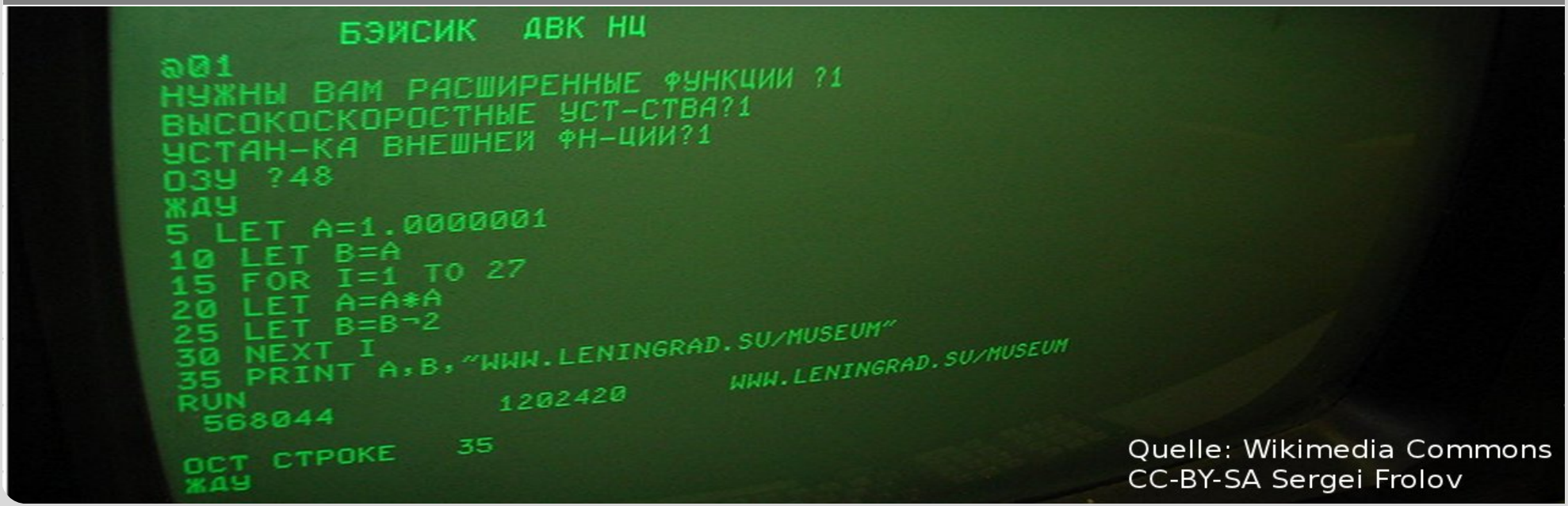
Karlsruher Institut für Technologie

C++ Workshop

6. Block, 08. Juni 2012

Markus Jung, Robert Schneider | 9. Juni 2012



Gliederung

1. forward declarations
2. friendship
   - access modifiers
   - friend
3. Operatoren
   - Arten von Operatoren
   - Aufruf von Operatoren
   - Zweck von Operator-Überladung
   - Syntax
   - member functions oder non-member functions?
   - Wichtige Hinweise
4. Praxis

1. forward declarations

Deklaration und Definition

Standard, 3.1

Eine Deklaration führt einen Namen ein (in eine translation unit).
Eine Deklaration ist zugleich eine Definition, außer:

- bei Funktionen/Klassen: fehlendem Rumpf

  (Rumpf: das zwischen den geschweiften Klammern)

- bei Variablen mit angegebenem linkage-specifier `extern` und ohne Initialisierung
- bei static member Variablen

Eine Deklaration sagt: Es gibt einen Namen
Eine Definition beschreibt, was genau mit dem Namen bezeichnet wird.

Deklaration und Definition: Beispiele

```
1  struct X {
2      int x;
3      X(): x(0) { }
4  };
5  int f(int x) { return x+a; }
6  struct S;
7  extern X anotherX;
8  struct S { int a; int b; };
9  int f(int);
10 namespace N { int d; }
11 extern int a;
12 enum { up, down };
13 extern const int c = 1;
14 X anX;
15 int a;
```

Deklaration und Definition: Beispiele

```
1  struct X {                              // defines X
2      int x;                              // defines data member x
3      X(): x(0) { }                       // defines a ctor of X
4  };
5  int f(int x) { return x+a; }            // defines f and defines x
6  struct S;                               // declares S
7  extern X anotherX;                      // declares anotherX
8  struct S { int a; int b; };             // defines S, S::a, and S::b
9  int f(int);                             // declares f
10 namespace N { int d; }                  // defines N and N::d
11 extern int a;                           // declares a
12 enum { up, down };                      // defines up and down
13 extern const int c = 1;                 // defines c
14 X anX;                                  // defines anX
15 int a;                                  // defines a
```

functions

Voraussetzungen an Funktionen beim Aufruf (Standard, 3.2:3, 3.5:2)

- Der Name muss zuvor deklariert sein.
- Es muss (irgendwo) eine entsprechende Definition geben.

Syntax

translation unit 1

```
1  void foo(int i);
2  void bar()
3  {
4      foo(42);
5  }
```

beliebige translation unit

```
1  #include <iostream>
2  void foo(int i)
3  {
4      std::cout << i;
5  }
```

Incomplete Types

Definition: incomplete type

Eine Klasse, die nur deklariert aber nicht definiert wurde, ist ein *incomplete type*.

Incomplete Types

Definition: incomplete type

Eine Klasse, die nur deklariert aber nicht definiert wurde, ist ein *incomplete type*.

- ≫Dinge≪ mit incomplete type anlegen: verboten
- Referenzen und Pointer auf incomplete types: erlaubt
  - Pointer auf einen incomplete type sind modifizierbar (Zuweisung)
  - . . . aber nicht dereferenzierbar

Standard, 3.2:4, 5.2.2:4, 5.3.1:4, 5.7:1, 8.3.5:6

Incomplete Types

Definition: incomplete type

Eine Klasse, die nur deklariert aber nicht definiert wurde, ist ein *incomplete type*.

- »Dinge« mit incomplete type anlegen: verboten
- Referenzen und Pointer auf incomplete types: erlaubt
  - Pointer auf einen incomplete type sind modifizierbar (Zuweisung)
  - ...aber nicht dereferenzierbar

Standard, 3.2:4, 5.2.2:4, 5.3.1:4, 5.7:1, 8.3.5:6

```
1  struct X;
2  void foo(X); // alt: void foo(X x);
3
4  struct X { int m; };
5  void foo(X x)
6  {
7      x.m = 42;
8  }
```

Incomplete Types: Beispiel

Syntax

```
1   class A;
2  struct B;
3   union C;
4    enum D;
```

```
1   class A { int m; };
2  struct B { void doit(); };
3   union C { int m; char c; };
4    enum D { X, Y, Z };
```

Incomplete Types: Beispiel

Syntax

```
1  class A;
2  struct B;
3  union C;
4  enum D;
```

```
1  class A { int m; };
2  struct B { void doit(); };
3  union C { int m; char c; };
4  enum D { X, Y, Z };
```

Wozu verwenden?

Incomplete Types: Beispiel

Syntax

```
1   class A;
2  struct B;
3   union C;
4    enum D;
```

```
1   class A { int m; };
2  struct B { void doit(); };
3   union C { int m; char c; };
4    enum D { X, Y, Z };
```

Wozu verwenden?

Für gegenseitige Abhängigkeiten oder auch information hiding!

```
1  class A;
2  class B;
3  class B { A* getA(); };
4  class A { void doB(B*); };
```

2. friendship
   - access modifiers
   - friend

access modifiers

Wir kennen bereits für die member von Klassen (`class`, `struct`, `union`). Wenn man sagt, eine Klasse habe Zugriff auf einen member, so bedeutet dies, dass die member dieser Klasse diesen Zugriff besitzten (z.B. member functions).

access modifiers

Wir kennen bereits für die member von Klassen (`class`, `struct`, `union`). Wenn man sagt, eine Klasse habe Zugriff auf einen member, so bedeutet dies, dass die member dieser Klasse diesen Zugriff besitzten (z.B. member functions).

Zugriffskontrolle für member

Zugriff für member in einer Klasse A für:

- `public` – jeden
- `protected` – nur die Klasse A selbst und deren Kindklassen
- `private` – nur die Klasse A selbst

encapsulation & information hiding

- Faustregel: Alles so weit wie möglich schützen!
- data member i.d.R. private
- nur sichere, in sich geschlossene Funktionen public

friendship

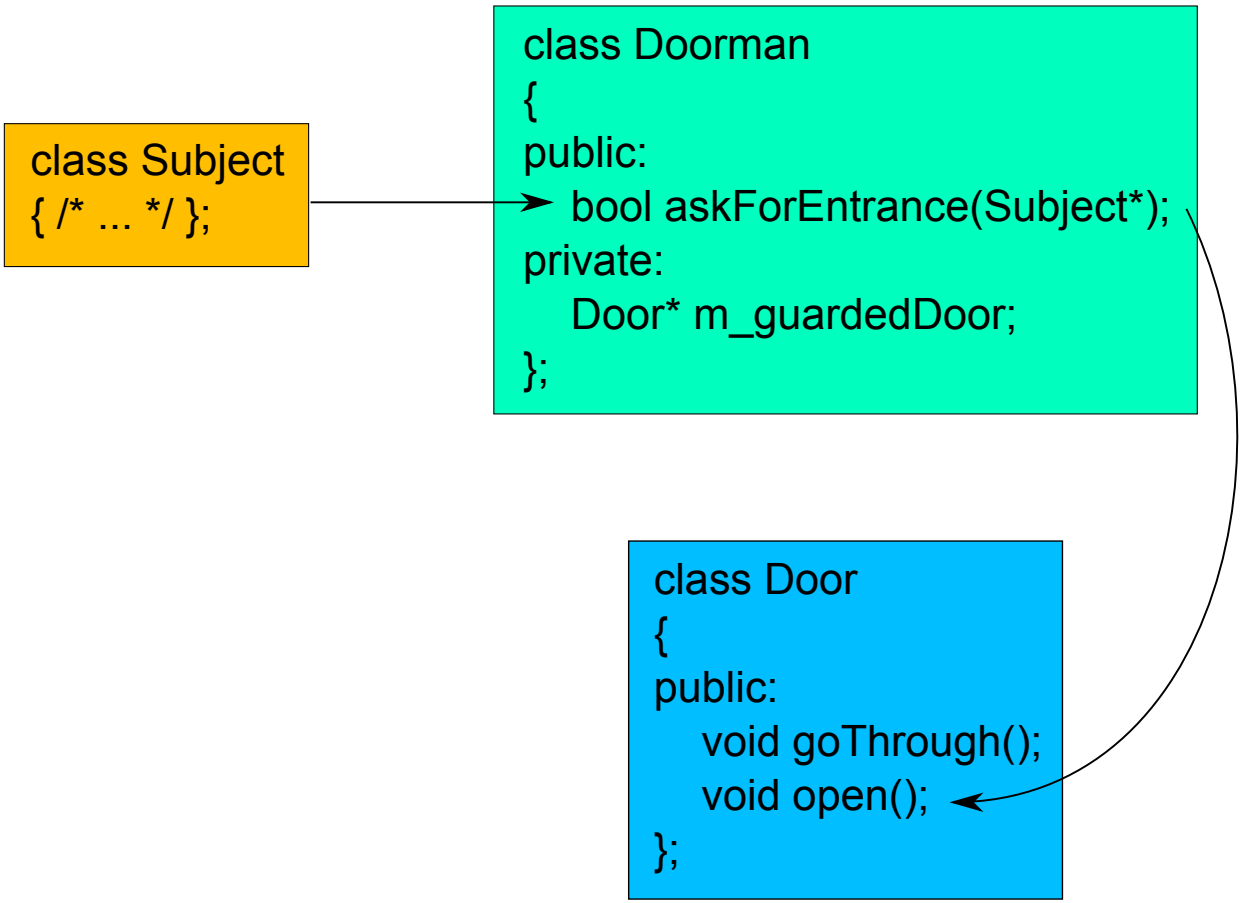
Friendship erlaubt eine feinere Kontrolle über member access:

```
class Subject
{ /* ... */ };
```

```
class Doorman
{
public:
    bool askForEntrance(Subject*);
private:
    Door* m_guardedDoor;
};
```

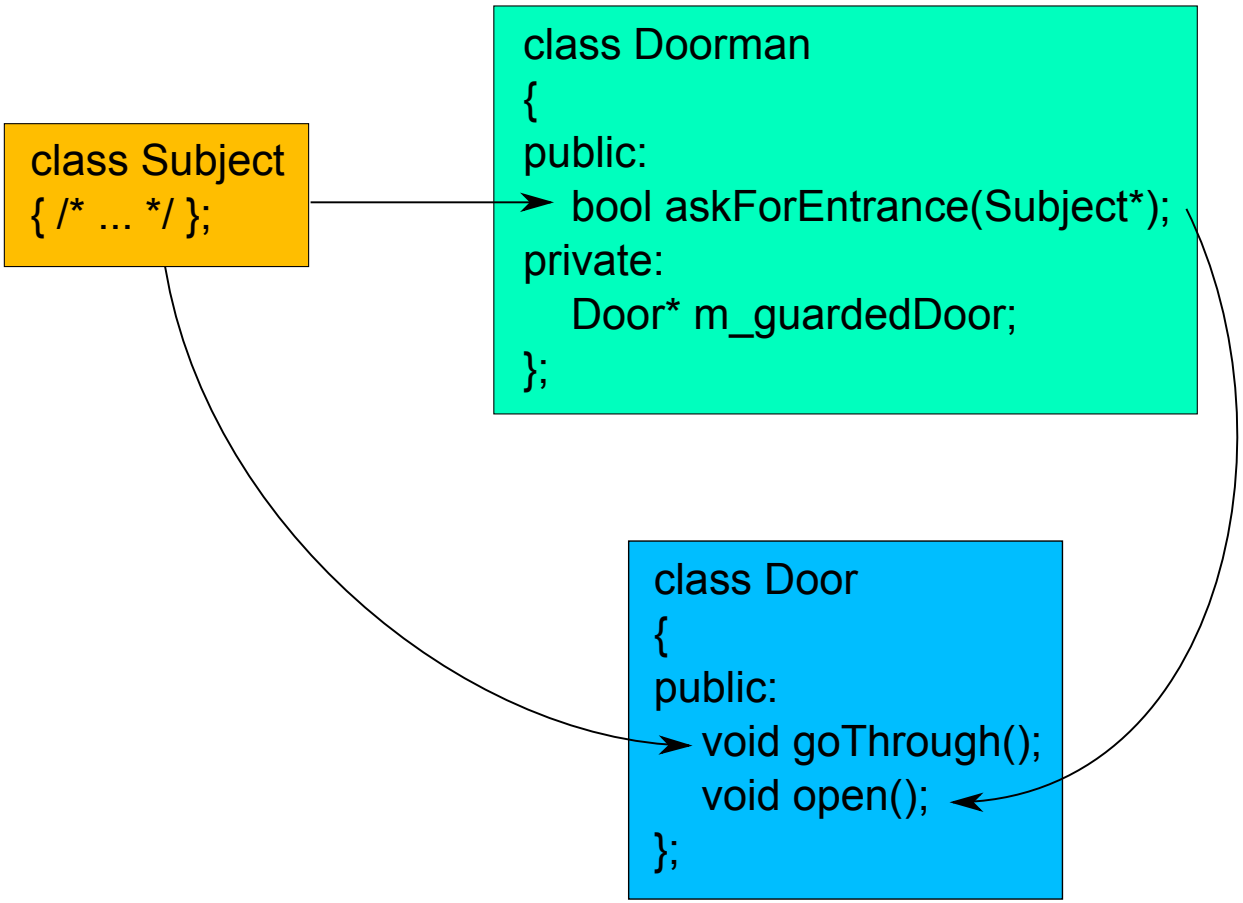
friendship

Friendship erlaubt eine feinere Kontrolle über member access:



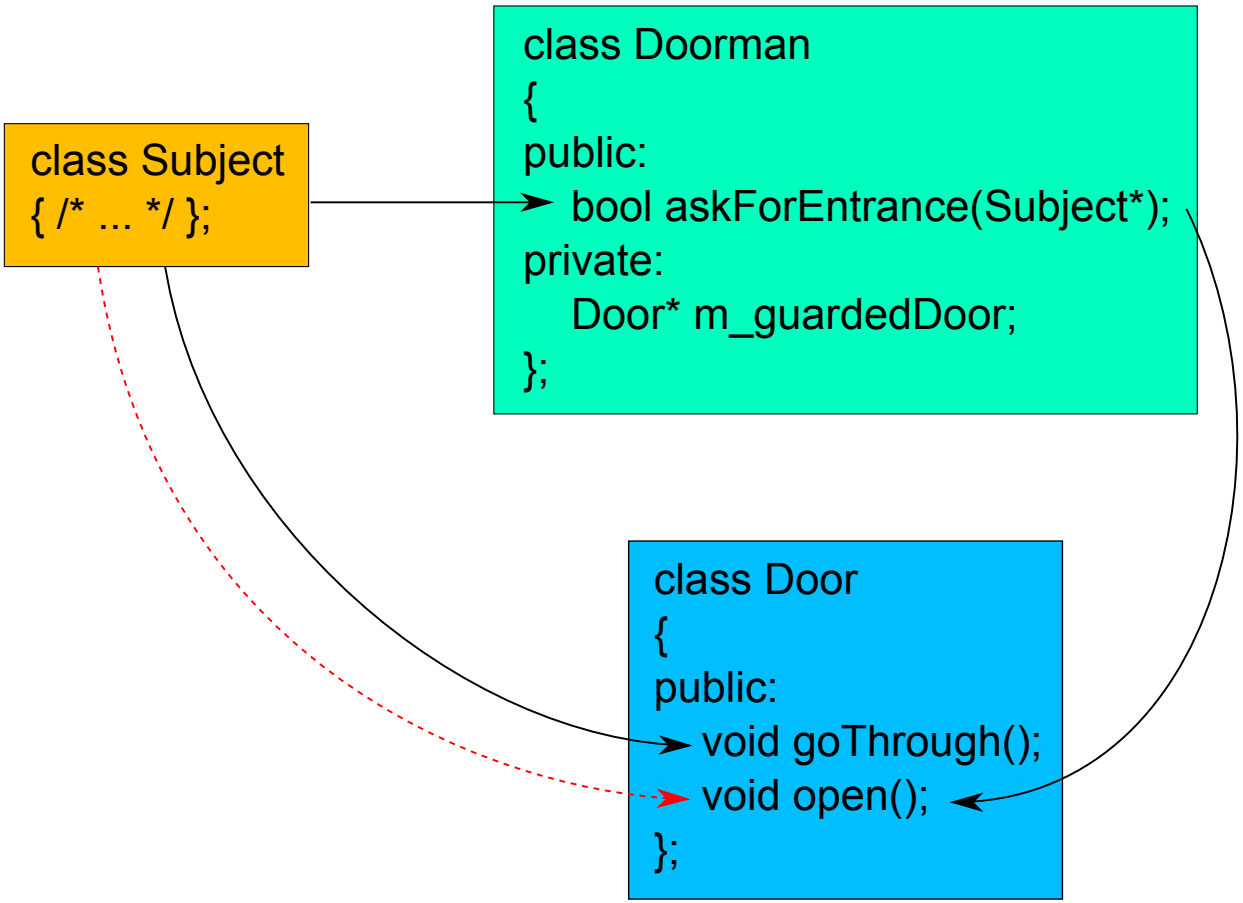
friendship

Friendship erlaubt eine feinere Kontrolle über member access:



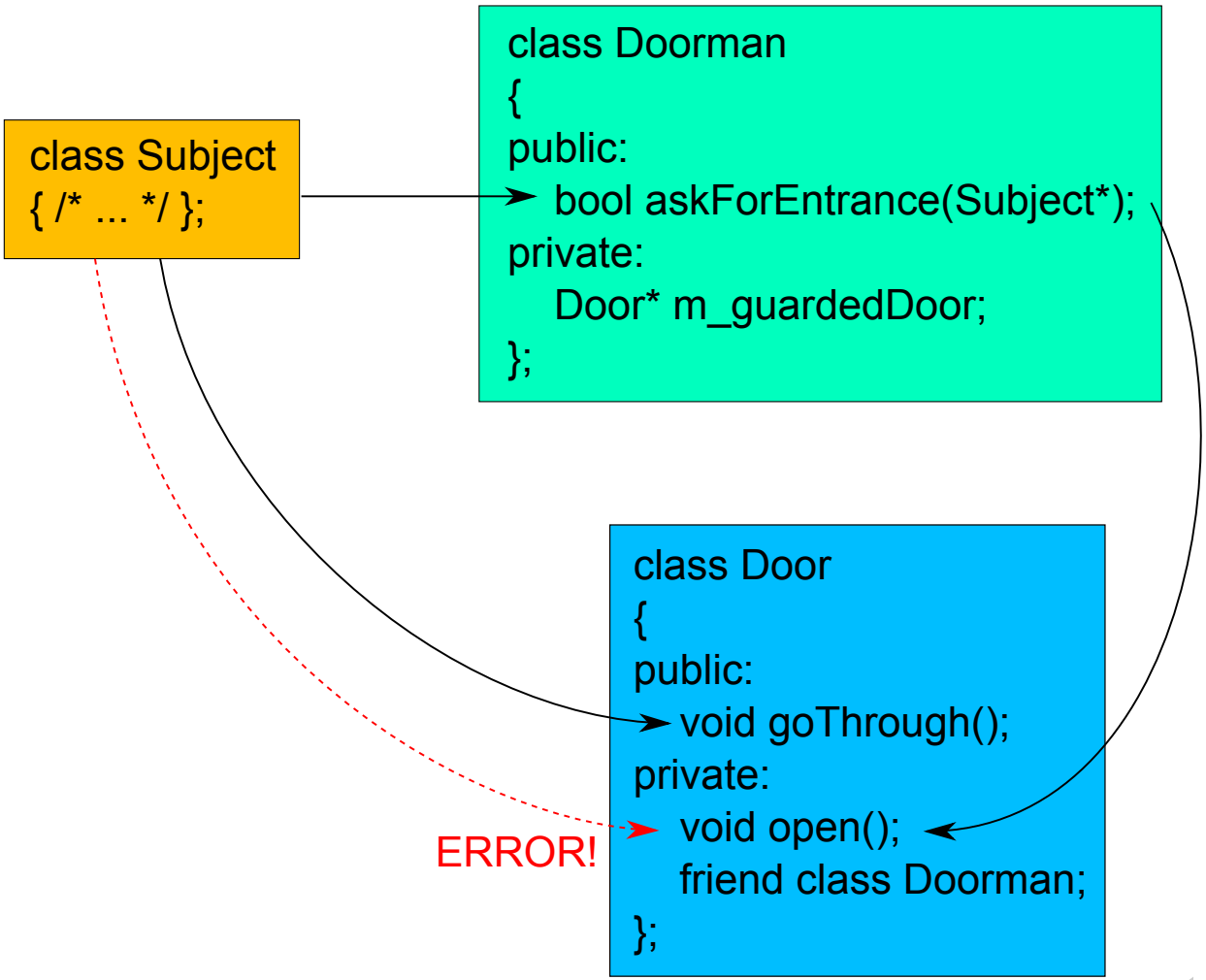
friendship

Friendship erlaubt eine feinere Kontrolle über member access:



friendship

Friendship erlaubt eine feinere Kontrolle über member access:



Syntax

```
1  struct B;
2  void globalFunc(B b);
3  struct A { void foo(B b); };
4
5  struct B {
6      // public, private, protected or nothing at all
7      friend struct A;
8      friend void globalFunc(B);
9  private:
10     int m;
11     void doInternal();
12 };
```

Syntax

```
1  struct B;
2  void globalFunc(B b);
3  struct A { void foo(B b); };
4
5  struct B {
6      // public, private, protected or nothing at all
7      friend struct A;
8      friend void globalFunc(B);
9  private:
10     int m;
11     void doInternal();
12 };
```

Syntax (Standard, 11.4)

Friendship gewährt man durch eine Deklaration in der Klassen-*member-specification* mit *friend-specifier* und für einen Typen mit *elaborated-type-specifier* (7.1.5.3).

Wirkung

Siehe Standard, 11.4. Hauptsächlich: man darf in member functions der befreundeten Klasse auf private und protected member der Freundschaft-gewährenden Klasse zugreifen.

Beispiel

```
1  struct B {
2      friend struct A;
3      friend void globalFunc(B);
4  private:
5      int m;
6      void doInternal();
7  };
```

```
1  void globalFunc(B b)
2  {
3      b.doInternal();
4  }
5  void A::foo(B b)
6  {
7      b.m = 42;
8      b.doInternal();
9  }
```

Hinweise

- Friendship wird nicht vererbt!
- Friendship ist nicht transitiv!
- Friendship ist nicht symmetrisch!

3. Operatoren
   - Arten von Operatoren
   - Aufruf von Operatoren
   - Zweck von Operator-Überladung
   - Syntax
   - member functions oder non-member functions?
   - Wichtige Hinweise

Alle Operatoren in C++

Alle Operatoren in C++

memory	`*` `&` `new` `new[]` `delete` `delete[]` `sizeof`

Alle Operatoren in C++

Alle Operatoren in C++

memory	`* & new new[] delete delete[] sizeof`
arithmetic	`+ - * / % ^ & | ~ << >>`

Alle Operatoren in C++

Alle Operatoren in C++

memory	`* & new new[] delete delete[] sizeof`
arithmetic	`+ - * / % ^ & | ~ << >>`
logic	`&& || !`

Alle Operatoren in C++

Alle Operatoren in C++

memory	`* & new new[] delete delete[] sizeof`
arithmetic	`+ - * / % ^ & | ~ << >>`
logic	`&& || !`
comparison	`< > <= >= == !=`

Alle Operatoren in C++

Alle Operatoren in C++

memory	`* & new new[] delete delete[] sizeof`
arithmetic	`+ - * / % ^ & | ~ << >>`
logic	`&& || !`
comparison	`< > <= >= == !=`
assignment	`= += -= *= /= %= ^= &= |= >>= <<= ++ --`

Alle Operatoren in C++

Alle Operatoren in C++

memory	`* & new new[] delete delete[] sizeof`
arithmetic	`+ - * / % ^ & | ~ << >>`
logic	`&& || !`
comparison	`< > <= >= == !=`
assignment	`= += -= *= /= %= ^= &= |= >>= <<= ++ --`
others	`() [] , :: ?: typeid`

Alle Operatoren in C++

Alle Operatoren in C++

memory	`* & new new[] delete delete[] sizeof`
arithmetic	`+ - * / % ^ & | ~ << >>`
logic	`&& || !`
comparison	`< > <= >= == !=`
assignment	`= += -= *= /= %= ^= &= |= >>= <<= ++ --`
others	`() [] , :: ?: typeid`
member access	`->* -> . .*`

Alle Operatoren in C++

Alle Operatoren in C++

memory	`* & new new[] delete delete[] sizeof`
arithmetic	`+ - * / % ^ & | ~ << >>`
logic	`&& || !`
comparison	`< > <= >= == !=`
assignment	`= += -= *= /= %= ^= &= |= >>= <<= ++ --`
others	`() [] , :: ?: typeid`
member access	`->* -> . .*`

42 overloadable operators + 4 unary forms

Nicht überladbar: `sizeof typeid . .* :: ?:`

Gruppiert nach Parameterzahl

Unäre Operatoren

`!` `++` `--` sowie die unären Varianten von `+` `-` `*` `&`

Gruppiert nach Parameterzahl

Unäre Operatoren

`!` `++` `--` sowie die unären Varianten von `+` `-` `*` `&`

Binäre Operatoren

(alle anderen)

Gruppiert nach Parameterzahl

Unäre Operatoren

`!` `++` `--` sowie die unären Varianten von `+` `-` `*` `&`

Binäre Operatoren

(alle anderen)

Ternärer Operator

`?:`

Unäre Operatoren

operator function invocation

For operator `@` and expression `a`:

expression	as member function	as (global) function
`@a`	`(a).operator@ ()`	`operator@ (a)`
`a@`	`(a).operator@ (0)`	`operator@ (a, 0)`
`a->`	`(a).operator->()`	n/a

```
1  MyClass a;
2  -a;      // a.operator- () OR operator- (a)
3  ++a;     // a.operator++ () OR operator++ (a)
4  a++;     // a.operator++ (0) OR operator++ (a, 0)
5  a->m;    // a.operator-> ()
6           //      and then: depends on return type!
```

Binäre Operatoren

operator function invocation

For operator `@` and expression `a`:

expression	as member function	as (global) function
`a@b`	`(a).operator@ (b)`	`operator@ (a, b)`
`a=b`	`(a).operator= (b)`	n/a
`a[b]`	`(a).operator[] (b)`	n/a
`a(b)`	`(a).operator() (b)`	n/a

```
1 MyClass a;
2 OtherClass b; OtherClass c;
3 a + b;     // a.operator+ (b) OR operator+ (a, b)
4 a[b];      // a.operator[] (b)
5 a(b, c);   // a.operator() (b, c)
```

Wozu Operator-Überladung?

- intuitive Syntax z.B. für Matrizen `A * B + C`

Wozu Operator-Überladung?

- intuitive Syntax z.B. für Matrizen `A * B + C`
- Hinteranderausführen, etwa `a + b + c` statt `add(add(a, b), c)` oder `a.add(b).add(c)`

Wozu Operator-Überladung?

- intuitive Syntax z.B. für Matrizen `A * B + C`
- Hinteranderausführen, etwa `a + b + c` statt `add(add(a, b), c)` oder `a.add(b).add(c)`
- Syntax-Kompatibilität (z.B. Pointer und Smart-Pointer)

Wozu Operator-Überladung?

- intuitive Syntax z.B. für Matrizen `A * B + C`
- Hinteranderausführen, etwa `a + b + c` statt `add(add(a, b), c)` oder `a.add(b).add(c)`
- Syntax-Kompatibilität (z.B. Pointer und Smart-Pointer)
- Spezialfälle
  - address-of `&`
  - class member access `->`
  - assigment `=`

Einfache binäre Operatoren

a.operator@ (b)

```
1  class OtherClass { /* ... */ };
2  class MyClass {
3  public:
4      RETURN_TYPE operator@ (OtherClass p) { /* ... */ };
5  };
6
7  MyClass a;
8  OtherClass b;
9  RETURN_TYPE result = a @ b;
```

Einfache binäre Operatoren

Karlsruher Institut für Technologie

Non-assigment operators und kein []

operator@ (a, b)

```
1  class OtherClass { /* ... */ };
2  class MyClass { /* ... */ };
3  RETURN_TYPE operator@ (MyClass p0, OtherClass p1) {
4      /* ... */
5  }
6
7  MyClass a;
8  OtherClass b;
9  RETURN_TYPE result = a @ b;
```

Binary assignment operators

a.operator@ (b)

```
1  class OtherClass { /* ... */ };
2  class MyClass {
3  public:
4      MyClass& operator@ (OtherClass p) {
5          /* modify this instance */
6          return *this;
7      };
8  };
9
10 MyClass a;
11 OtherClass b; OtherClass c;
12 MyClass result = a @ b @ c;
```

Binary assignment operators

Kein [] und kein ()

operator@ (a, b)

```
1  class OtherClass { /* ... */ };
2  class MyClass { /* ... */ };
3
4  MyClass& operator@ (MyClass& p0, OtherClass p1)
5  {
6      /* modify p0 */
7      return p0;
8  }
9
10 MyClass a;
11 OtherClass b; OtherClass c;
12 MyClass result = a @ b @ c;
```

Unäre Operatoren: Präfix

Karlsruher Institut für Technologie

Non-assignment operators

a.operator@ ()

```
1  class OtherClass { /* ... */ };
2  class MyClass {
3  public:
4      RETURN_TYPE operator@ () { /* ... */ }
5  };
6
7  MyClass a;
8  RETURN_TYPE result = @a;
```

Unäre Operatoren: Präfix

Non-assignment operators, kein ->

operator@ (a)

```
1  class OtherClass { /* ... */ };
2  class MyClass { /* ... */ };
3
4  RETURN_TYPE operator@ (MyClass p)
5  { /* ... */ }
6
7  MyClass a;
8  RETURN_TYPE result = @a;
```

Unäre Operatoren: Suffix

Die einzigen unären Suffix-Operatoren sind ++ und --, also assigment-ops.

a.operator@ (0)

```
1  class OtherClass { /* ... */ };
2  class MyClass {
3  public:
4      MyClass operator@ (int) {
5          MyClass ret = *this;     // copy current state
6          /* modify this instance */
7          return ret; // return state before modification
8      }
9  };
10
11 MyClass a;
12 MyClass result = a@;
```

Unäre Operatoren: Suffix

Die einzigen unären Suffix-Operatoren sind ++ und --, also assigment-ops.

operator@ (a)

```
1  class OtherClass { /* ... */ };
2  class MyClass { /* ... */ };
3
4  MyClass operator@ (MyClass& p, int)
5  {
6      MyClass ret = p;     // copy current state
7      /* modify p */
8      return ret; // return state before modification
9  }
10
11 MyClass a;
12 MyClass result = a@;
```

Fälle ohne Wahlmöglichkeit

Nur als member-function-Variante: `= [] () ->`

Fälle ohne Wahlmöglichkeit

Nur als member-function-Variante: `= [] () ->`

Bei allen anderen Operatoren hat man die Wahl.
Einschränkung: operator function invocation

```
1  #include <ostream>
2  #include <iostream>
3
4  using namespace std;
5  // definition of the cout object according to 27.3
6  extern ostream cout;
7
8  class MyClass { /* ... */ };
9  std::ostream& operator <<(ostream& o, MyClass p)
10 { /* ... */ return o;}
11
12 MyClass p;
13 cout << p; // calls: operator <<(cout, p)
```

Fälle mit Wahlmöglichkeit

Habe keine guten Entscheidungshilfen gefunden!

Bemerkung:
Zwecks Symmetrie nimmt man meist die Variante mit der non-member function:

```
1  class MyInt;
2  MyInt operator+ (MyInt, int);
3  MyInt operator+ (int, MyInt);
4
5  class MyInt {
6      friend MyInt operator+ (MyInt, int);
7      // MyInt operator +(MyInt int);
8      friend MyInt operator+ (int, MyInt);
9      // n/a
10
11 private:
12     int m;
13 };
```

Self-assignment

```
1  struct MyStruct {
2      int* m;
3
4      MyStruct() { m = new int; }
5      ~MyStruct() { delete m; }
6
7      MyStruct& operator= (MyStruct& p) {
8          this->m = p.m;
9          delete p.m;
10         p.m = 0;
11
12
13
14         return *this;
15     }
16 };
17 MyStruct s;
18 s = s;
```

Self-assignment

```
1  struct MyStruct {
2      int* m;
3
4      MyStruct() { m = new int; }
5      ~MyStruct() { delete m; }
6
7      MyStruct& operator= (MyStruct& p) {
8          if(p.m != this->m) {
9              this->m = p.m;
10             delete p.m;
11             p.m = 0;
12         }
13
14         return *this;
15     }
16 };
17 MyStruct s;
18 s = s;
```

Weitere Hinweise

`http://www.parashift.com/c++-faq-lite/operator-overloading.html#faq-13.9`

- operator overloading ist nicht dafür da, dem class designer das Leben einfacher zu machen, sondern dem Nutzer einer Klasse!

Weitere Hinweise

`http://www.parashift.com/c++-faq-lite/operator-overloading.html#faq-13.9`

- operator overloading ist nicht dafür da, dem class designer das Leben einfacher zu machen, sondern dem Nutzer einer Klasse!
- Operatoren sollten intuitive und eindeutige Bedeutung haben

Weitere Hinweise

`http://www.parashift.com/c++-faq-lite/operator-overloading.html#faq-13.9`

- operator overloading ist nicht dafür da, dem class designer das Leben einfacher zu machen, sondern dem Nutzer einer Klasse!
- Operatoren sollten intuitive und eindeutige Bedeutung haben
- Identitäten sollten erhalten bleiben, z.B. `x == x + y - y`

Weitere Hinweise

`http://www.parashift.com/c++-faq-lite/operator-overloading.html#faq-13.9`

- operator overloading ist nicht dafür da, dem class designer das Leben einfacher zu machen, sondern dem Nutzer einer Klasse!
- Operatoren sollten intuitive und eindeutige Bedeutung haben
- Identitäten sollten erhalten bleiben, z.B. `x == x + y - y`
- manche Operatoren sollten den ersten Operanden verändern, z.B. `= += *=`

Weitere Hinweise

`http://www.parashift.com/c++-faq-lite/operator-overloading.html#faq-13.9`

- operator overloading ist nicht dafür da, dem class designer das Leben einfacher zu machen, sondern dem Nutzer einer Klasse!
- Operatoren sollten intuitive und eindeutige Bedeutung haben
- Identitäten sollten erhalten bleiben, z.B. `x == x + y - y`
- manche Operatoren sollten den ersten Operanden verändern, z.B. `= += *=`
- manche aber eben nicht, z.B. `+ * /`

4 Praxis

Praxis!

- Aufgabe 1: Schachprogramm bis nächste Woche fertig machen

`https://github.com/kit-cpp-workshop/workshop-ss12-06`

Aufgabenbeschreibungen und Hinweise: Siehe `README.md`