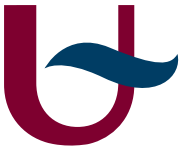
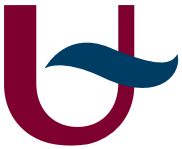


Gevorderd programmeren

Tobia De Koninck, Thomas Avé, Thomas Van Bogaert, Niels Aerens, Robin Jadoul

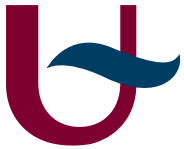
Universiteit Antwerpen



De opdracht

- Operator overloading
- Overloading van de call operator
- Friend functions
- Link met lambdas

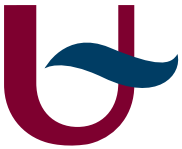
Belangrijk: waar kan het mis gaan?



Operator overloading

- Nuttige operaties op User Defined Types, laat gewoonlijke notatie toe
- Abstract denken over de types en operaties.
- vb. Persoon1 == Persoon2
- Als Member (Toegang private members)
- Als niet-member (Geen toegang private members, of moet friend zijn)

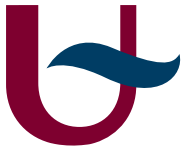
De referentie:
`https://en.wikipedia.org/wiki/Operators_in_C_and_C++`



Operator Overloading: member

Voordeel: Toegang private members.

```
1 class Integer {
2 public:
3     Integer operator+(const Integer& other) {
4         return Integer(value + other.value);
5     }
6 private:
7   int value = 0;
8 }
```

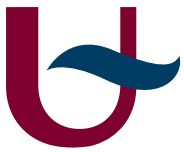


Operator Overloading: non-member

Moet via friend functions, of getters.

```
1  // Via Friends
2  class Integer {
3  ...
4  friend Integer operator+(const Integer& a, const Integer& b);
5
6  }
7
8  Integer operator+(const Integer& a, const Integer& b) {
9    return Integer(a.value + b.value);
10 }
11
12 // Of via getters
13 Integer operator+(const Integer& a, const Integer& b) {
14   return Integer(a.getValue() + b.getValue());
15 }
```

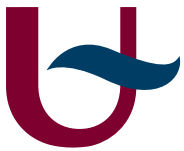
- Sommige operators kunnen niet als non-meber. (e.g. function call, static_cast, dereference operator)



Operator Overloading: non-member

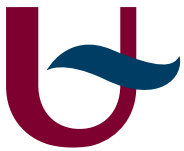
Bij non-members kunnen we het linkertype specifieren.

```
1 Integer operator +(double a, Integer b);
2 Integer operator +(Integer a, double b);
```



Operator Overloading: soorten operators I

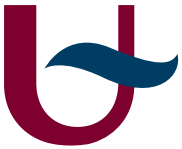
```
1  // @type
2  Color Color::operator!() {
3      return Color(255-r, 255-g, 255-b);
4  }
5
6  // type1 @ type2
7  bool Human::operator<(const Human& other){
8      return height < other.height;
9  }
10
11 // type1 = type2 assignment operator
12 // type1[type2]
13 // type1->
14 // type@ vb. type++
15 // type1 (type2) Call operator
16
17 // Speciale: Literal
18 class Integer {
19 private:
20 int m_val = 0;
21
22 public:
23 Integer(int i) : m_val(i){};
24
25 };
26
27 Integer operator""_I(unsigned long long int i) {
28 return Integer(i);
29 }
```



Operator Overloading: soorten operators II

```
30
31 int main() {
32 Integer i = 3_I;
33 }
34 ~
```

- Maar dit kan niet op alle operators:
- :: scope qualifier
- . member access
- .* member access via pointer

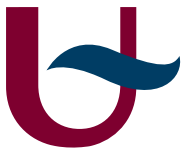


Operator Overloading: Conversion Operators

Hebben verschillend gedrag/volgorde bij bepaalde casts.
De voordelen tegenover constructor met enkele paramater:

- Constructor kan niet UDT naar ingebouwd type.
- Omzetten van nieuwe klasse naar oude klasse zonder de oude aan te passen.

```
1  class Integer {
2    private:
3      int      m_val = 0;
4
5    public:
6      operator int() const { return m_val;}
7  };
8
9  // Gebruikt als:
10   int main(){
11   Integer anInteger;
12   int i = anInteger;
13   int b(anInteger);
14   int c{anInteger};
15 }
```

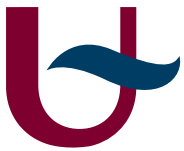


Operator Overloading: Conversion Operators

De nadelen van conversion operators:

- Is impliciet dus vaak ambigu, moeilijk op te lossen.
- Beter om expliciete method call make-int() wanneer weinig gebruikt.

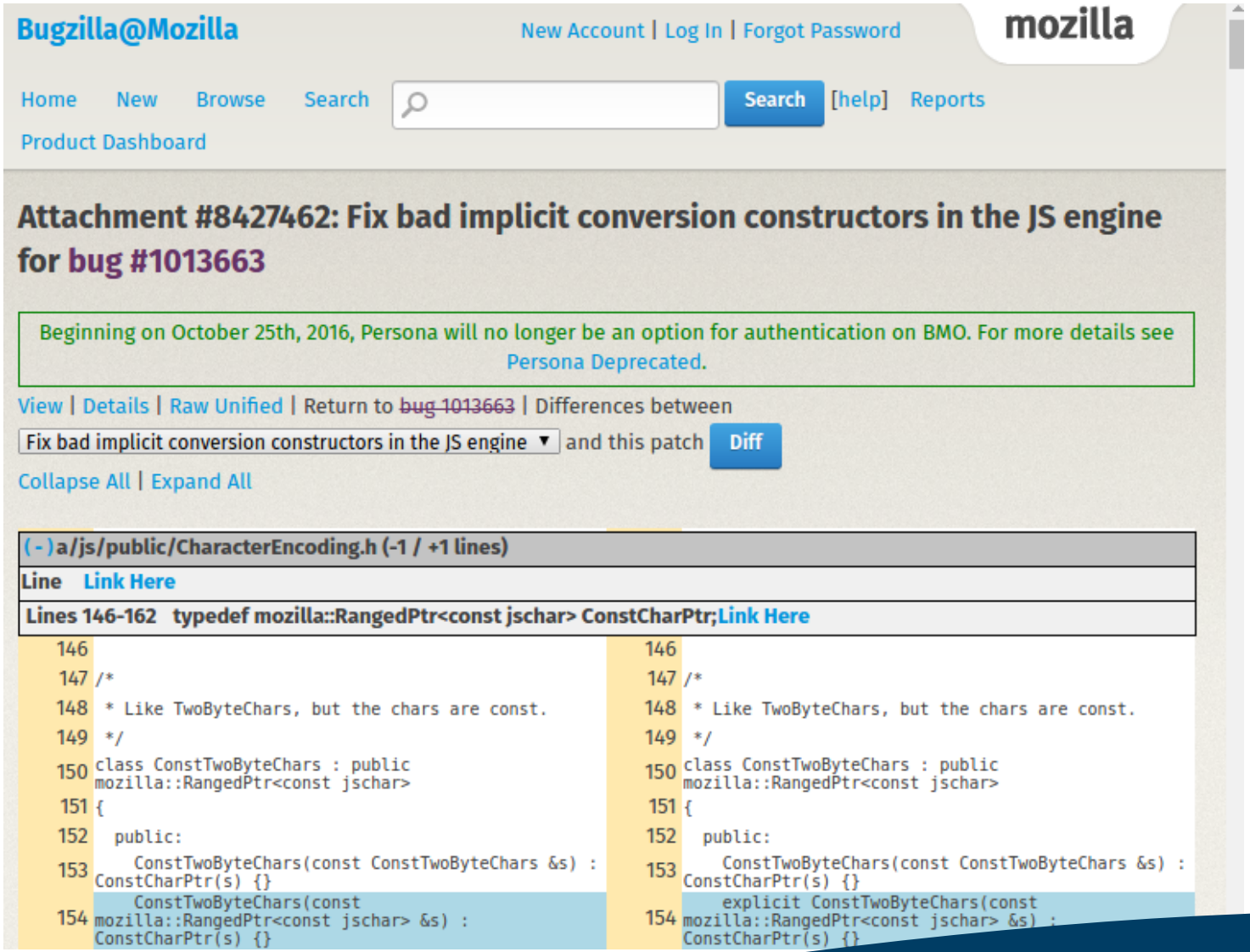
```
1 int operator+(Integer a, Integer b) {
2   return int(a) + int(b);
3 }
4
5 Integer anInt;
6 int basicInt;
7 anInt + basicInt; // Ambigue
8 //int(anInt) + basicInt
9 //anInt + Integer(basicInt);
```



Operator Overloading: Explicit

- Onderdrukt impliciete conversies.
- Telt voor zowel constructor als operators.
- Enkel als de gegeven parameters exact het juiste type zijn, wordt deze gekozen.
- Bemiddelt het nadeel hiervoor genoemd.

Operator Overloading: Explicit



Bugzilla@Mozilla

New Account | Log In | Forgot Password

mozilla

Home New Browse Search Search [help] Reports

Product Dashboard

Attachment #8427462: Fix bad implicit conversion constructors in the JS engine for bug #1013663

Beginning on October 25th, 2016, Persona will no longer be an option for authentication on BMO. For more details see Persona Deprecated.

View | Details | Raw Unified | Return to ~~bug 1013663~~ | Differences between

Fix bad implicit conversion constructors in the JS engine ▾ and this patch Diff

Collapse All | Expand All

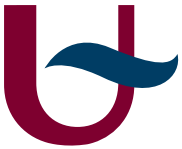
(-) a/js/public/CharacterEncoding.h (-1 / +1 lines)

Line Link Here

Lines 146-162 typedef mozilla::RangedPtr<const jschar> ConstCharPtr;Link Here

```
146
147 /*
148  * Like TwoByteChars, but the chars are const.
149  */
150 class ConstTwoByteChars : public
    mozilla::RangedPtr<const jschar>
151 {
152   public:
153     ConstTwoByteChars(const ConstTwoByteChars &s) :
    ConstCharPtr(s) {}
154     ConstTwoByteChars(const
    mozilla::RangedPtr<const jschar> &s) :
    ConstCharPtr(s) {}
```

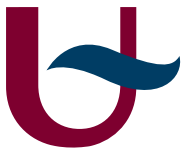
```
146
147 /*
148  * Like TwoByteChars, but the chars are const.
149  */
150 class ConstTwoByteChars : public
    mozilla::RangedPtr<const jschar>
151 {
152   public:
153     ConstTwoByteChars(const ConstTwoByteChars &s) :
    ConstCharPtr(s) {}
154     explicit ConstTwoByteChars(const
    mozilla::RangedPtr<const jschar> &s) :
    ConstCharPtr(s) {}
```



Overloading van de call operator

```
1  class Plotter {
2  public:
3    Plotter();
4    virtual ~Plotter();
5      // Old way
6    std::string plot(std::function<double (double)>,
7                     double from = 0.0, double to = 1.0) const;
8    // New way
9    std::string operator()(std::function<double (double)>,
10                          double from = 0.0, double to = 1.0) const;
11
12 private:
13   double granularity = 0.01;
14 };
```

```
1  int main() {
2      std::vector<double> input {0,0,-0.6,-3,1};
3      Polynomial poly(input);
4
5      Plotter plot;
6      std::cout << plot.plot(poly, -0.8, 0.8) << std::endl;
7      std::cout << plot(poly, -5.9, 6.4) << std::endl;
8      return 0;
9  }
```

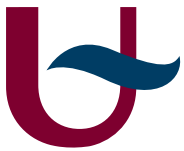


Toepassing: Functor

```
1 class Distance {
2 public:
3     Distance(int x, int y) : fromX(x), fromY(y) {}
4     int operator()(int x, int y) const;
5 private:
6     int fromX, fromY;
7 };
```

```
1 int main() {
2     Distance d(11, 10);
3     std::cout << d(1,5) << std::endl;
4     return 0;
5 }
```

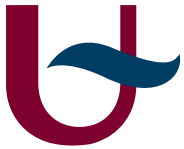
- Informatie opslaan in functor



Toepassing: Functor

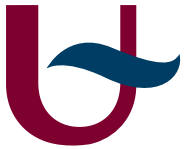
```
1  #include <iostream>
2  #include <vector>
3  #include <algorithm>
4
5  class Sum{
6  public:
7      Sum() : m_total(0) {}
8      void operator()(int value){
9          m_total += value;
10     }
11     friend std::ostream& operator<<(std::ostream& os, Sum& s) {
12         os << s.m_total;
13         return os;
14     }
15
16 private:
17     int m_total;
18 };
19
20 int main() {
21     std::vector<int> vect({1,2,3,4,5,6});
22     Sum sum = for_each(vect.begin(), vect.end(), Sum());
23     std::cout << sum << std::endl;
24     return 0;
25 }
```

- In de functor een staat bijhouden



Friend functions

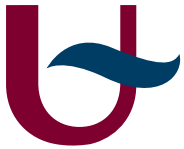
- Een functie die geen member van de class is maar die wel toegang heeft tot de private en protected members van de class



Friend functions

```
1  using std::istream;
2
3  class Integer {
4  public:
5      int getValue() const {return value;};
6
7      friend istream& operator>>(istream& is, Integer& integer);
8  private:
9      int value;
10 };
11
12 istream& operator>>(istream& is, Integer& integer) {
13     is >> integer.value;
14     return is;
15 }
```

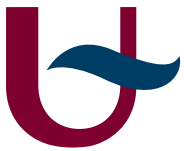
- Handig om toegang te geven tot private members aan functies zonder publieke getters en setters te definiëren



Friend functions

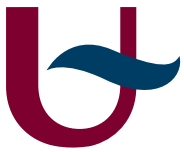
```
1  class Object;
2
3  class Factory {
4  public:
5      Object operator()();  // Maakt een object aan.
6  private:
7      std::vector<int> createdObjectIds;  // Houdt ID's van gegenereerde objecten
           bij.
8  };
9
10 class Object {
11 public:
12     friend Object Factory::operator()();
13 private:
14     Object(int id) : id{id} {};  // Private constructor.
15     const int id;
16 };
17
18 Object Factory::operator()() {
19     static int objectID = 0;
20     createdObjectIds.push_back(objectID);
21     return Object(objectID++);
22 }
```

- Laat enkel Factory toe om een Object te maken.



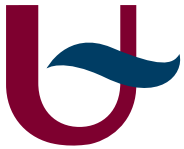
Friend functions

- $+$ Kan encapsulatie verbeteren
- $+$ Handig voor unit tests van private/protected functies



Link call operator met lambdas

- Syntactic sugar
- Lambda heeft een uniek type $\Rightarrow$ vertaling naar een unieke *functor*-klasse
- Captures worden member van de klasse
- Captures worden in de constructor toegewezen
- Function body in de call operator
- **Pitfall**: reference naar lokale variable in capture van een langer levende lambda



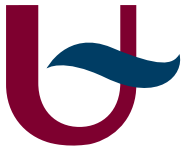
Link call operator met lambdas

- Als lambda:

```
1  std::function<int(int)> makeCountingAdder(int& counter) {
2      return [&counter](int x) -> int {
3          int val = counter + x;
4          counter++;
5          return val;
6      };
7  }
```

- Als functor:

```
1  class Lambda1 {
2      public:
3          Lambda1(int& counter) : counter(counter) {}
4          int operator()(int x) {
5              int val = counter + x;
6              counter++;
7              return val;
8          }
9
10     private:
11         int& counter;
12 };
13
14 Lambda1 makeCountingAdder(int& counter) {
15     return Lambda1(counter);
16 }
```

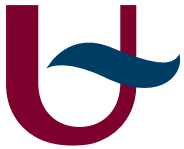


Voorbeelden van lambda's I

```
1  int main() {
2
3      std::string suffix = "\n";
4
5      // deze lambda zal een nieuwe string teruggeven die bestaat uit de meegegeven
           string plus een suffix
6      // merk op dat de suffix als value wordt doorgegeven
7      auto lambda1 = [suffix](const std::string& string) -> std::string {
8          return string + suffix;
9      };
10
11     std::string ex = "Hello World";
12
13     std::cout << lambda1(ex);
14
15     suffix = "ab";
16
17     std::cout << lambda1(ex); // suffix is the same in lambda\\
18 }
```

Output:

```
1 Hello World
2 Hello World
```

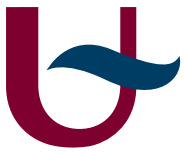


Voorbeelden van lambda's I

```cpp
int main() {

    std::string suffix = "\n";

    // deze lambda zal een nieuwe string teruggeven die bestaat uit de meegegeven
         string plus een suffix
    // merk op dat de suffix als reference wordt doorgegeven
    auto lambda1 = [&suffix](const std::string& string) -> std::string {
        return string + suffix;
    };

    std::string ex = "Hello World";

    std::cout << lambda1(ex);

    suffix = "ab\n";

    std::cout << lambda1(ex); // suffix has changed in in the lambda
}
```

Output:

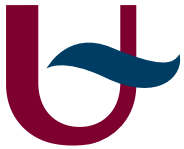
```
Hello World
Hello World
```



Toepassing van lambda's

```
1 int offset {10};
2 std::set<int*, std::function<bool(int*, int*)>> ints {[offset](int* lhs, int* rhs)
      -> bool {
3     return *lhs < *rhs + offset;
4 }};
```

- \+ Vermijd onnodige declarties van bv. structs (als alternatief voor de lambda)
- \+ Zeer handige syntax
- \+ Functie definitie direct zichtbaar
- \+ Lambda kan wel in functie/klasse gedefinieerd worden (functies niet)
- \+ Toegang tot variabel buiten de scope van de lambda (=closure) (indien gespecifieerd)

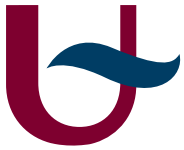


Toepassing van lambda's

- Uniek type per lambda. Onmogelijk om twee verschillende lambda's aan dezelfde variable toe te wijzen.

Pitfalls:

- auto ipv std::function (als tweede par van de template)
- Te veel dure vergelijkingen (in dit voorbeeld)



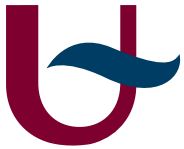
Toepassing van lambda's

Stel modulair programma, modules moeten CLI commando's registreren bij hoofdprogramma. Stel dat het bepalen van welke commando's geregistreerd moeten worden (in de modules) en het registreren van CLI commando's (in het hoofdprogamma) lange tijd in beslag nemen. Slechte aanpak:

- In hoofdprogmma een functie "`void registerCLICommand(std::string)`"
- In elk module wordt bepaald welke commando's geregistreerd worden en voor elke commando de "registerCommand" functie aangeroepen.

Waarom is dit slecht?

- Bij elke start van het programma wordt alle code nodig voor het bepalen van de commando's in elke module uitgevoerd. De `registerCommand` functie wordt voor elk commando aangeroepen.



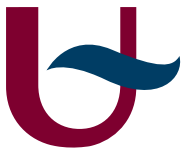
Oplossing

- In hoofdprogramma:

```
1 static std::vector<std::function<std::string>> functions; // IRL zou dit met
      een klasse en method gedaan worden
2
3 void registerCommand(std::function<std::string()>) command} {
4     commands.push_back(command);
5 }
```

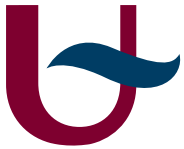
- In modules:

```
1 registerCommand([]() -> std::sting {
2     std::string result;
3
4     // calculate commando
5     // e.g. read XML files etc
6
7     return result;
8
9 });
```



Oplossing

- Principe: hoofdprogmma heeft weet dat er commando's geregistreerd zijn, maar het eigenlijke commando is weg geabstracteerd. Code pas uitvoeren wanneer nodig.
- + Bij zoeken of een bepaald commando bestaat: kleiner deel van commando's moet geregistreerd worden
- + Bij het niet nodig hebben van commando's (bv. het versie nummer van het programma tonen) worden geen enkele commando's geregistreerd.
- - mogelijk overhead als alle commando's ingeladen worden
- - developer van module moet meer nadenken



Bronnen

- http://stackoverflow.com/a/22082818
- https://en.wikipedia.org/wiki/Operators_in_C_and_C++