lessons 17 - 20

LESSON 17

delegating constractor ex:

```
class my_class{
public:
my_class(int x) : my_class(x,x)   // constractor bir başka const
{}
my_class(int x , int y)
{}
my_class(const char*x) : my_class(std::atoi(x))
{}
private:
}
```

friend : private ile erişim engeli olan sınıflara erişim hakkı verir. çoğu zaman sınıfın kendi kodlarına veriliyor. örneğin; global fonksiyonlarına, yardımcı türlere

(ADL: argument dependant lookup) ilede ilgili, ilerde görcez.

```
class my_class{
public:

  friend void my_class_support_function(my_class);  // bu sayede

  friend void my_class_support_function2(my_class)   // istisnai
  {
```

```
  }

private:

  int mx{};
  void foo();
}

void my_class_support_function(my_class m)
{
   m.mx = 5;   // NORMAL de error, friend sayesinde erişti
}
```

Sınıf kendi üye fonksiyonunu friend yapamaz.

```
class my_class{
 public:
   void foo();   // my_class2 nin x ine ulaşabilir friend sayesi
}

class my_class2{
 private:
   friend void my_class::foo();  // constractor bile friend olu
   int x;
}
```

```
class my_class{
 public:
   void foo();
}

class my_class2{
 private:
   friend class my_class;  // my_class , my_class2 nin herseyine
```

```
    int x;
}
```

A B ye friend dese

B C ye friend dese

A C YE ERİŞEMEZ !

OPERATOR OVERLOADİNG

operator kullanarak fonksiyon çağırmak.(operator fonksiyonu) . runtime maliyeti yok. compile time da.

operator function: free(global) function ve non-static member function olmalı

```
int main(){
   /** hepsi operator overloading **/
   std::String name{"can"};
   name += "test";
   if(name == "cantest"){}
   auto str = name + "123" + "456";
   (name[1] == 'a')
   cout << name;
}
```

kurallar;

- isim operator keyworldü ve o tokenı yazcaz ex: “operator ++”
- c++ da olmayan operatorler overload olmaz ex: “@”
- operandlardan en az biri class olmalı
- bazı operatorlerin overload edilmesi yasak. ex: “.” “? :” “::” “.*”(c de yok c++ da var)

- global fonksiyonlar bunları overload edemez. “[]”,”→”,”()”. **class member func lar yapabilir**,
- operator öncelik seviyesi ve yönü değişmez.
- operatorlerin “arity” si değişmez. “arity” operatorün unary yada binary olması

!x → unary operator , “!” tek operand alıo

a > b → binary operator,”>” sağı solu dolu 2 operant alıo

global operator functions: örneğin “>” için 2 değer almak zorunda. 1 değil 3 değil 2. çünkü yukardaki mantık.

ex; a> b → operator>(a,b) oluyor yani global için

ve aynı şekilde “!” de bir değer almak zorunda. normalde nasıl kullanıosan öyle yani.

ex !a; → operator!(a);

member class functions: unary için 0 parametre olmalı. parametre değişkeni olmamalı.

ex: !x; → x.operator!();

binary arity için ise 1 parametre olur member functionsda.

ex: a > b; → a.operator>(b);

```
class my_class{

};

my_class a , b;

a = b;
a.operator=(b);  // bu ve a=b aynı , derleyici buna dönüştürüyo
```

NOT: c++ insights , bu siteden derleyicinin ürettiği kodu görebilirsin.

https://cppinsights.io/

Derleyici optimizasyonu ve yani çevrilen asm kodu için :

https://godbolt.org/

```
class my_class{
public:
};
/** global operator functions **/
my_class operator*(const my_class& , const my_class&);
my_class operator+(const my_class& , const my_class&);
my_class operator/(const my_class& , const my_class&);
my_class operator-(const my_class& , const my_class&);

int main(){
   my_class m1, m2 , m3 , m4;

   auto m = m1 * m2 + m3 / m4;  // c++ insights bakabilirsin der
}
```

DERS 18

```
class my_class{
public:
    /** member functions **/
    my_class operator*(const my_class&  )const; // bu 2. const s
    my_class operator+(const my_class& )const;
    my_class operator/(const my_class&  )const;
    my_class operator-(const my_class&  )const;
};
```

```
int main(){
   my_class m1, m2 , m3 , m4;

   auto m = m1 * m2 + m3 / m4;  // c++ insights bakabilirsin der
}
```

```
class my_class{
public:
    /** member functions **/    // hem operator overload hem fur
    my_class operator+(const my_class& )const;   // x + b
  my_class operator+()const;                  // +x
};
```

```
class my_class{
public:
    /** member functions **/
    my_class operator+(const my_class& )const;
};
int main(){
   my_class m1 ;

   auto m = m1 + 5;  // doğru
   auto m = 5 + m1;  // hata
}
```

global operator loading nerde kullanılır, örnek;

```
class matrix{

}

std::stream& operator<<(std::ostream& os , const matrix&);// bur
```

```
int main(){

  matrix m;

  std::cout << m ;
}
```

return types of operator functions: mantıklı olan genelde nesnenin kendisini dönmek.

const coorectness & operator functions: eğer operator function ilgili nesneyi değiştirme durumu olmazsa algoritma içinde, const yap. Ki genelde const yapılmalı.

value category: (x+y) R value. operator function returnunda & ile dönmek sıkıntı olabilir. Mesela a = b , x +=y , ++x R value yani & dönmesi mantıklı

Note: c++ 20 de == overload edince ! = de overload oluor.

Binary simetrik operatorler genelde global yapmak mantıklı ; a>b , b<c vb..

```
class counter{
   counter();
   counter(int);
   counter operator+(int)const;
}
void main(){
   counter a , b;
   b = a + 5;   // çalışır
   b = 5 + a;   // HATA
}

//************** ÇÖZÜM:************************

counter operator+(counter,int);    //global, problem şimdide bu
counter operator+(int,counter);
```

```
   b = a + 5;   // çalışır
   b = 5 + a;   // çalışır

********** DAHA İyisi

class counter{
   counter();
   counter(int);
   friend counter operator+(counter,int);
   friend counter operator+(int,counter); // namespaci değişti y
}
```

ATTANTION : std::cout << “slm” << endl ; endl nedir?

\n yazıo ve ekrana direk basıor. ve endl bir fonksiyon girdisi bir callback fp ex:

```
std::ostream & dashline(std::ostream & os){
   return os << "\n-----------------------\n"
}
int main(){
    cout << "test1" << dashline << "test2" << dashline;
}
```

std::ostream → << ile ; kullanıcı çıkışı printf gibi

std::instream → >> ile; kullanıcı girişi scanf gibi

Örnek, operator overload, kendi “int” imizi yapalım adı nint olsun:

```
class nint{
public:
```

```
    nint() = default;
    explicit nint(int x):mx{x}{}  // bu int girişine örtülü dönü

 // auto operator <=>(const nint&)const = default;  // C++20 biz

    friend std::ostream& operator<<(std::ostream& os, const nint
    {
        return os << '(' << nn.mx << ')'; // bu zaten hep zincir
        //return os;  // yukardakiyle aynı
    }

    friend std::istream& operator>>(std::istream& os, const nint
    {
        return os >> nn.mx ;
    }


    // karşılaştırma operatorleri iplemantasyonu:
    // NOT: c++20 de işler kolaylaşmış  "<=>" bunu operload et d

    // equality operators   : == , !=
    friend bool operator==(const nint& lhs , const nint& rhs) /
    {
        return lhs.mx == rhs.mx;
    }
/**
    bool operator==(const nint& other) const {          // BUNU
            return this->mx == other.mx ;
    }
**/

    friend bool operator!=(const nint& lhs , const nint& rhs) /
    {
       // return !(lhs.mx == rhs.mx); // olur
        return !(lhs == rhs);  // buda çalışır, bu zaten yukarda
```

```
}


// relational operators : > , < , >= , <=
friend bool operator<(const nint& lhs , const nint& rhs)
{
   return lhs.mx < rhs.mx;
}
//
friend bool operator>(const nint& lhs , const nint& rhs)
{
   return rhs < lhs; //  operator< çağrıldı , left right ter
}
friend bool operator<=(const nint& lhs , const nint& rhs)
{
   return !(rhs < lhs);  // aynı
}
friend bool operator>=(const nint& lhs , const nint& rhs)
{
   return !(lhs < rhs);   // aynı
}

// aritmatik operatorler: scott meyers önerisi: += + çağırca
nint& operator+=(const nint&other) // const değil , değişcel
{
    mx += other.mx;
    return *this;
}
// kalanlar aşağıda inline operator + vs

nint& operator-=(const nint&other) // const değil , değişcel
{
    mx -= other.mx;
    return *this;
}
```

```
// ++x ve x++ çok farklı KURAL: int DUMY ++x ve x++ derleyici ay
/**
   nint& operator++();     prefix   -> ++x   adres döner çünkü d
   nint operator++(int);   postfix  -> x++   bu int DUMY ,yani s
   nint& operator--();     prefix   -> --x
   nint operator--(int);   postfix  -> x--   bu int DUMY ,yani s

   // friendler için:
   friend nint operator++(nint& obj);       ->   ++x
   friend nint operator++(nint& obj, int);  ->   x++   aynı kura
**/
    nint& operator++(){
       ++mx;
       return *this;
    }
    nint operator++(int){   // bu parametreye(int) ASLA İSİM VER
       nint temp{*this};
       operator++();   // veya "++mx;" veya -> "++*this;"     ->
       return temp;
    }
    nint& operator--(){
       --mx;
       return *this;
    }
    nint operator--(int){
       nint temp{*this};
       operator--();
       return temp;
    }

  // +x ve -x -> x in değeri değişmez -> const ve R value dönmel
  nint operator+()const{
       return *temp;  // + hiç bişey değiştirmez
    }
  nint operator-()const{
       return nint(-mx);
```

```
    }

private:
    int mx{};
}

inline nint operator+(const nint&lhs , const nint&rhs)  {
    nint temp(lhs);
    temp += rhs; // scott meyers önersi, mantıklı !! yukarda zat
    return temp;

    return nint(lhs) += rhs;  // bu yukardaki 3 satır ile aynı
}

inline nint operator-(const nint& lhs , const nint& rhs)  {
    nint temp(lhs);
    temp -= rhs;
    return temp
}

int main(){

  using namespace std;

    for(i = 0 ; i < 100; i++){
       cout << nint(i) << '\n';
    }

  // tam sayı girme,
  nint n1,n2,n3;
  cin >> n1 >> n2 >> n3;
  cout << n1 << n2 << n3;

}
```

NOT : **çoğunlukla ++x yani prefix KULLAN, diğer librariyler için bazen maliyeti daha düşük olabilir.**

DERS 19

referance qualifiers : ilerde görcez, kısaca : “void lvalueFunction() & {//lvalue nesneler üzerinde çağrılabilir}” ve “void rvalueFunction() && {//rvalue nesneler üzerinde çağrılabilir ve move burda taşınır}”

operator[]: geri dönüş L value

ex: str[1] = ‘a’ → str.operator[](1)

```
class string{
public:
   string(const char* p):mp{new char[std::strlen(p)+1]}{
      std::strcpy(mp,p) ;
   }
   char& operator[](std:size_t idx)
   {
      return mp[idx];
   }
   char& operator[](std:size_t idx)const
   {
      return mp[idx];
   }
private:
   char* mp;
};

int main(){
   string s("selam"); // ama const string olsa olmaz mp const de
   str[1] = 'a';   // string const olsa -> char& operator[](std
   std::cout << str[1];
}
```

: * dereferencing/indirection operator

: . dot operator

: → arrow (ok operatorü binary op olmasına rağmen unary op gibi implement edilir , yani parametre değişkeni olmaz) geri dönüş pointer.

ex: operator overload ettin diyelim: "p→x" demek "p.operator→()→x" böyle yapıo derleyici **TUHAF ! bu yüzden dönüş pointer olmalı**

Neden, diyelim ki operator loading yok: " my_class *m; m→foo(); " bunu legal yapmak için.

```
class my_class{
}

class pointer_like{
public:
    explicit pointer_like(my_class*f):mp(f);
  ~pointer_like(){ if(mp){ delete mp;} }
  /** copy close **/
  pointer_like(const pointer_like&) = delete;
    pointer_like& operator = (const pointer_like&) = delete;

    my_class& operator*()    // *p
  {
     return *mp;
  }
    my_class* operator->()    // p->x
  {
    return mp;
  }
private:
  my_class *mp{nullptr};   // TODO: generic programlamada my_cla
}

// generic programlama örneği: ilgili T = my_class nin speciali

template <typename T>
class pointer_like{
public:
```

```
    explicit pointer_like(T*f):mp(f);
  ~pointer_like(){ if(mp){ delete mp;} }
  /** copy close **/
  pointer_like(const pointer_like&) = delete;
    pointer_like& operator = (const pointer_like&) = delete;

    T& operator*()    // *p
  {
     return *mp;
  }
    T* operator->()    // p->x
  {
    return mp;
  }
private:
  T*mp{nullptr};
}
```

DERS 20

Fonksiyon çağrı operatorü ⇒ **( ) - "function object" için**

MEMBER(üye) fonksiyon olmak zorunda. varsayılan argüman alabilir.

ex: foo(2,5); → foo.operator()(2,5);

```
class my_class{
   void operator()(){            // int operator()(int x){   ola
      std::cout  << "test";
   }
    void operator()(int x , int y){   //     function overloadi
      std::cout  << "test2";
   }
}

int main(){
   my_class m;
```

```
    m();     // gördüğün gibi görüntüsü itibariyle => function ol
    m(2,3);
}
```

MANTIK ÇOK İYİ: düşün C de bir fonksiyon yaptın ve içine statik veri yazdın. O statik ver o fonksiyona özel ama halbu ki veri global de. Başka fonksiyonlarda ulaşsın istedin. O zaman function object kullanmak çok mantıklı oldu. GENERİC programlamada çok iş yapıyor. Yani yarattığın static objeyi bağzı seçtiğin fonksiyonlarla paylaştın, private/public başka bişey.

```
class random{
   random(int low , int high):mlow(low),mhigh(high){}
   int  operator()()
   {
      return std::Rand()%(mhigh - mlow + 1) + mlow
   }
   int mlow , mhigh
}

int main(){
   random m1{5 , 9};
   random m2{50 , 89};

   m1();
   m2();
}
```

TÜR DÖNÜŞTÜRME OPERATORLERİ: member functions olmalı

kullanırken dikkat et niyetin dışında dönüşüm olabilir, explicit yap ki niyetin dışı dönüşmesin

```
class my_class{
   operator int()const;   // geri dönüş türü yazılmaz
   //explicit operator double()const; // kendisinin örtülü dönüş
```

```
}

int main(){
   my_class m;
   int x = 5;
   double y;

    x = m; // x = m.operator int();
    y = m; // çalışır double dönüşür. kullanırken dikkat et niye
}
/*****************************************************/
class test_class{
}

class my_class{
   operator test_class()const;    // tür dilediğin gibi olabilir
}

int main(){
   my_class m;
   test_class x;

    x = m; // x = m.operator test_class();
}
```

MANTIKLI ÖRNEK KULLANIM:

```
class my_string{
   my_string(const char*)
   operator const char*()const;
}

int main(){
   my_string x{"hello"}
```

```
   const char * t = x;   // logic-> x.operator const char*();
}
```

bool biraz daha özel → örnek c de bu ne: if (ival) çok doğal bir koşul. objeyi böyle yaptığını düşün !!!!!!!!!! EXPLİCİT önemli, yoksa istenmeyen değerlere dönüşür

```
class nint{
   // bizim nint fonksiyonları burda olduğunu düşün
   explicit operator bool()const
   {
     return mx != 0;
   }
}

int main(){
   nint x{568}
   if(x){   // logic    x.operator bool();

   }
   int r = x;   // explicit yaptığımız için HATA
   bool b(x);   // çalışır
}
/*std örnek*/
int x;
while(std::cin >> x)  // burdada operator bool çağrılıor
   std::cout << x;
```

```
class nint{
   // bizim nint fonksiyonları burda olduğunu düşün
     operator int() const {
        return mx;
    }
}
```

```
int main(){
   nint x{568}
   int r = x;   // artık amaç bu
}
```

Note aynı mantık: int *ptr = nullptr; if(ptr) → aynı mantık bool tür dönüştürme mantıklı !!