<<)’nün sol operandı ostream typedır. Bir operatörün member fonksiyon olabilmesi için sol operandın kullanıcının yaptığı class type’ında olması lazım. << operatöründe böyle bir durum olmadığı için bu operator non-member fonksiyon olarak yazılmak zorunda. Aynı şekilde extraction operatörü(>>)’nün de sol operandı istream type olduğu için bu operatörde non-member fonksiyon olarak yazılmalı.

**-> Unary(Tekil) Operatörleri Overloadlama**

1- Member Fonksiyon olarak overloadlanabilirler. Bu durumda hiçbir argüman almaz.

2- Non-member fonksiyon olarak overloadlanabilir. Bu durumda tek bir argüman alır. Bu argüman, classın bir nesnesi veya nesnenin referansı olmalıdır. Bu tür bir fonksiyon, classın private datalarına erişemez.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Saat 06.30-08.00

**-Member Fonksiyon olarak Unary(Tekil) Operatörleri Overloadlama:**

**->** ! operatörü Boolean bir değer döndürür. “!” ile yazılan Boolean expression true ise false, false ise true döner. Kullanıcı tarafından overload edileceği zaman class objesi boş mu dolu mu diye bakmak için kullanılabilir. Member fonksiyon olarak overloadlamak isteniyorsa ! operatörü herhangi bir argüman almaz.

bool operator!() const{  
 if (value == 0)  
 return false;  
 else  
 return true;  
}

**Compiler’in çağırdığı fonksiyon:**

operator!()

Bu overloaded member fonksiyon class’ın valuesi 0’sa false returnler, 0 değilse true returnler.

**-Non-member Fonksiyon olarak Unary(Tekil) Operatörleri Overloadlama:**

**->** ! operatörü non-member fonksiyon olarak overloadlanmak sadece 1 argüman alır.

bool operator!( const myClass &obj );

**Compiler’in çağırdığı fonksiyon:**

operator!(obj);

**-Prefix ve Postfix increment(++) operatörlerini overloadlama:**

**->** Compilerin prefix ve postfix operatörlerini ayırt edebilmesi için farklı imzaları(declare edilme şekilleri) olmalı. ++operatörünün overloadlanma biçimi –operatörünün de overloadlanma biçimiyle aynıdır.

***Prefix Operatörünü Overloadlama:***

**-**Diyelimkibir myclass objesi olan obj1’in valuesini 1 arttırmak istiyoruz. Compiler **++obj1** kodunu gördüğü zaman obj1.operator++() member fonksiyonunu çağırır. Bu member fonksiyonunun prototipi şu şekildedir:

myClass& operator++();

**-**Eğer operator non-member olarak overload edildiyse ++obj1 yazıldığında Compiler operator++(obj1) non-member fonksiyonunu çağırır. Bu fonksiyonun prototipi de şu şekildedir:

myClass& operator++(myClass& obj);

-Prefix ++ operatörünün return typesinin myClass& şeklinde olmasının sebebi **cascaded(zincirleme)** işlem yapılabilmesini sağlamak içindir. Çünkü herhangi bir int variablede

int x = 5;

++(++x);

Işlemini yapabilmek mümkündür. Return type **reference return** olunca myClass objesi için de **cascaded** işlem yapılması sağlanır.

***Postfix Operatörünü Overloadlama:***

**-** Compiler’ın prefix ++operatörünü postfix ++operatöründen ayırt edebilmesi için argüman olarak bir dummy int değeri atanır( Parametreye yazılır). Compiler obj1++ operatörünü gördüğünde:

obj1.operator++( 0 );

Member fonksiyonunu çağırır. Postfix ++operatörünün prototipi şu şekildedir:

myClass operator++(int);

***Buradaki 0 argümanı compilerin postfix’i prefixten ayırt edebilmesi için atanmış bir dummy değerdir***. Aynı yazılış biçimi non-member fonksiyonlar için de geçerlidir. Postfix non-member fonksiyon olarak overload edilmişse, obj1++ yazılınca compiler:

operator++(obj1, 0);

Non-member fonksiyonu çağırır. Bu fonksiyonun protipi şu şekildedir:

myClass operator++(myClass& obj, int);

0 yine burada da prefix ile postfixi ayırt etmek için atanmış bir dummy değerdir. *Bu operatörün prefixten diğer bir farkı ise return type olarak value return yapar.* Bunun sebebi postfix ++operatörü diğer data typelerde de cascaded(zincirleme) işlemlere izin vermez. Bu yüzden *return type’yi reference return yapmaya gerek YOK*tur.

**Prefix ve Postfix++ operatörünün define edilmiş halleri:**

***Prefix:***

myClass& operator++() {  
 ++value;  
 return \*this;  
}

***Postfix:***  
myClass operator++(int) {  
 myClass temp = \*this;  
 ++value;  
 return temp;  
}

**Prefix ve Postfix-- operatörlerinin ++ operatörlerinden herhangi bir farkı yoktur. Tek yapılan ++value yerine --value yazmaktır.**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Saat 11.00-12.00

İzlediğim videodan öğrendiğim ek bilgiler:

Bir class’a static variable eklenebilir. Eğer bir class’ta herhangi bir variable varsa, her bir class objesinin kendisine özel variablesi vardır. Fakat eğer class’ta static variable varsa, bu variable ortak bir variabledir, yani bütün objeler bu variableye erişebilir, üzerinde değişiklik yapabilir. Bu variableler bu yüzden “static” keyword ile belirtilir.

static int x; // Class’in içine yazılır (privateye ya da publice)

int myClass::x = 0; // Class’ın dışında definelanır, ***.cpp’ye yazılır!!!!***

Classlara “static fonksiyon” da eklenebilir. Bu statik fonksiyonlar, herhangi bir class objesi olmasa bile eğerki classta static bir variable varsa bu variableye güncellemeyi ya da displaylemeyi sağlar.

static void updateShared(int i) { shared = i; }  
static int getShared() { return shared; }

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

2. gün Saat 23.00-00.00

Friend’in 3 farklı typesi var: class, class member fonksiyon, herhangi bir düz fonksiyon(class içinde olmayan). Friendler class’ın private kısmına erişim sağlar

->Class member fonksiyonu başka bir class’a friend yaparken: (mainPCcalisma2’ye bak)

friend datatype Classtype::fonksiyon(…);

->Non-member fonksiyonu friend yaparken:

friend datatype fonksiyon(…);

->Başka bir class’ı friend yaparken:

friend class classname;