

EMS30 Week 5 Les 2: UDT in C++

Leerdoelen week 5 les 2. Je leert deze les hoe je:

- een UDT genaamd Breuk kunt definiëren in de vorm van een **class**;
- de **implementatie** van de class Breuk kunt afschermen van de gebruiker door **private** datavelden en **private** memberfuncties te definiëren;
- de **interface** van de class Breuk beschikbaar kunt maken voor de gebruiker door **public** memberfuncties te definiëren;
- een object van de class Breuk kunt **initialiseren** (door middel van **constructors**);
- memberfuncties kunt definiëren die ook voor **read-only** objecten van de class Breuk gebruikt kunnen worden;
- er met behulp van **operator overloading** voor kunt zorgen dat een Breuk op dezelfde wijze gebruikt kan worden als een ingebouwd datatype (b.v. **int**);

Herbruikbare component: Breuk

- Waarom wil je programma's maken die rekenen met **breuken** in plaats van met floating point getallen (**double**)?
- Waarom wil je een **component (UDT)** Breuk maken?
- Hoe doe je dat in **C**?
- Wat zijn de **nadelen** van de C oplossing?
- Hoe kan het beter in **C++**?
- Wat zijn de **voordelen** van de C++ oplossing?
- Kan het nog **mooier**?

UDT Breuk in C

- Gebruik **struct** voor dataopslag.
- Gebruik **functies** voor bewerkingen

Struct type declaratie

```
typedef struct { /* een breuk bestaat uit: */
    int boven; /* een teller en */
    int onder; /* een noemer */
} Breuk;
```

Prototypes of
Functie declaraties

```
Breuk normaliseer(Breuk b);
Breuk som(Breuk b1, Breuk b2);
```

Functie definitie

```
Breuk som(Breuk b1, Breuk b2) {
    Breuk s;
    s.boven = b1.boven * b2.onder + b1.onder * b2.boven;
    s.onder = b1.onder * b2.onder;
    return normaliseer(s);
}
```

Gebruik UDT Breuk in C

```
Breuk b1, b2, b3;  
b1.boven = 5; b1.onder = 12;  
b2.boven = 4; b2.onder = 9;  
b3 = som(b1, b2);
```

Kan overal in het programma staan!

Nadelen Breuk in C

- `b1.onder = 0;` → een “ramp die wacht om te gebeuren”.
- Programmeur die het beter denkt te weten kan zelf breuken gaan optellen:

```
b3.boven = b1.boven + b2.boven;  
b3.onder = b1.onder + b2.onder;
```

OOPS!
- Verschillende programmeurs kunnen in verschillende delen van het programma de component Breuk uitbreiden: B.v. functies: `product`, `maal`, `times`, en `multiply`.

Eigenschappen van UDT Breuk in C

Onderhoudbaarheid: **Slecht!**

- Fout in UDT is **niet** gemakkelijk te vinden.
- Iedereen kan data van UDT “verzielen”.
- Iedereen kan bewerking op UDT implementeren zonder de bestaande implementatie van de bewerking te gebruiken.
- Als er ‘iets’ niet goed gaat met variabele van UDT (b.v. vermenigvuldigen van breuken) moeten we het hele programma doorzoeken.

Aanpasbaarheid en uitbreidbaarheid: **Te Goed!**

- Iedereen kan UDT aanpassen en uitbreiden.

Herbruikbaarheid: **Slecht!**

- **Onduidelijk** welke functies bij UDT horen en welke functies bij deze applicatie horen (en toevallig dit UDT gebruiken).

UDT Breuk in C++

Gebruik **class** voor dataopslag en bewerkingen.

```
class Breuk {  
public:  
    void leesin();  
    void drukaf() const;  
    void plus(Breuk b);  
private:  
    int boven;  
    int onder;  
    void normaliseer();  
};  
  
void Breuk::plus(Breuk b) {  
    boven = boven * b.onder + onder * b.boven;  
    onder *= b.onder;  
    normaliseer();  
}
```

Class declaratie

Public memberfuncties
= interface

Private data members

Private memberfunctie

Memberfunctie definitie

```
Breuk a, b;  
a.leesin();  
b.leesin();  
a.plus(b);  
a.drukaf();
```

Voordelen Breuk in C++

- `b1.onder = 0;` → geeft **compilerfout**.
- Programmeur die het beter denkt te weten kan zelf **geen** breuken gaan optellen (zonder `Breuk::plus` te wijzigen).
- UDT Breuk kan slechts op **1** plaats in het programma uitgebreid worden.

Eigenschappen van UDT Breuk in C++

Onderhoudbaarheid: **Goed!**

- Fout in UDT is gemakkelijk te vinden.
- Als er 'iets' niet goed gaat met UDT hoef je **alleen** de implementatie van de UDT te doorzoeken. Fout **moet** in memberfuncties van de UDT zitten.
- Je kunt **niet** (eenvoudig) om de interface van de class heenwerken. (Je kan zelf geen `p1us` maken als je niet bij boven en onder kunt komen.)

Aanpasbaarheid en uitbreidbaarheid: **Goed!**

- UDT kan maar op **1** plaats uitgebreid worden.
- Private delen kunnen aangepast worden zonder dat de interface verandert. **Dus zonder dat de code die de UDT gebruikt dit merkt!** Zie practicum opgave 5.2.3.

Herbruikbaarheid: **Redelijk.**

- **Duidelijk** welke functies bij UDT horen.

Leerdoelen week 5 les 2. Je leert deze les hoe je:

- een UDT genaamd Breuk kunt definiëren in de vorm van een `class`;
- de implementatie van de `class Breuk` kunt afschermen van de gebruiker door `private` datavelden en `private` memberfuncties te definiëren;
- de interface van de `class Breuk` beschikbaar kunt maken voor de gebruiker door `public` memberfuncties te definiëren;
- een object van de `class Breuk` kunt **initialiseren** (door middel van **constructors**);
- memberfuncties kunt definiëren die ook voor **read-only** objecten van de `class Breuk` gebruikt kunnen worden;
- er met behulp van **operator overloading** voor kunt zorgen dat een `Breuk` op 10 dezelfde wijze gebruikt kan worden als een ingebouwd datatype (b.v. **int**);

User-defined Data Type in C++

EMBEDDED SYSTEMS

Constructor en const memberfuncties **zelf bestuderen!**

Dictaat H2.3 t/m 2.10

Leerdoelen week 5 les 2. Je leert deze les hoe je:

- een UDT genaamd Breuk kunt definiëren in de vorm van een `class`;
- de implementatie van de `class Breuk` kunt afschermen van de gebruiker door `private` datavelden en `private` memberfuncties te definiëren;
- de interface van de `class Breuk` beschikbaar kunt maken voor de gebruiker door `public` memberfuncties te definiëren;
- een object van de `class Breuk` kunt initialiseren (door middel van constructors);
- memberfuncties kunt definiëren die ook voor read-only objecten van de `class Breuk` gebruikt kunnen worden;
- er met behulp van **operator overloading** voor kunt zorgen dat een `Breuk` op 12 dezelfde wijze gebruikt kan worden als een ingebouwd datatype (b.v. `int`);

Gebruik UDT Breuk in C++

```
Breuk a, b;  
a.leesin();  
b.leesin();  
a.plus(b);  
a.drukaf();
```

Kan het **beter**?

Stel: **hergebruik** UDT Breuk is **succes**!

- Helpfiles
- FAQ

Gebruik Breuk is
vergelijkbaar met **int**!

```
Breuk a, b;  
cin >> a >> b;  
a += b;  
cout << a;
```

Veel werk!
Moeite waard?




Bron: <https://www.leshulp.nl/breuken-oefenen/>

Operator overloading



`a.operator+=(b);`



`a += b;`

De C++ compiler
denkt met je mee!

UDT Breuk in C++

```
class Breuk {  
public:  
    void leesin();  
    void drukaf() const;  
    void operator+=(Breuk b);  
private:  
    int boven;  
    int onder;  
    void normaliseer();  
};
```



Operator overloading is
simpel!



```
void Breuk::operator+=(Breuk b) {  
    boven = boven * b.onder + onder * b.boven;  
    onder *= b.onder;  
    normaliseer();  
}
```



UDT Breuk is een succes!

Help! Probleem met gebruik van UDT Breuk.



Aan: Broeders, J.Z.M. (Harry)

Beste meneer Broeders,

Met veel plezier maak ik gebruik van de herbruikbare C++ class Breuk van uw website <https://harrybroeders.bitbucket.io/>. Helaas ben ik nu een probleem tegengekomen. Bij het vertalen van de programmacode:

```
Breuk a, b, c;  
a += b += c;
```

Geeft de gcc C++ compiler de volgende foutmelding:

```
no match for 'operator+=' (operand types are 'Breuk' and 'void')
```

Ik heb geen idee wat dit betekent. Kunt u mij helpen?

Met vriendelijke groet,

UDT Breuk is een succes!

Mag

```
a += b += c;
```

met **int**?

- Wat betekent het dan?

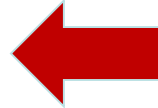
Oplossing?

- ```
b += c;
```

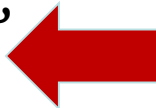
```
a += b;
```
- Kan dat niet beter?

# Fix Breuk : : operator+=

```
class Breuk {
public:
 void leesin();
 void drukaf() const;
 Breuk operator+=(Breuk b);
private:
 int boven;
 int onder;
 void normaliseer();
};
```



```
Breuk Breuk::operator+=(Breuk b) {
 boven = boven * b.onder + onder * b.boven;
 onder *= b.onder;
 normaliseer();
 return ???;
}
```



**Wat** moeten we teruggeven?

# Pointer this

```
class Breuk {
public:
 void leesin();
 void drukaf() const;
 Breuk operator+=(Breuk b);
private:
 int boven;
 int onder;
 void normaliseer();
};
```

```
Breuk Breuk::operator+=(Breuk b) {
 boven = boven * b.onder + onder * b.boven;
 onder *= b.onder;
 normaliseer();
 return *this; ← this is een pointer naar de receiver
}
```



# Is UDT Breuk een succes?

Re: Help! Probleem met gebruik van UDT Breuk.



Aan: Broeders, J.Z.M. (Harry)

Beste meneer Broeders,

De laatste twee dagen heb ik besteed aan het opsporen van een BUG in mijn programma die veroorzaakt bleek te worden door UW UDT Breuk. U begrijpt dat ik daar NIET blij mee ben!

De code:

```
Breuk a, b, c;
(a += b) += c;
```

Werkt NIET CORRECT!!!

Met (niet zo) vriendelijke groet,

Probleem?

- c wordt **niet** opgeteld bij a

Oplossing?

- reference



In C++ zijn er **verschillende** 'soorten' variabelen:

- 'gewone' variabelen;
- pointers;
- references.





Een reference is een **andere naam** voor een variabele die al bestaat.

```
int i;
int& j {i}; // initialisatie is verplicht!
```

```
i = 3;
cout << j << '\n';
// een reference is een "pseudoniem"
```

Is dit goed voor de  
onderhoudbaarheid?

Je kunt een reference gebruiken als:

- Globale variabele. 
- Lokale variabele. 
- Parameter. 
- Return type. 

# Call by value (remember EMS10)

```
void swap(int p, int q) {
 int t {p};
 p = q;
 q = t;
}

// ...
int i {3};
int j {4};
swap(i, j);
// ...
```

Deze code werkt  
**niet goed!**

Weet je nog  
waarom?

Oplossing?

# Call by reference (de C-manier)

```
void swap(int* p, int* q) {
 int t {*p};
 *p = *q;
 *q = t;
}
```

```
// ...
int i {3};
int j {4};
swap(&i, &j);
// ...
```

p wijst naar i  
q wijst naar j



# Call by reference in C++

```
void swap(int& p, int& q) {
 int t {p};
 p = q;
 q = t;
}
```

```
// ...
int i {3};
int j {4};
swap(i, j);
// ...
```

p is een andere naam voor i  
q is een andere naam voor j



Bron: <https://www.flickr.com/photos/rossendalewadey/4127442871>

Onder de 'motorkap' wordt een reference geïmplementeerd met een pointer.

# Reference return

Je kunt een reference ook teruggeven vanuit een functie.

```
int& max(int& a, int& b) {
 if (a > b) return a;
 else return b;
}
```

```
int main() {
 int x {2}, y {7}, z;
 max(x, y) = 0;
 z = max(x, y);
 // ...
}
```



Een functie die een reference teruggeeft kan ook **links** van een = teken gebruikt worden.

# Fix Breuk : : operator+=

```
class Breuk {
public:
 void leesin();
 void drukaf() const;
 Breuk& operator+=(Breuk b);
private:
 int boven;
 int onder;
 void normaliseer();
};
```

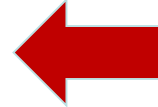
```
Breuk& Breuk::operator+=(Breuk b) {
 boven = boven * b.onder + onder * b.boven;
 onder *= b.onder;
 normaliseer();
 return *this;
}
```

Met behulp van een reference kunnen we ook **onnodige kopietjes** voorkomen.

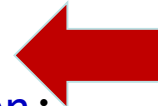
# Fix Breuk : :operator+=

```
class Breuk {
public:
 void leesin();
 void drukaf() const;
 Breuk& operator+=(const Breuk& b);
private:
 int boven;
 int onder;
 void normaliseer();
};
```

Waarom **const**?



```
Breuk& Breuk::operator+=(const Breuk& b) {
 boven = boven * b.onder + onder * b.boven;
 onder *= b.onder;
 normaliseer();
 return *this;
}
```



Als je het **leuk** vind kun je paragraaf 16.1 t/m 16.8 uit het dictaat lezen.

- Daar maak je dan kennis met zogenoemde **friend** functies.
- Vriendschap in C++ gaat wel erg ver ...

**A friend is someone who may touch your private parts.**

# Volgende les...

Generiek programmeren in C++ d.m.v. een **template**.

We maken ook een begin met **polymorfisme** en **overerving**.



# Aan de slag!

Aan de slag met [Opdrachten\\_Week\\_5\\_Les\\_2.pdf](#)

