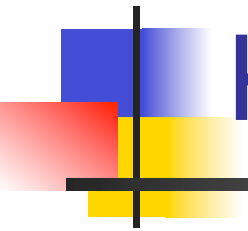


Klassen

Datenkapselung, Klassen-Typen,
Mitgliedsfunktionen,
Konstruktoren, Konversionen,
Zufallszahlen

Datenkapselung: Motivation

Erinnern wir uns:

```
struct rational {
  int n;
  int d; // INV: d != 0
};
```

neuer Typ für rationale Zahlen...

Datenkapselung: Motivation

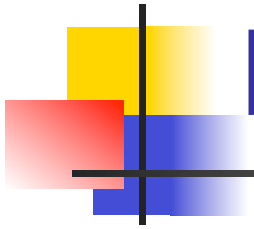
Erinnern wir uns:

neuer Typ für rationale Zahlen...

```
struct rational {
  int n;
  int d; // INV: d != 0
};
// POST: return value is the sum of a and b
rational operator+ (const rational a, const rational b)
{
  rational result;
  result.n = a.n * b.d + a.d * b.n;
  result.d = a.d * b.d;
  return result;
}
```

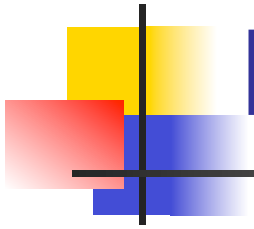
...und Funktionalität darauf

Datenkapselung: Motivation



Resultierende Bibliothek:

- **`rational.h`:**
  - Definition des Structs **`rational`**
- **`rational.cpp`:**
  - Arithmetische Operatoren (**`operator+`, `operator+=`,...**)
  - Relationale Operatoren (**`operator==`,...**)
  - Ein/Ausgabe (**`operator>>`, `operator<<`**)

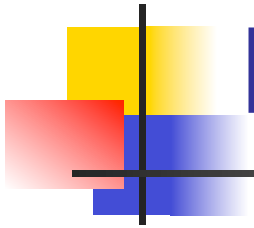


Datenkapselung: Motivation

Gedankenexperiment:

- Verkauf der Bibliothek an einen Kunden: *Rationales Denken AG* (RAT)
- RAT entwickelt Anwendung unter Benutzung der Bibliothek (wie in **`userational2.cpp`**)

Datenkapselung: Motivation Problem 1



Initialisierung ist mühsam:

- Will RAT die rationale Zahl ½ haben, müssen sie folgendes schreiben:

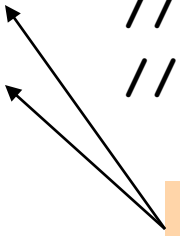
```
rational r; // default-initialization of r
r.n = 1;    // assignment to data member
r.d = 2;    // assignment to data member
```

Datenkapselung: Motivation Problem 1

Initialisierung ist mühsam:

- Will RAT die rationale Zahl ½ haben, müssen sie folgendes schreiben:

```
rational r; // default-initialization of r
r.n = 1;    // assignment to data member
r.d = 2;    // assignment to data member
```



zwei Zuweisungen für einen Wert (½)

Datenkapselung: Motivation Problem 1

Initialisierung ist mühsam:

- Will RAT die rationale Zahl ½ haben, müssen sie folgendes schreiben:

```
rational r; // default-initialization of r
r.n = 1;    // assignment to data member
r.d = 2;    // assignment to data member
```

zwei Zuweisungen für einen Wert (½)

Bei grösseren Structs (z.B. `rational_vector_3`) sind es noch viel mehr...

Datenkapselung: Motivation Problem 1

Initialisierung ist mühsam:

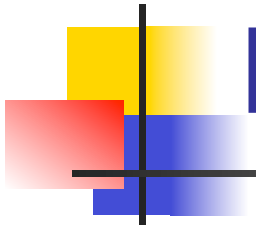
- Will RAT die rationale Zahl ½ haben, müssen sie folgendes schreiben:

```
rational r; // default-initialization of r
r.n = 1;    // assignment to data member
r.d = 2;    // assignment to data member
```

zwei Zuweisungen für einen Wert (½)

Wenn nur eine der Zuweisungen vergessen wird, ist der Wert undefiniert!

Datenkapselung: Motivation Problem 2

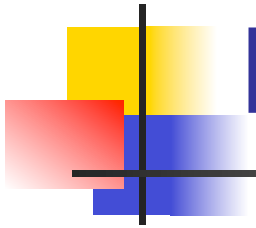


Invarianten sind nicht garantiert:

- Programmierer bei RAT könnte schreiben:

```
rational r;
r.n = 1;
r.d = 0; // Integrity of r is violated!
```

Datenkapselung: Motivation Problem 2



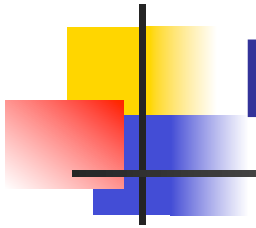
Invarianten sind nicht garantiert:

- Programmierer bei RAT könnte schreiben:

Selbst, wenn der Programmierer bei RAT nicht so "doof" ist, kann durch fehlerhafte Berechnungen irgendwo anders ein Nenner von 0 aus Versehen zustandekommen!

```
rational r;
r.n = 1;
r.d = 0; // Integrity of r is violated!
```

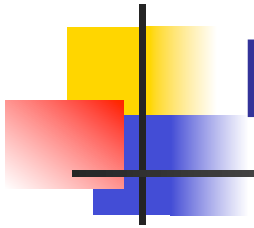
Datenkapselung: Motivation Lösungsansatz für Probleme 1,2



Erweitern der Bibliothek um eine sichere *Initialisierungsfunktion*:

```
// PRE:  d != 0
// POST: return value is n/d
rational create_rational (const int n, const int d) {
  assert (d != 0);
  rational result;
  result.n = n;
  result.d = d;
  return result;
}
```

Datenkapselung: Motivation Lösungsansatz für Probleme 1,2

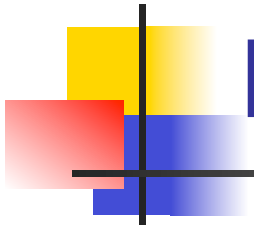


Erweitern der Bibliothek um eine sichere *Initialisierungsfunktion*:

- RAT kann nun schreiben:

```
rational r = create_rational (1, 2);
```

Datenkapselung: Motivation Lösungsansatz für Probleme 1,2



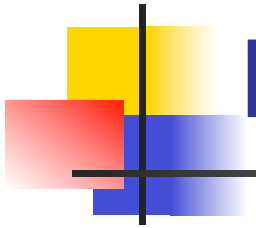
Erweitern der Bibliothek um eine sichere *Initialisierungsfunktion*:

- RAT kann nun schreiben:

```
rational r = create_rational (1, 2);
```

Aber (1): `r.d = 0;` ist weiterhin möglich!

Datenkapselung: Motivation Lösungsansatz für Probleme 1,2



Erweitern der Bibliothek um eine sichere *Initialisierungsfunktion*:

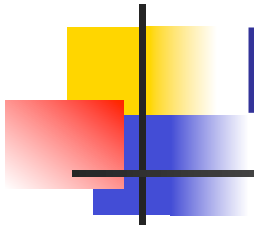
- RAT kann nun schreiben:

```
rational r = create_rational (1, 2);
```

Aber (1): `r.d = 0;` ist weiterhin möglich!

Aber (2): Initialisierung kann weiterhin vergessen werden!

Datenkapselung: Motivation Problem 3 (das eigentliche)



Interne Repräsentation nicht änderbar!

- RAT fragt an: können wir rationale Zahlen mit erweitertem Wertebereich haben?

Datenkapselung: Motivation Problem 3 (das eigentliche)

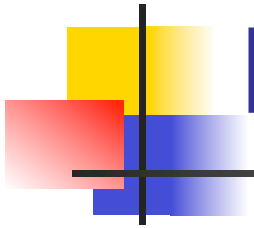
Interne Repräsentation nicht änderbar!

- RAT fragt an: können wir rationale Zahlen mit erweitertem Wertebereich haben?
- Klar, kein Problem, z.B.

```
struct rational {
  int n;
  int d; // INV: d != 0
};
```

```
struct rational {
  unsigned int n;
  unsigned int d;
  bool is_negative;
};
```

Datenkapselung: Motivation Problem 3 (das eigentliche)



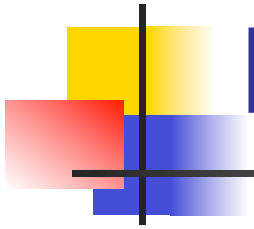
Notwendige Schritte:

- Anpassen von **`rational.h`** und **`rational.cpp`** an die neue interne Repräsentation rationaler Zahlen
- Lieferung der neuen Bibliotheksversion an RAT

Datenkapselung: Motivation Problem 3 (das eigentliche)

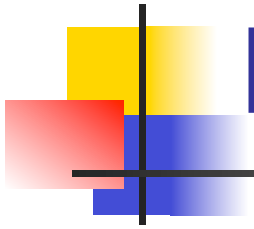
- RAT ruft an: *nichts geht mehr!*

Datenkapselung: Motivation Problem 3 (das eigentliche)



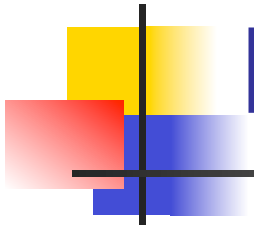
- RAT ruft an: *nichts geht mehr!*
- Problem:
  - Anwendungscode enthält jede Menge **`.n's`** und **`.d's`**

Datenkapselung: Motivation Problem 3 (das eigentliche)



- RAT ruft an: *nichts geht mehr!*
- Problem:
  - Anwendungscode enthält jede Menge **`.n's`** und **`.d's`**
  - ...aber die bedeuten jetzt *etwas anderes* (Absolutwerte von Zähler und Nenner)

Datenkapselung: Motivation Problem 3 (das eigentliche)

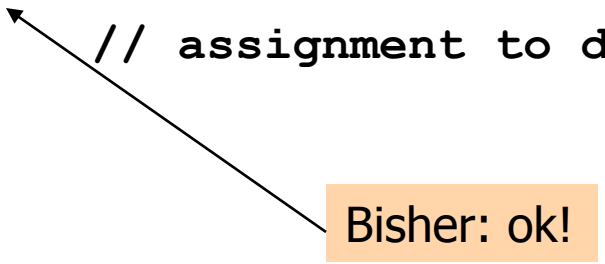


- RAT ruft an: *nichts geht mehr!*
- Problem:
  - Anwendungscode enthält jede Menge **`.n's`** und **`.d's`**
  - ...aber die bedeuten jetzt *etwas anderes* (Absolutwerte von Zähler und Nenner)
  - RAT's Anwendung kompiliert zwar noch, berechnet aber Unsinn

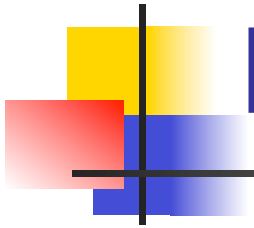
Datenkapselung: Motivation Problem 3 (das eigentliche)

- RAT ruft an: *nichts geht mehr!*
- Beispiel:

```
rational r; // default-initialization of r
r.n = 1;    // assignment to data member
r.d = 2;    // assignment to data member
```



Datenkapselung: Motivation Problem 3 (das eigentliche)

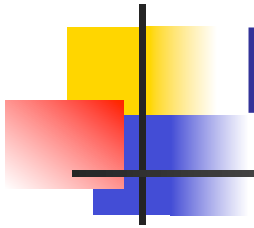


- RAT ruft an: *nichts geht mehr!*
- Beispiel:

```
rational r; // default-initialization of r
r.n = 1;    // assignment to data member
r.d = 2;    // assignment to data member
```

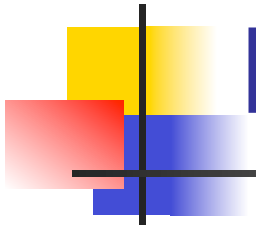
Neu: das neue Datenmitglied **`r.is_negative`** wird nicht initialisiert und hat daher undefinierten Wert; Die Folge: **`r`** ist undefiniert!

Datenkapselung: Motivation Problem 3 (das eigentliche)



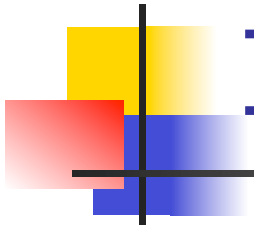
Wenn man dem Kunden erlaubt, auf die interne Repräsentation zuzugreifen, kann man sie in Zukunft nur mit Zustimmung des Kunden ändern!

Datenkapselung: Motivation Problem 3 (das eigentliche)



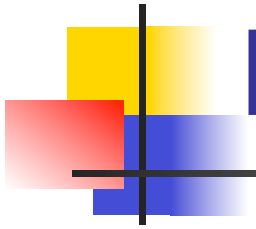
Wenn man dem Kunden erlaubt, auf die interne Repräsentation zuzugreifen, kann man sie in Zukunft nur mit Zustimmung des Kunden ändern!

Die bekommt man nicht, wenn der Kunde dafür seinen Anwendungscode sehr stark umschreiben müsste!



Idee der Datenkapselung

- Ein Typ ist durch Wertebereich und Funktionalität vollständig definiert
- wie die Werte durch Daten-Mitglieder repräsentiert werden, sollte für den Kunden nicht sichtbar sein
- Dem Kunden wird repräsentationsun-abhängige Funktionalität angeboten



Klassen

- sind das Konzept von C++ zur Daten-kapselung
- verallgemeinern Structs
- sind Bestandteil jeder “objektorientier-ten Programmiersprache”

Verstecken der Daten: `public` und `private`

```
class rational {
private:
    int n;
    int d; // INV: d!= 0
};
```

wird anstatt **`struct`** verwendet, wenn überhaupt etwas versteckt werden soll.

Verstecken der Daten: `public` und `private`

```
class rational {
private:
    int n;
    int d; // INV: d!= 0
};
```

wird anstatt **`struct`** verwendet, wenn überhaupt etwas versteckt werden soll.

`struct` : standardmässig wird *nichts* versteckt

`class` : standardmässig wird *alles* versteckt

das ist auch schon der einzige Unterschied!

Verstecken der Daten: `public` und `private`

```
class rational {
private:
    int n;
    int d; // INV: d!= 0
};
```

alles, was nicht **`public:`** ist, ist für den Kunden versteckt, kann also nicht über Mitgliedszugriff benutzt werden.

Verstecken der Daten: `public` und `private`

```
class rational {
private:
    int n;
    int d; // INV: d!= 0
};
```

alles, was nicht **`public:`** ist, ist für den Kunden versteckt, kann also nicht über Mitgliedszugriff benutzt werden.

Anwendungscode:

```
rational r;
r.n = 1;      // error: n is private
r.d = 2;      // error: d is private
int i = r.n; // error: n is private
```

Verstecken der Daten: `public` und `private`

```
class rational {
private:
    int n;
    int d; // INV: d!= 0
};
```

alles, was nicht `public:` ist, ist für den Kunden versteckt, kann also nicht über Mitgliedszugriff benutzt werden.

Anwendungscode:

Gute Nachricht: `r.d = 0;` geht nicht mehr!

```
rational r;
r.n = 1;      // error: n is private
r.d = 2;      // error: d is private
int i = r.n; // error: n is private
```

Verstecken der Daten: `public` und `private`

```
class rational {
private:
    int n;
    int d; // INV: d!= 0
};
```

alles, was nicht `public:` ist, ist für den Kunden versteckt, kann also nicht über Mitgliedszugriff benutzt werden.

Anwendungscode:

Gute Nachricht: `r.d = 0;` geht nicht mehr!

```
rational r;
r.n = 1;      // error: n is private
r.d = 2;      // error: d is private
int i = r.n; // error: n is private
```

Schlechte Nachricht: der Kunde kann nun gar nichts mehr machen...

Verstecken der Daten: `public` und `private`

```
class rational {
private:
    int n;
    int d; // INV: d!= 0
};
```

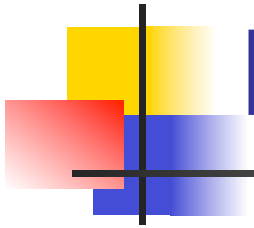
alles, was nicht `public:` ist, ist für den Kunden versteckt, kann also nicht über Mitgliedszugriff benutzt werden.

Anwendungscode:

Gute Nachricht: `r.d = 0;` geht nicht mehr!

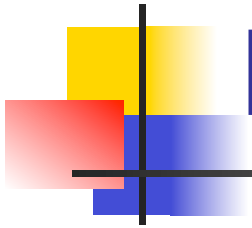
```
rational r;
r.n = 1;      // error: n is private
r.d = 2;      // error: d is private
int i = r.n; // error: n is private
```

Schlechte Nachricht: ...und wir auch nicht (kein `operator+` mit `.n, .d`)



Mitglieds-Funktionen

- erlauben kontrollierten Zugriff auf die versteckten Daten



Mitglieds-Funktionen

- erlauben kontrollierten Zugriff auf die versteckten Daten

```
class rational {
public:
   // POST: return value is the numerator of *this
   int numerator () const
   {
     return n;
   }
   // POST: return value is the denominator of *this
   int denominator () const
   {
     return d;
   }
private:
   int n;
   int d; // INV: d!= 0
};
```

öffentlicher Bereich

Mitglieds-Funktionen

- erlauben kontrollierten Zugriff auf die versteckten Daten

```
class rational {
public:
   // POST: return value is the numerator of *this
   int numerator () const
   {
     return n;
   }
   // POST: return value is the denominator of *this
   int denominator () const
   {
     return d;
   }
private:
   int n;
   int d; // INV: d!= 0
};
```

Anwendungscode:

```
int n = r.numerator();     // get numerator of r
int d = r.denominator();   // get denominator of r
```

Mitglieds-Funktionen

- erlauben kontrollierten Zugriff auf die versteckten Daten

```
class rational {
public:
   // POST: return value is the numerator of *this
   int numerator () const
   {
     return n;
   }
   // POST: return value is the denominator of *this
   int denominator () const
   {
     return d;
   }
private:
   int n;
   int d; // INV: d!= 0
};
```

Mitglieds-Funktion

Mitglieds-Zugriff

Anwendungscode:

```
int n = r.numerator();     // get numerator of r
int d = r.denominator();   // get denominator of r
```

Mitglieds-Funktionen

- erlauben kontrollierten Zugriff auf die versteckten Daten

```
class rational {
public:
   // POST: return value is the numerator of *this
   int numerator () const
   {
     return n;
   }
   // POST: return value is the denominator of *this
   int denominator () const
   {
     return d;
   }
private:
   int n;
   int d; // INV: d!= 0
};
```

Mitgliedsfunktionen haben Zugriff auf private Daten!

Mitglieds-Funktion

Mitglieds-Zugriff

Anwendungscode:

```
int n = r.numerator();     // get numerator of r
int d = r.denominator();   // get denominator of r
```

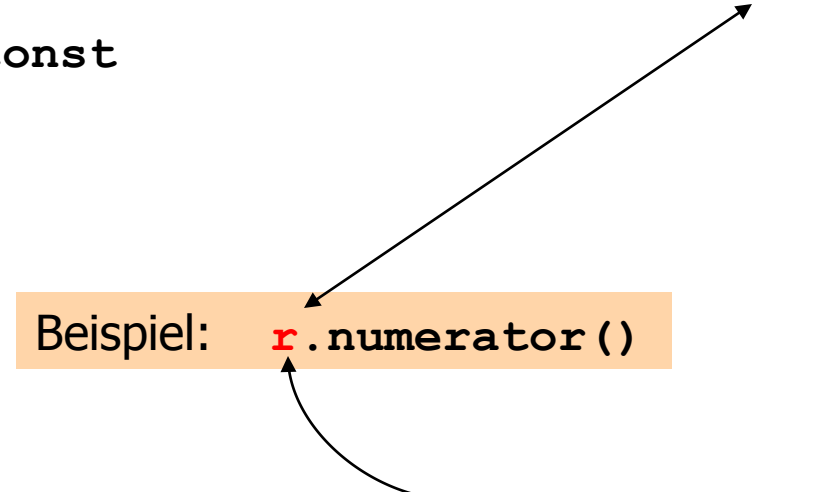
Mitglieds-Funktionen: der implizite Parameter

```
// POST: return value is the numerator of *this
int numerator () const
{
  return n;
}
```

Eine Mitgliedsfunktion wird *für* einen Ausdruck der Klasse aufgerufen (der Ausdruck vor dem Mitglieds-Zugriff-Operator); dieser Ausdruck ist ein implizites Argument des Aufrufs und wird mit `*this` bezeichnet.

Mitglieds-Funktionen: der implizite Parameter

```
// POST: return value is the numerator of *this
int numerator () const
{
  return n;
}
```



Eine Mitgliedsfunktion wird *für* einen Ausdruck der Klasse aufgerufen (der Ausdruck vor dem Mitglieds-Zugriff-Operator); dieser Ausdruck ist ein implizites Argument des Aufrufs und wird mit `*this` bezeichnet.

Mitglieds-Funktionen und `const`

```
// POST: return value is the numerator of *this
int numerator () const
{
  return n;
}
```

Das **`const`** hinter der Deklaration einer Mitglieds-Funktions bezieht sich auf das implizite Argument **`*this`**, verspricht also, dass dieser durch den Aufruf nicht im Wert modifiziert wird.

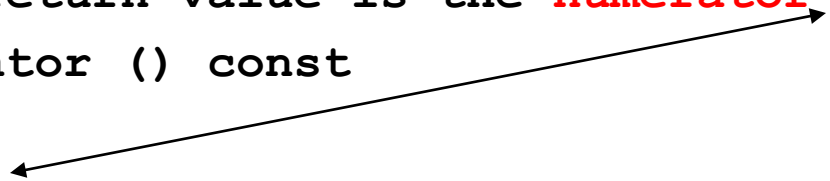
Mitglieds-Funktionen und Zugriff auf Datenmitglieder

```
// POST: return value is the numerator of *this
int numerator () const
{
  return n;
}
```

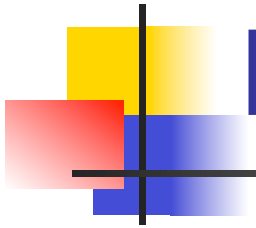
Im Rumpf einer Mitglieds-Funktion bezieht sich der Name eines Daten-Mitglieds auf das entsprechende Daten-Mitglied des impliziten Arguments **`*this`**

Mitglieds-Funktionen und Zugriff auf Datenmitglieder

```
// POST: return value is the numerator of *this
int numerator () const
{
  return n;
}
```

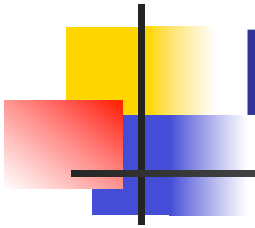


Im Rumpf einer Mitglieds-Funktion bezieht sich der Name eines Daten-Mitglieds auf das entsprechende Daten-Mitglied des impliziten Arguments **`*this`**



Konstruktoren

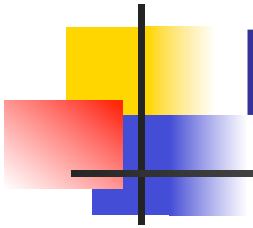
- sind spezielle Mitglieds-Funktionen, “zuständig” für die Initialisierung
- haben den Namen der Klasse und keinen Rückgabetyp



Konstruktoren

- sind spezielle Mitglieds-Funktionen, “zuständig” für die Initialisierung
- haben den Namen der Klasse und keinen Rückgabetyp

```
// PRE: d != 0
// POST: *this is initialized with numerator / denominator
rational (const int numerator, const int denominator)
  : n (numerator), d (denominator)
{
  assert (d != 0);
}
```



Konstruktoren

- sind spezielle Mitglieds-Funktionen, “zuständig” für die Initialisierung
- haben den Namen der Klasse und keinen Rückgabetyp

```
// PRE: d != 0
// POST: *this is initialized with numerator / denominator
rational (const int numerator, const int denominator)
  : n (numerator), d (denominator)
{
  assert (d != 0);
}
```

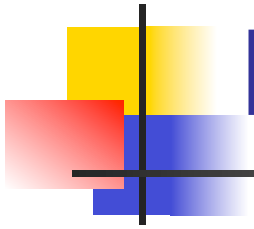
es gibt im allgemeinen mehrere Konstruktoren (des gleichen Namens), der Compiler findet jeweils den passenden.

Konstruktoren

- sind spezielle Mitglieds-Funktionen, "zuständig" für die Initialisierung
- haben den Namen der Klasse und keinen Rückgabetyp

```
// PRE: d != 0
// POST: *this is initialized with numerator / denominator
rational (const int numerator, const int denominator)
  : n (numerator), d (denominator)
{
  assert (d != 0);
}
```

Initialisierer: Daten-Mitglieder werden (in der Reihenfolge ihrer Deklaration) initialisiert



Konstruktoren: Aufruf

- implizit:

```
rational r (1,2); // initializes r with value 1/2
```

Konstruktoren: Aufruf

- implizit:

```
rational r (1,2); // initializes r with value 1/2
```

Konstruktor mit passender formaler Argumentliste wird ausgewählt; neue Variable **`r`** dient als **`*this`**

Konstruktoren: Aufruf

- implizit:

```
rational r (1,2); // initializes r with value ½
```

Konstruktor mit passender formaler Argumentliste wird ausgewählt; neue Variable **`r`** dient als **`*this`**

- explizit:

```
rational r = rational (1, 2);
```

Konstruktoren: Aufruf

- implizit:

```
rational r (1,2); // initializes r with value ½
```

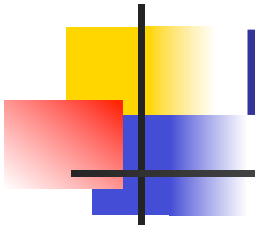
Konstruktor mit passender formaler Argumentliste wird ausgewählt; neue Variable **`r`** dient als **`*this`**

- explizit:

Konstruktor mit passender formaler Argumentliste wird ausgewählt; ein neues **`*this`** wird erzeugt und dient als Rückgabewert

```
rational r = rational (1, 2);
```

Audruck vom Typ **`rational`**

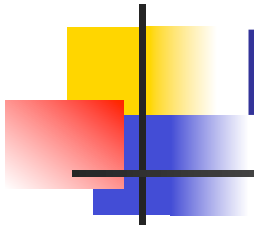


Der Default-Konstruktor

- wird automatisch aufgerufen bei Deklarationen der Form

  ```
  rational r;
  ```

- ist der eindeutige Konstruktor mit leerer Argumentliste (falls existent)
- muss existieren, wenn `rational r;` kompilieren soll!



Der Default-Konstruktor

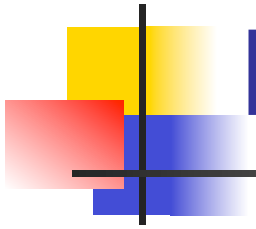
- wird automatisch aufgerufen bei Deklarationen der Form

```
rational r;
```

- ist der eindeutige Konstruktor mit leerer Argumentliste (falls existent)
- muss existieren, wenn **`rational r;`** kompilieren soll!

Gute Nachricht: es gibt keine uninitialisierten Variablen mit Klassentyp!

Der Default-Konstruktor



- wird automatisch aufgerufen bei Deklarationen der Form

```
rational r;
```

- ist der eindeutige Konstruktor mit leerer Argumentliste (falls existent)
- muss existieren, wenn `rational r;` kompilieren soll!

Schlechte Nachricht: für Klassen ohne Konstruktoren wird der Default-Konstruktor ausnahmsweise automatisch erzeugt (wegen Sprache C)

Der Default-Konstruktor

- wird automatisch aufgerufen bei Deklarationen der Form

```
rational r;
```

- ist der eindeutige Konstruktor mit leerer Argumentliste (falls existent)
- muss existieren, wenn **`rational r;`** kompilieren soll!

`struct rational`

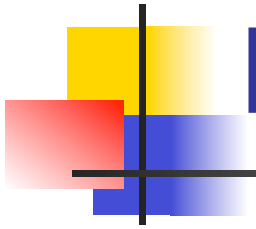
Schlechte Nachricht: für Klassen ohne Konstruktoren wird der Default-Konstruktor ausnahmsweise automatisch erzeugt (wegen Sprache C)

Der Default-Konstruktor für die Klasse `rational`

- initialisiert eine rationale Zahl sinnvollerweise mit Wert 0

```
// POST: *this is initialized with 0
rational ()
  : n (0), d (1)
{}
```

Benutzerdefinierte Konversionen



- sind definiert durch Konstruktoren mit *genau einem* Argument

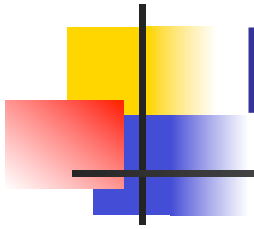
Benutzerdefinierte Konversionen

- sind definiert durch Konstruktoren mit *genau einem* Argument

```
// POST: *this is initialized with value i
rational (int i)
  : n (i), d (1)
{}
```

Benutzerdefinierte Konversion von **`int`** nach **`rational`**. Damit wird **`int`** zu einem Typ, "dessen Werte nach **`rational`** konvertierbar sind";

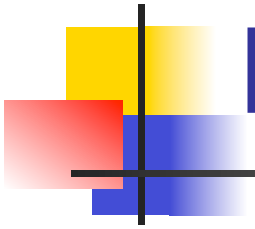
Benutzerdefinierte Konversionen



- sind definiert durch Konstruktoren mit *genau einem* Argument

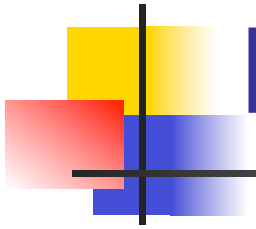
```
// POST: *this is initialized with value i
rational (int i)
  : n (i), d (1)
{}

rational r = 2; // implizite Konversion
```

Benutzerdefinierte Konversionen



Wie kann man implizite Konversion von **`rational`** nach **`double`** realisieren?

Benutzerdefinierte Konversionen



Wie kann man implizite Konversion von **`rational`** nach **`double`** realisieren?

- Problem: **`double`** ist keine Klasse, wir können dem Typ keinen Konstruktor “verpassen” (gilt auch für alle anderen Zieltypen, die nicht “uns” gehören)

Benutzerdefinierte Konversionen

Wie kann man implizite Konversion von **`rational`** nach **`double`** realisieren?

- Lösung: wir bringen "unserem" Typ **`rational`** die Konversion nach **`double`** bei (als Mitglieds-Funktion):

```
operator double ()
{
  return double(n)/d;
}
```

impliziter Rückgabetyp **`double`**

Benutzerdefinierte Konversionen

Wie kann man implizite Konversion von **`rational`** nach **`double`** realisieren?

- Lösung: wir bringen "unserem" Typ **`rational`** die Konversion nach **`double`** bei (als Mitglieds-Funktion):

```
operator double ()
{
  return double(n)/d;
}

double x = rational (1, 2); // implizite Konversion
```

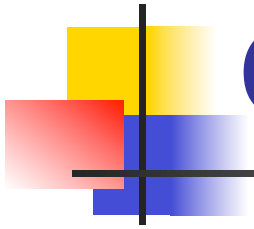
impliziter Rückgabetyp **`double`**

Geschachtelte Typen

- Datenkapselung geht oft nicht weit genug:

```
// POST: return value is the numerator of *this
int numerator () const
{
  return n;
}
```

hier legen wir uns auf den Wertebereich (**`int × int`**) fest, können ihn also nicht ohne potentielle Änderungen im Anwendungscode erweitern, z.B. auf **`ifm::integer`**!



Geschachtelte Typen

Idee:

- der *Typ* von Zähler und Nenner ist für den Kunden nicht sichtbar
- dem Kunden werden Eigenschaften des Typs garantiert, und er kann den Typ repräsentationsunabhängig benutzen

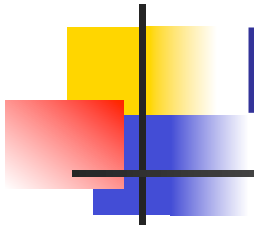
Geschachtelte Typen

Idee:

- der *Typ* von Zähler und Nenner ist für den Kunden nicht sichtbar
- dem Kunden werden Eigenschaften des Typs garantiert, und er kann den Typ repräsentationsunabhängig benutzen

z.B. : “integraler Typ, in den Werte vom Typ `int` konvertierbar sind”

Geschachtelte Typen: Realisierung



```
class rational {
public:
    // nested type for numerator and denominator
    typedef int rat_int;

    ...
    // realize all functionality in terms of "rat_int"
    // instead of "int"
    ...

private:
    rat_int n;
    rat_int d; // INV: d!= 0
};
```

Geschachtelte Typen: Realisierung

`rat_int` wird *Synonym* für `int` (kein neuer Typ!)

```
class rational {
public:
    // nested type for numerator and denominator
    typedef int rat_int;

    ...
    // realize all functionality in terms of "rat_int"
    // instead of "int"
    ...

private:
    rat_int n;
    rat_int d; // INV: d!= 0
};
```

Geschachtelte Typen: Realisierung

`rat_int` wird *Synonym* für `int` (kein neuer Typ!)

```
class rational {
public:
    // nested type for numerator and denominator
    typedef int rat_int;

    ...
    // realize all functionality in terms of "rat_int"
    // instead of "int"
    ...

private:
    rat_int n;
    rat_int d; // INV: d!= 0
};
```

Anwendungscode:

```
typedef rational::rat_int rat_int;
rational r (1,2);
rat_int numerator = r.numerator();        // 1
rat_int denominator = r.denominator();    // 2
```

Geschachtelte Typen: Realisierung

```
class rational {
public:
    // nested type for numerator and denominator
    typedef int rat_int;

    ...
    // realize all functionality in terms of "rat_int"
    // instead of "int"
    ...

private:
    rat_int n;
    rat_int d; // INV: d!= 0
};
```

rat_int wird *Synonym* für **int** (kein neuer Typ!)

int-Werte konvertierbar nach **rat_int**

Anwendungscode:

```
typedef rational::rat_int rat_int;
rational r (1,2);
rat_int numerator = r.numerator();        // 1
rat_int denominator = r.denominator();    // 2
```

Geschachtelte Typen: Realisierung

So geht's dann auch!

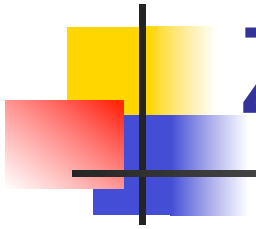
```
class rational {
public:
    // nested type for numerator and denominator
    // typedef int rat_int;
    typedef ifm::integer rat_int;
    ...
    // realize all functionality in terms of "rat_int"
    // instead of "int"
    ...

private:
    rat_int n;
    rat_int d; // INV: d!= 0
};
```

int-Werte konvertierbar nach **rat_int**

Anwendungscode:

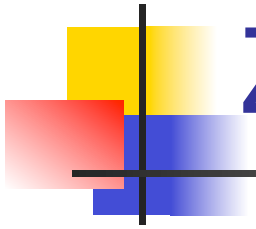
```
typedef rational::rat_int rat_int;
rational r (1,2);
rat_int numerator = r.numerator();        // 1
rat_int denominator = r.denominator();    // 2
```

Klassen in Aktion: Zufallszahlen



- Viele Algorithmen setzen den Zufall gezielt ein: *randomisierte Algorithmen*
- Beispiele:
  - Vergleich zweier Rechnerinhalte durch ein Protokoll mit wenig Datenaustausch
  - Mehrere “Algorithmen der Woche” (im Rahmen *Informatikjahr 2006* (D))

Klassen in Aktion: Zufallszahlen



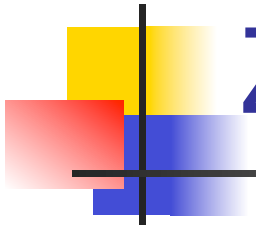
Für Computerspiele brauchen wir

unvorhersehbares Verhalten

des Computers:

- Schachprogramm sollte auf meine immer gleichen Züge abwechselnd reagieren
- Feinde im Aktionsspiel sollten nicht immer zur gleichen Zeit und an der gleichen Stelle auftauchen

Klassen in Aktion: Zufallszahlen



Für Computerspiele brauchen wir

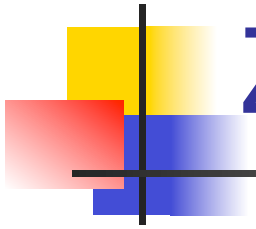
unvorhersehbares Verhalten

des Computers:

Unvorhersehbarkeit dank mangelnden Wissens genügt, absolute Unvorhersehbarkeit ist nicht nötig

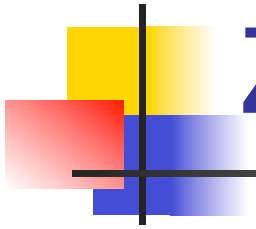
- Schachprogramm sollte auf meine immer gleichen Züge abwechselnd reagieren
- Feinde im Aktionsspiel sollten nicht immer zur gleichen Zeit und an der gleichen Stelle auftauchen

Klassen in Aktion: Zufallszahlen



- Programme erzeugen unvorhersehbares Verhalten mit *Zufallszahlengenerator...*

Klassen in Aktion: Zufallszahlen



- Programme erzeugen unvorhersehbares Verhalten mit *Zufallszahlengenerator...*
- ...aber nach einer *festen* Regel

Wir erhalten dabei Pseudozufallszahlen

Die lineare Kongruenzmethode

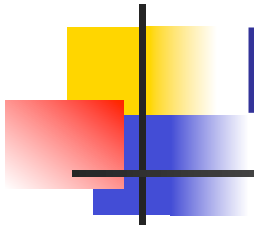
- erzeugt Folge von Pseudozufallszahlen, abhängig von vier natürlichen Zahlen
  - a (der Multiplikator)
  - c (die Verschiebung)
  - m (der Modulus)
  - x_0 (der Keim)

Die lineare Kongruenzmethode

- erzeugt Folge von Pseudozufallszahlen, abhängig von vier natürlichen Zahlen
  - a (der Multiplikator)
  - c (die Verschiebung)
  - m (der Modulus)
  - x_0 (der Keim)

$$x_i := (a \cdot x_{i-1} + c) \bmod m, \quad i > 0$$

Die lineare Kongruenzmethode Beispiel:



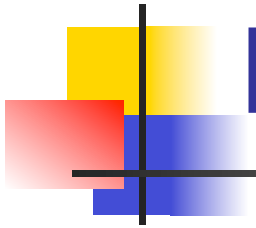
$a = 137;\ \ c = 187;\ \ m = 2^8 = 256;\ \ x_0 = 0$

Die lineare Kongruenzmethode Beispiel:

$a = 137;\ c = 187;\ m = 2^8 = 256;\ x_0 = 0$

- $x_1, x_2, \ldots$ = 187, 206, 249, 252, 151, 138, 149, 120, 243, 198, 177, 116, 207, 130, 77, 240, 43, 190, 105, 236, 7, 122, 5, 104, 99, 182, 33, 100, 63, 114, 189, 224, 155, 174, 217, 220, 119, 106, 117, 88, 211, 166, 145, 84, 175, 98, 45, 208, 11, 158, 73, 204, 231, 90, 229, 72, 67, 150, 1, 68, 31, 82, 157, 192, 123, 142, 185, 188, 87, 74, 85, 56, 179, 134, 113, 52, 143, 66, 13, 176, 235, 126, 41, 172, 199, 58, 197, 40, 35, 118, 225, 36, 255, 50, 125, 160, 91, 110, 153, 156, 55, 42, 53, 24, 147, 102, 81, 20, 111, 34, 237, 144, 203, 94, 9, 140, 167, 26, 165, 8, 3, 86, 193, 4, 223, 18, 93, 128, 59, 78, 121, 124, 23, 10, 21, 248, 115, 70, 49, 244, 79, 2, 205, 112, 171, 62, 233, 108, 135, 250, 133, 232, 227, 54, 161, 228, 191, 242, 61, 96, 27, 46, 89, 92, 247, 234, 245, 216, 83, 38, 17, 212, 47, 226, 173, 80, 139, 30, 201, 76, 103, 218, 101, 200, 195, 22, 129, 196, 159, 210, 29, 64, 251, 14, 57, 60, 215, 202, 213, 184, 51, 6, 241, 180, 15, 194, 141, 48, 107, 254, 169, 44, 71, 186, 69, 168, 163, 246, 97, 164, 127, 178, 253, 32, 219, 238, 25, 28, 183, 170, 181, 152, 19, 230, 209, 148, 239, 162, 109, 16, 75, 222, 137, 12, 39, 154, 37, 136, 131, 214, 65, 132, 95, 146, 221, 0, 187, ...

Die lineare Kongruenzmethode Normalisierung



Man erhält reelle Pseudozufallszahlen im halboffenen Intervall [0, 1) als Folge

$x_1/m, x_2/m, x_3/m, \ldots$

Eine Zufallszahlen-Bibliothek: Die Header-Datei `random.h`

```
// Prog: random.h
// define a class for pseudorandom numbers.

namespace ifm {
  // class random: definition
  class random {
  public:
    // POST: *this is initialized with the linear congruential
    //       random number generator
    //           x_i = ( a * x_{i-1} + c) mod m
    //       with seed x0.
    random (unsigned int a, unsigned int c,
            unsigned int m, unsigned int x0);

    // POST: return value is the next pseudorandom number
    //       in the sequence of the x_i, divided by m
    double operator()();

  private:
    const unsigned int a_; // multiplier
    const unsigned int c_; // offset
    const unsigned int m_; // modulus
    unsigned int xi_;      // current sequence element
  };
} // end namespace ifm
```

Klasse zur Repräsentation einer Folge $x_0, x_1, x_2, \ldots$ von Pseudozufallszahlen

Eine Zufallszahlen-Bibliothek: Die Header-Datei `random.h`

```
// Prog: random.h
// define a class for pseudorandom numbers.

namespace ifm {
  // class random: definition
  class random {
  public:
    // POST: *this is initialized with the linear congruential
    //       random number generator
    //           x_i = ( a * x_{i-1} + c) mod m
    //       with seed x0.
    random (unsigned int a, unsigned int c,
            unsigned int m, unsigned int x0);

    // POST: return value is the next pseudorandom number
    //       in the sequence of the x_i, divided by m
    double operator()();

  private:
    const unsigned int a_; // multiplier
    const unsigned int c_; // offset
    const unsigned int m_; // modulus
    unsigned int xi_;      // current sequence element
  };
} // end namespace ifm
```

Konstruktor (Deklaration): initialisiert den Generator mit a, c, m, x_0

Eine Zufallszahlen-Bibliothek: Die Header-Datei `random.h`

```
// Prog: random.h
// define a class for pseudorandom numbers.

namespace ifm {
  // class random: definition
  class random {
  public:
    // POST: *this is initialized with the linear congruential
    //       random number generator
    //           x_i = ( a * x_{i-1} + c) mod m
    //       with seed x0.
    random (unsigned int a, unsigned int c,
            unsigned int m, unsigned int x0);

    // POST: return value is the next pseudorandom number
    //       in the sequence of the x_i, divided by m
    double operator()();

  private:
    const unsigned int a_; // multiplier
    const unsigned int c_; // offset
    const unsigned int m_; // modulus
    unsigned int xi_;      // current sequence element
  };
} // end namespace ifm
```

Mitglieds-Funktion zur Ausgabe der jeweils nächsten Pseudozufallszahl x_i/m

Eine Zufallszahlen-Bibliothek: Die Header-Datei `random.h`

```
// Prog: random.h
// define a class for pseudorandom numbers.

namespace ifm {
  // class random: definition
  class random {
  public:
    // POST: *this is initialized with the linear congruential
    //       random number generator
    //           x_i = ( a * x_{i-1} + c) mod m
    //       with seed x0.
    random (unsigned int a, unsigned int c,
            unsigned int m, unsigned int x0);

    // POST: return value is the next pseudorandom number
    //       in the sequence of the x_i, divided by m
    double operator()();

  private:
    const unsigned int a_; // multiplier
    const unsigned int c_; // offset
    const unsigned int m_; // modulus
    unsigned int xi_;      // current sequence element
  };
} // end namespace ifm
```

Mitglieds-Funktion zur Ausgabe der jeweils nächsten Pseudozufallszahl x_i/m

überladener *Klammeroperator* (hier: keine Parameter); erlaubt funktionale Notation

Eine Zufallszahlen-Bibliothek: Die Header-Datei `random.h`

```
// Prog: random.h
// define a class for pseudorandom numbers.

namespace ifm {
  // class random: definition
  class random {
  public:
    // POST: *this is initialized with the linear congruential
    //       random number generator
    //           x_i = ( a * x_{i-1} + c) mod m
    //       with seed x0.
    random (unsigned int a, unsigned int c,
            unsigned int m, unsigned int x0);

    // POST: return value is the next pseudorandom number
    //       in the sequence of the x_i, divided by m
    double operator()();

  private:
    const unsigned int a_; // multiplier
    const unsigned int c_; // offset
    const unsigned int m_; // modulus
    unsigned int xi_;      // current sequence element
  };
} // end namespace ifm
```

Datenmitglieder zur Speicherung der Werte a, c, m und x_i (initial gleich x_0)

Eine Zufallszahlen-Bibliothek: Die Implementierung `random.C`

```
// Prog: random.cpp
// implement a class for pseudorandom numbers.

#include <IFM/random.h>

namespace ifm {
  // class random: implementation
  random::random(unsigned int a, unsigned int c,
          unsigned int m, unsigned int x0)
    : a_(a), c_(c), m_(m), xi_(x0)
  {}

  double random::operator()()
  {
    // update xi acording to formula,...
    xi_ = (a_ * xi_ + c_) % m_;
    // ...normalize it to [0,1), and return it
    return double(xi_) / m_;
  }
} // end namespace ifm
```

Konstruktor-Definition (initialisiert die Daten-Mitglieder)

Eine Zufallszahlen-Bibliothek: Die Implementierung `random.C`

```
// Prog: random.cpp
// implement a class for pseudorandom numbers.

#include <IFM/random.h>

namespace ifm {
  // class random: implementation
  random::random(unsigned int a, unsigned int c,
          unsigned int m, unsigned int x0)
    : a_(a), c_(c), m_(m), xi_(x0)
  {}

  double random::operator()()
  {
    // update xi acording to formula,...
    xi_ = (a_ * xi_ + c_) % m_;
    // ...normalize it to [0,1), and return it
    return double(xi_) / m_;
  }
} // end namespace ifm
```

Definitionen von Mitglieds-Funktionen ausserhalb der Klassendefinition müssen mit dem Klassennamen *qualifiziert* werden

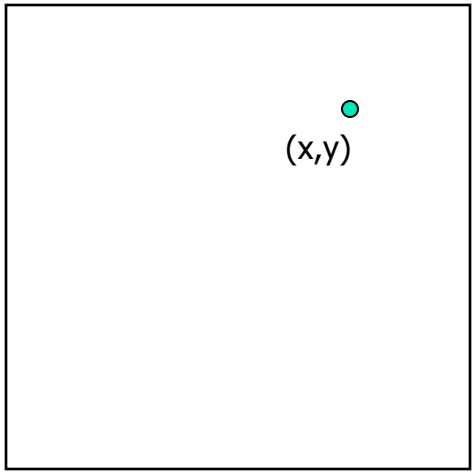
Eine Zufallszahlen-Bibliothek: Die Implementierung `random.C`

```
// Prog: random.cpp
// implement a class for pseudorandom numbers.

#include <IFM/random.h>

namespace ifm {
  // class random: implementation
  random::random(unsigned int a, unsigned int c,
          unsigned int m, unsigned int x0)
    : a_(a), c_(c), m_(m), xi_(x0)
  {}

  double random::operator()()
  {
    // update xi acording to formula,...
    xi_ = (a_ * xi_ + c_) % m_;
    // ...normalize it to [0,1), and return it
    return double (xi_) / m_;
  }
} // end namespace ifm
```

Definition der Mitglieds-Funktion für die nächste Pseudozufallszahl

Anwendung Zufallszahlen: Approximation von π

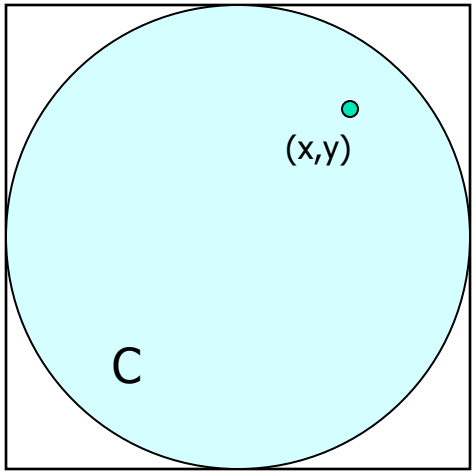
- Wähle zufälligen Punkt (x,y) aus $[0,1]^2$



Anwendung Zufallszahlen: Approximation von π

- Wähle zufälligen Punkt (x,y) aus $[0,1]^2$
- Wahrscheinlichkeit für "(x,y) in C" ist

$$\pi / 4$$



Anwendung Zufallszahlen: Approximation von π

- Wähle zufälligen Punkt (x,y) aus $[0,1]^2$
- Wahrscheinlichkeit für "(x,y) in C" ist

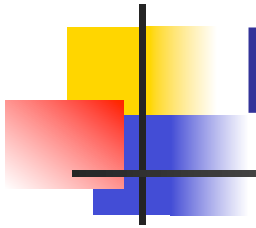
  $$\pi / 4$$

- Bei n-facher Wiederholung des Experiments ist der Anteil der Punkte in C ungefähr $n \cdot \pi / 4$
- π "=" 4 ·Anzahl Punkte in C / n

Anwendung von Zufallszahlen: Das Spiel "Zahlen wählen"

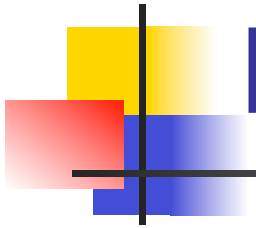
- zwei Spieler A und B schreiben jeweils unabhängig eine Zahl aus {1,...,6} auf
- die Zahlen a und b werden verglichen:
  - (a) a = b: Unentschieden
  - (b) |a-b| = 1: Spieler mit *kleinerer* Zahl gewinnt und erhält CHF 2 vom anderen
  - (c) |a-b| ≥ 2: Spieler mit *grösserer* Zahl gewinnt und erhält CHF 1 vom anderen

Anwendung von Zufallszahlen: Das Spiel “Zahlen wählen”



Wie spielen Sie gegen Ihre Kollegin, wenn Sie 100 Runden vereinbart haben?

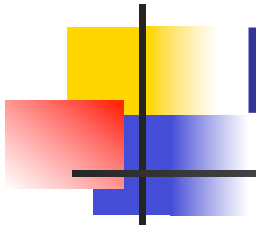
Anwendung von Zufallszahlen: Das Spiel “Zahlen wählen”



Wie spielen Sie gegen Ihre Kollegin, wenn Sie 100 Runden vereinbart haben?

- sicher nicht immer die gleiche Zahl (Kollegin kann Sie dann stets schlagen)

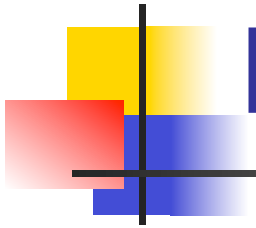
Anwendung von Zufallszahlen: Das Spiel "Zahlen wählen"



Wie spielen Sie gegen Ihre Kollegin, wenn Sie 100 Runden vereinbart haben?

- sicher nicht immer die gleiche Zahl (Kollegin kann Sie dann stets schlagen)
- Jede Zahl mit Wahrscheinlichkeit 1/6 (Würfel) ist besser, Sie können aber auch dann geschlagen werden

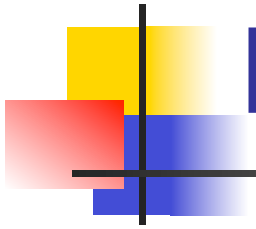
Anwendung von Zufallszahlen: Das Spiel "Zahlen wählen"



Wie spielen Sie gegen Ihre Kollegin, wenn Sie 100 Runden vereinbart haben?

- "gezinkter Würfel": Zahl i wird mit Wahrscheinlichkeit p_i gewählt, $\Sigma\, p_i = 1$
- kann leicht mit "unserer" Bibliothek von Zufallszahlen implementiert werden
- Es gibt unschlagbaren gezinkten Würfel!

Anwendung von Zufallszahlen: Das Spiel "Zahlen wählen"



Wie spielen Sie gegen Ihre Kollegin, wenn Sie 100 Runden vereinbart haben?

- "gezinkter Würfel": Zahl i wird mit Wahrscheinlichkeit p_i gewählt, $\Sigma\, p_i = 1$
- kann leicht mit "unserer" Bibliothek von Zufallszahlen implementiert werden
- Es gibt unschlagbaren gezinkten Würfel!

Es lohnt sich auch finanziell, Informatik zu studieren (Details und Programme im Skript)!