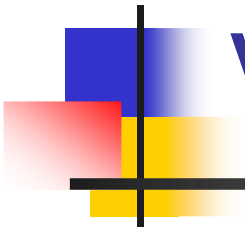
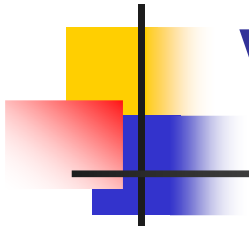


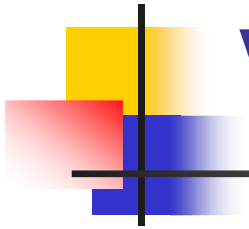
Visuelle Kryptographie

Anwendung von Zufallszahlen



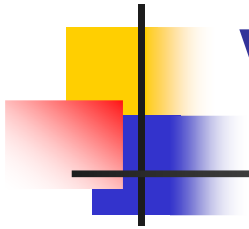
Verschlüsseln eines Bildes

- Wir wollen ein Bild an Alice und Bob schicken, so dass



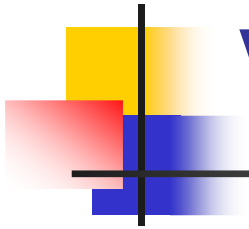
Verschlüsseln eines Bildes

- Wir wollen ein Bild an Alice und Bob schicken, so dass
  - Alice allein keine Information über das Bild bekommt



Verschlüsseln eines Bildes

- Wir wollen ein Bild an Alice und Bob schicken, so dass
  - Alice allein keine Information über das Bild bekommt
  - Bob allein keine Information über das Bild bekommt

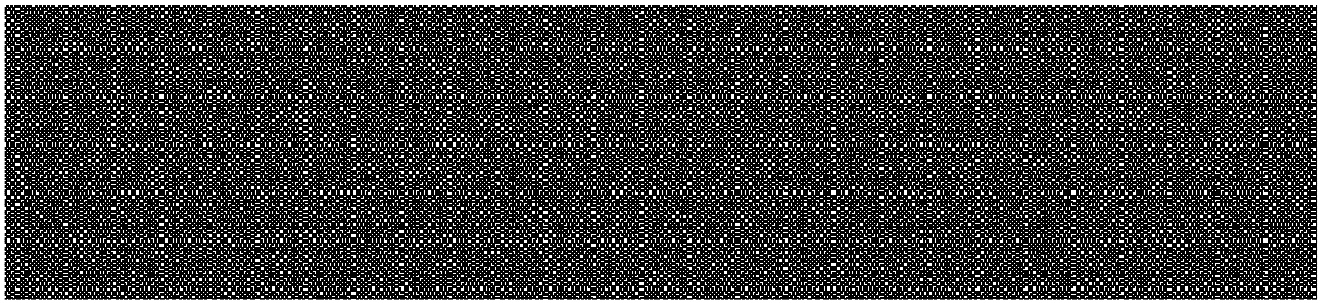


Verschlüsseln eines Bildes

- Wir wollen ein Bild an Alice und Bob schicken, so dass
  - Alice allein keine Information über das Bild bekommt
  - Bob allein keine Information über das Bild bekommt
  - Alice und Bob das Bild *zusammen* aber leicht ansehen können

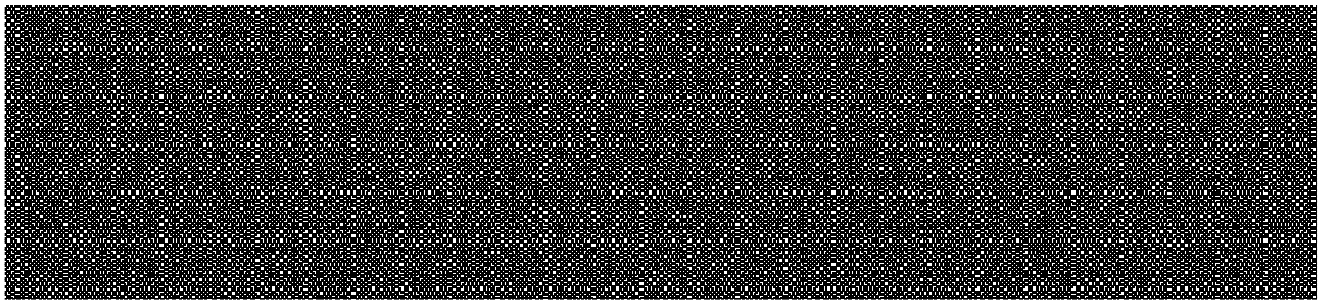
Verschlüsseln eines Bildes

- Das bekommt Alice:

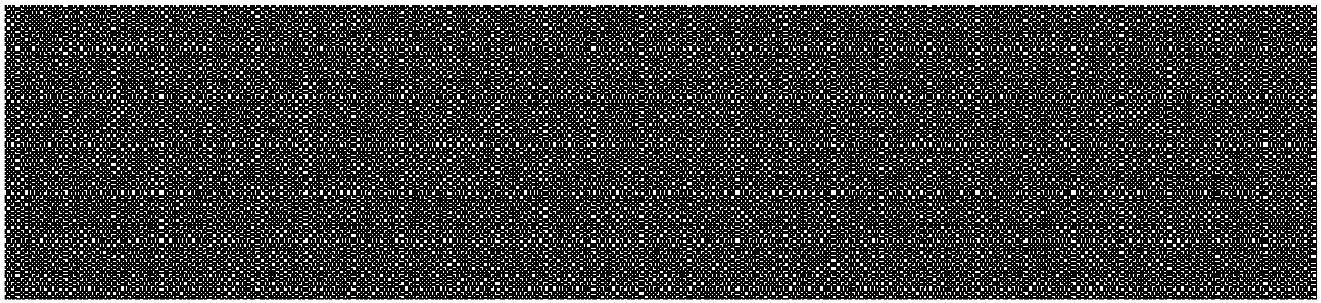


Verschlüsseln eines Bildes

- Das bekommt Alice:



- Das bekommt Bob:



Verschlüsseln eines Bildes

- Übereinanderlegen ergibt:

Verschlüsseln eines Bildes: Wie funktiniert es?

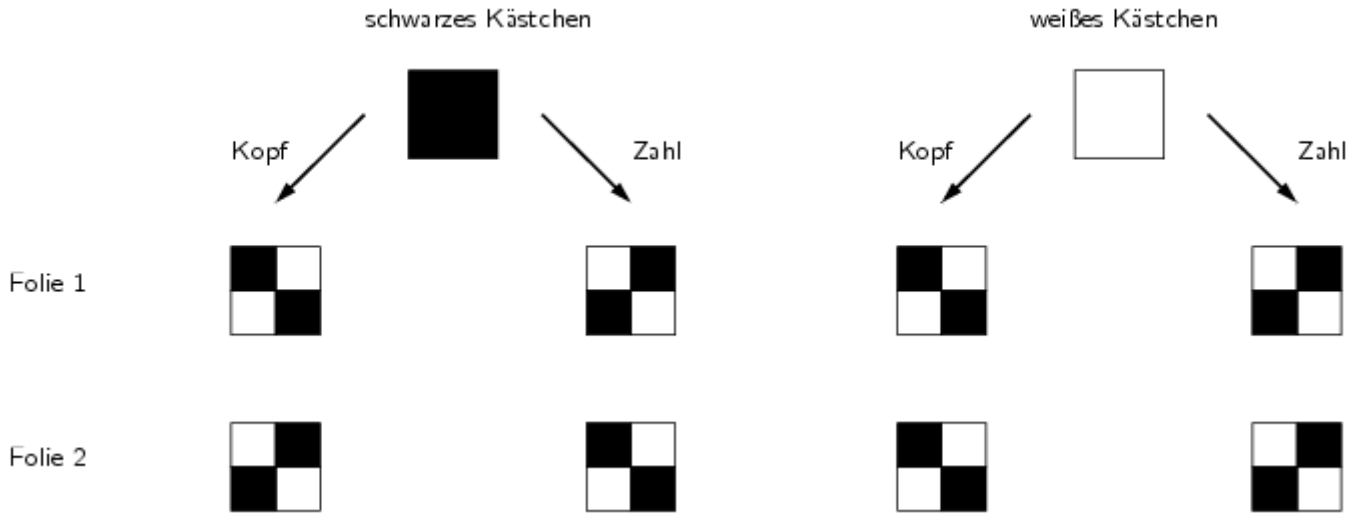
- Das Originalbild wird Pixel für Pixel mit einem Münzwurf verschlüsselt:

Algorithmus

Verschlüsseln eines Bildes: Wie funktiniert es?

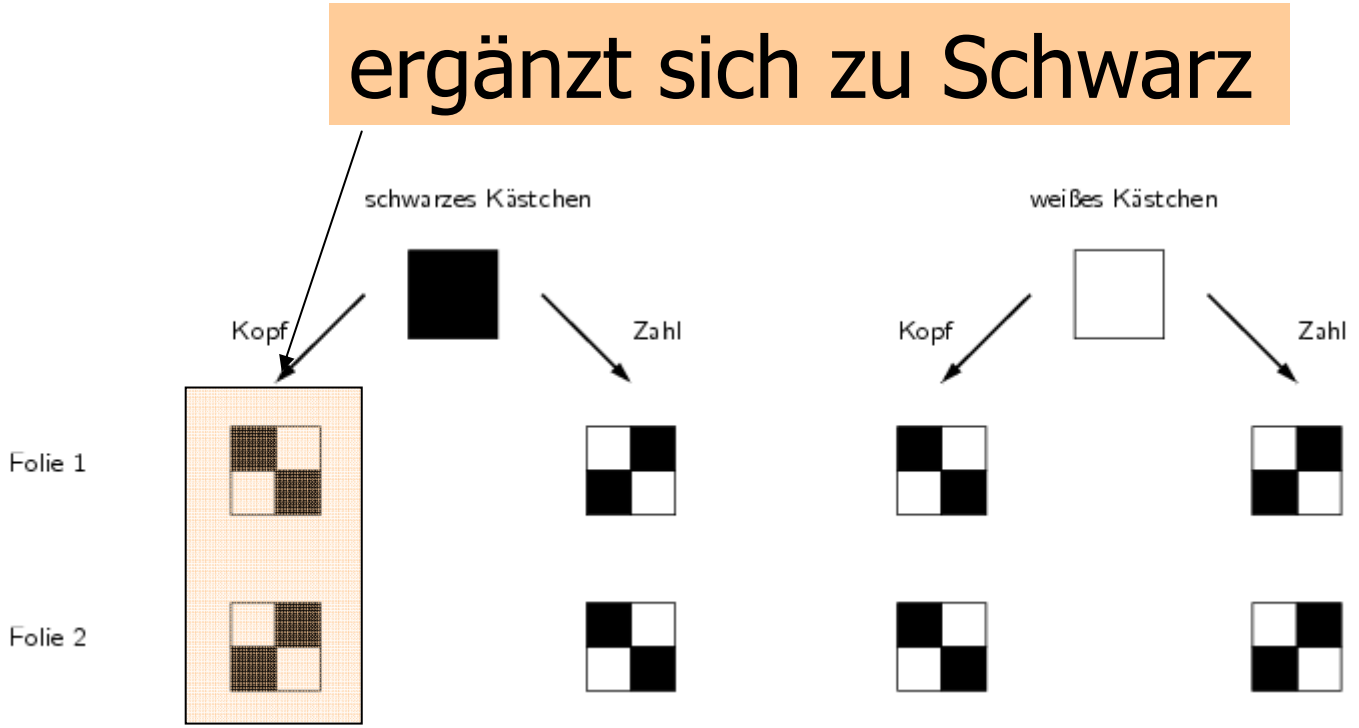
- Das Originalbild wird Pixel für Pixel mit einem Münzwurf verschlüsselt

Algorithmus



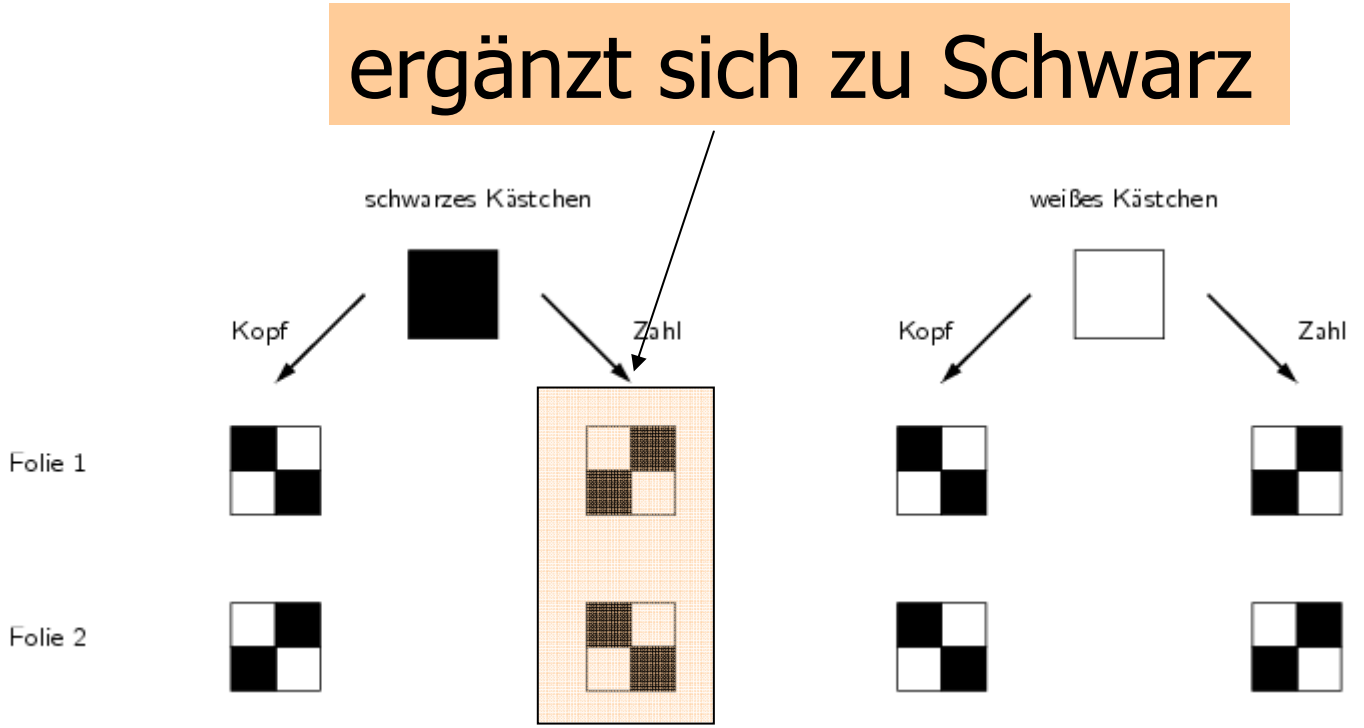
Verschlüsseln eines Bildes: Wie funktiniert es?

- Das Originalbild wird Pixel für Pixel mit einem Münzwurf verschlüsselt



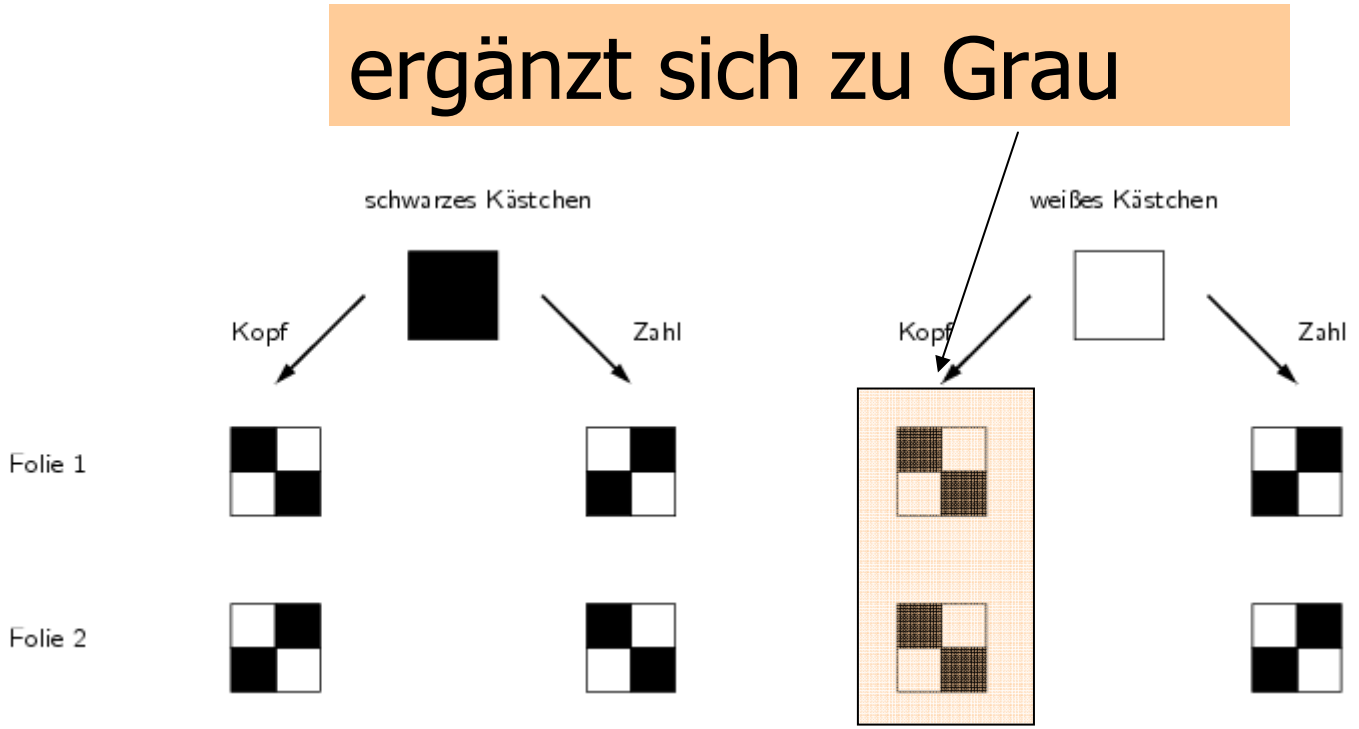
Verschlüsseln eines Bildes: Wie funktiniert es?

- Das Originalbild wird Pixel für Pixel mit einem Münzwurf verschlüsselt



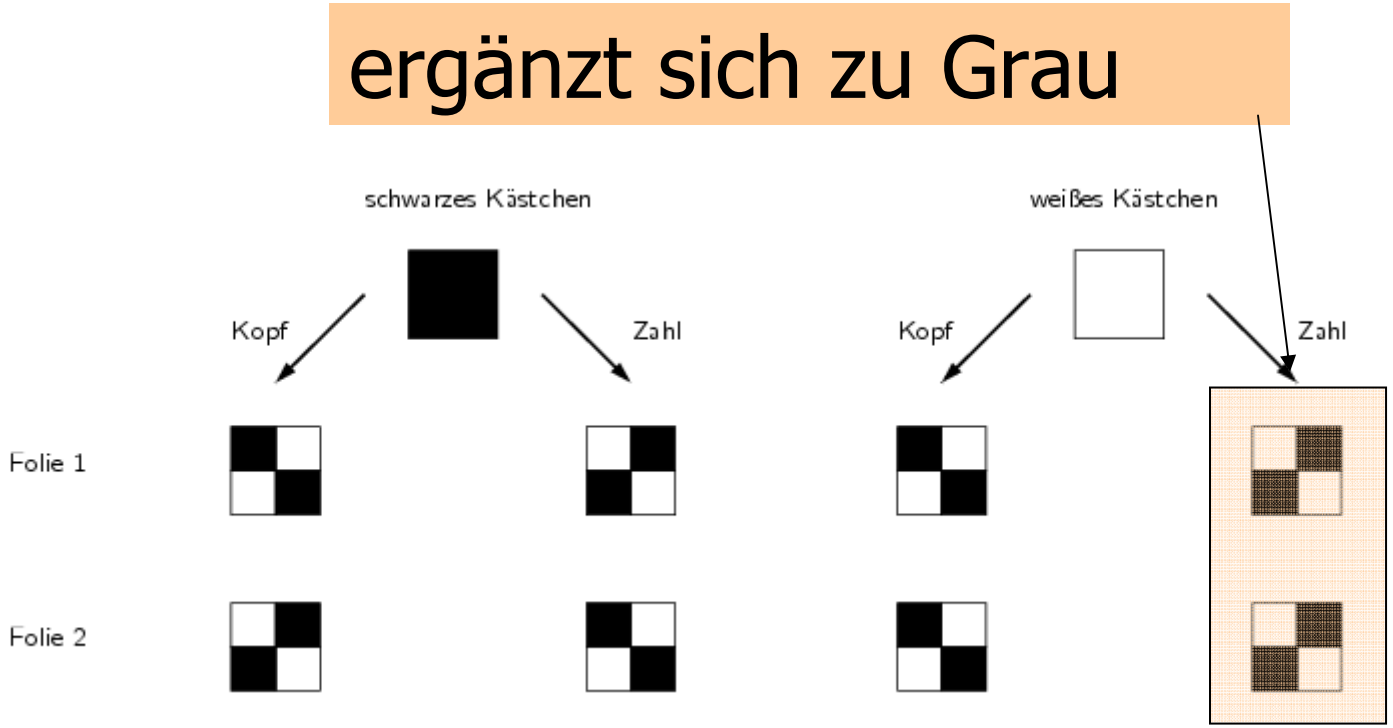
Verschlüsseln eines Bildes: Wie funktiniert es?

- Das Originalbild wird Pixel für Pixel mit einem Münzwurf verschlüsselt



Verschlüsseln eines Bildes: Wie funktiniert es?

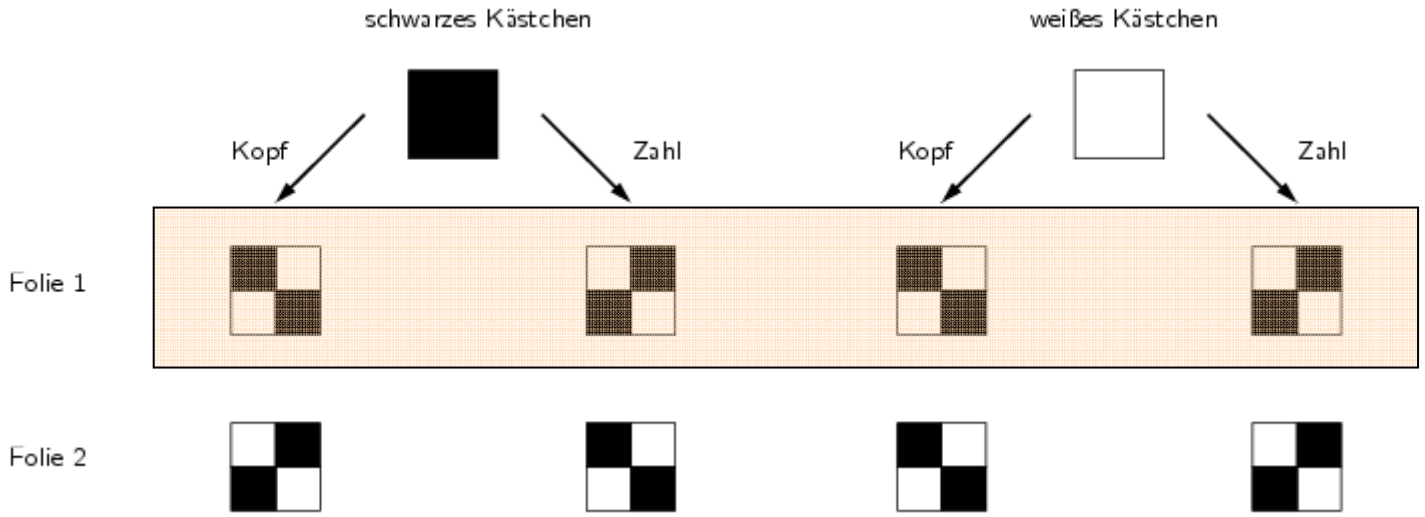
- Das Originalbild wird Pixel für Pixel mit einem Münzwurf verschlüsselt



Verschlüsseln eines Bildes: Wie funktiniert es?

- Das Originalbild wird Pixel für Pixel mit einem Münzwurf verschlüsselt

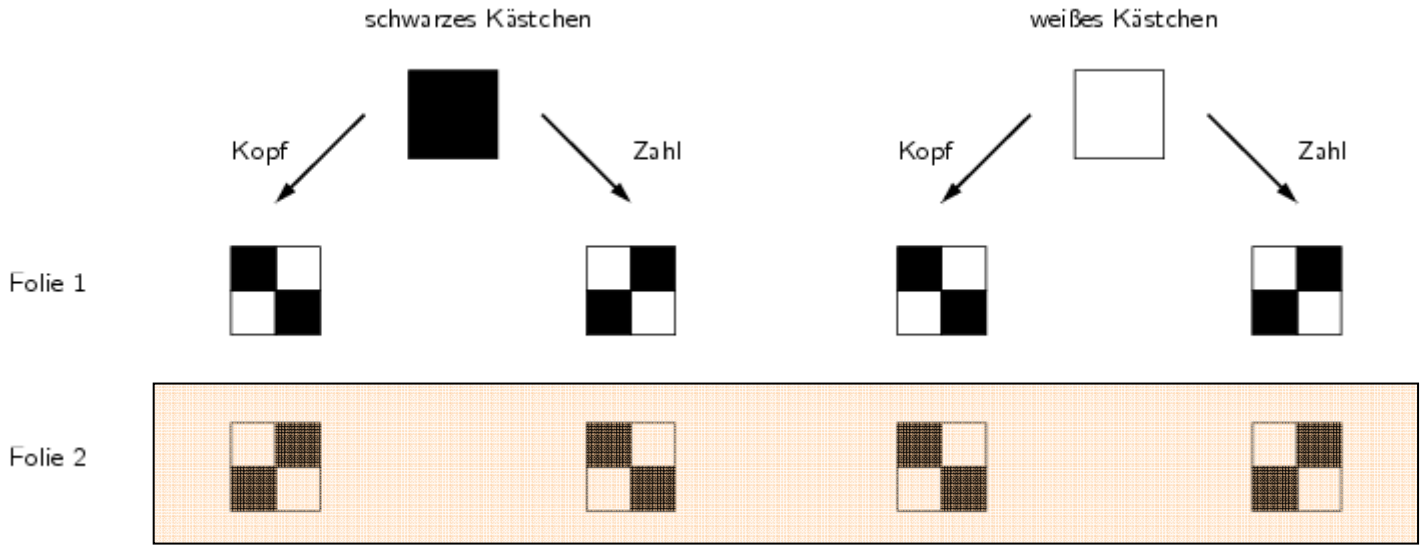
Folie 1 ist völlig zufällig!

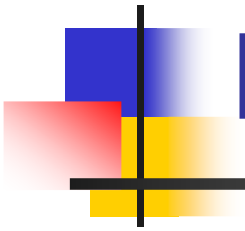


Verschlüsseln eines Bildes: Wie funktiniert es?

- Das Originalbild wird Pixel für Pixel mit einem Münzwurf verschlüsselt

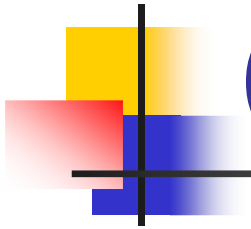
Folie 2 ist völlig zufällig!





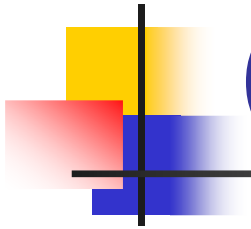
Dynamische Datentypen

Probleme mit Feldern (variabler Länge)



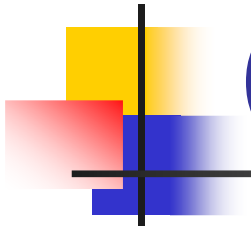
- man kann sie nicht direkt kopieren und zuweisen

Probleme mit Feldern (variabler Länge)



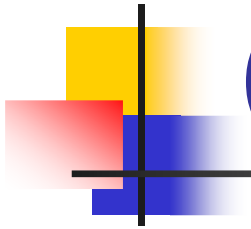
- man kann sie nicht direkt kopieren und zuweisen
- falls Länge nicht zur Kompilierungszeit bekannt, muss Speicher explizit mit **`new`** geholt und mit **`delete[]`** wieder freigegeben werden (fehleranfällig)

Probleme mit Feldern (variabler Länge)



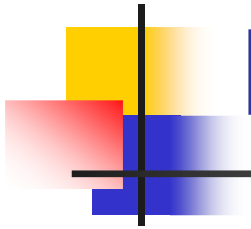
- man kann sie nicht direkt kopieren und zuweisen
- falls Länge nicht zur Kompilierungszeit bekannt, muss Speicher explizit mit **`new`** geholt und mit **`delete[]`** wieder freigegeben werden (fehleranfällig)
- sie können nicht wachsen/schrumpfen

Probleme mit Feldern (variabler Länge): Lösung



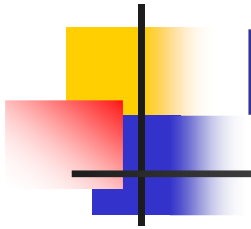
- Eine *Klasse* für Felder
- Realisiert
  - korrekte Initialiserung und Zuweisung
  - Automatische Speicherverwaltung (keine expliziten **`new`** und **`delete[]`** im Kundencode mehr notwendig)

Eine Klasse für Felder: Daten-Mitglieder und Konstruktor



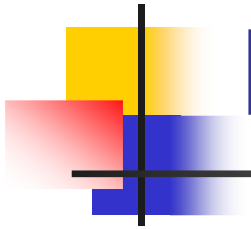
```
class array {
public:
  typedef bool T; // nested type, easy to change

  // --- Constructor from int ---------------------------
  // PRE: n >= 0
  // POST: *this is initialized with an array of length n
  array (int n)
    : ptr (new T[n]), size (n)
  {}

private:
  T* ptr;    // pointer to first element
  int size;  // number of elements
};
```

Eine Klasse für Felder: Random Access



- Indexzugriff mittels Überladung von **`operator[]`**(als Mitglieds-Funktion)

Eine Klasse für Felder: Random Access



- Indexzugriff mittels Überladung von **`operator[]`**(als Mitglieds-Funktion)

```
// --- Index operator (const) -------------------------
// PRE: the array has length at least i
// POST: return value is i-th element (as a copy)
const T& operator[] (int i) const
{
  return ptr[i];
}
```

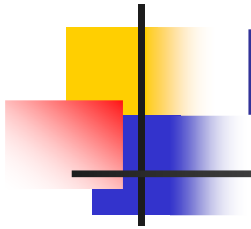
Eine Klasse für Felder: Random Access

- Indexzugriff mittels Überladung von **`operator[]`**(als Mitglieds-Funktion)

```
// --- Index operator (const) -------------------------
// PRE: the array has length at least i
// POST: return value is i-th element (as a copy)
const T& operator[] (int i) const
{
  return ptr[i];
}
```

Problem: erlaubt keinen schreibenden Zugriff wie in **`crossed_out[i] = false;`**

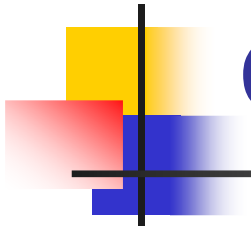
Eine Klasse für Felder: Random Access



- Zwei Überladungen von **`operator[]`**!

```
// --- Index operator (const) ------------------------
// PRE: the array has length at least i
// POST: return value is i-th element (as a copy)
const T& operator[] (int i) const
{
  return ptr[i];
}
// --- Index operator (non-const) ------------------------
// PRE: the array has length at least i
// POST: return value is i-th element (as a reference)
T& operator[] (int i)
{
  return ptr[i];
}
```

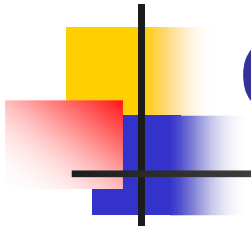
Eine Klasse für Felder: Sieb des Eratosthenes



```
int main()
{
  // input of n
  std::cout << “Compute prime numbers in [2, n) for n = ?”;
  unsigned int n;
  std::cin >> n;

  // definition and initialization: provides us with
  // Booleans crossed_out[0],..., crossed_out[n-1]
  bool* crossed_out = new bool[n];
  ...
  ...
  delete[] crossed_out;
  return 0;
}
```

Bisher

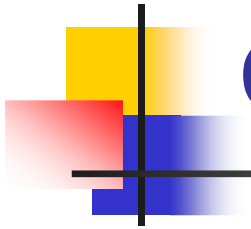
Eine Klasse für Felder: Sieb des Eratosthenes



```
int main()
{
  // input of n
  std::cout << “Compute prime numbers in [2, n) for n = ?”;
  unsigned int n;
  std::cin >> n;

  // definition and initialization: provides us with
  // Booleans crossed_out[0],..., crossed_out[n-1]
  array crossed_out (n); // constructor call
  ...
  ...
  delete[] crossed_out;
  return 0;
}
```

Neu

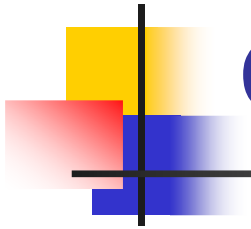
Eine Klasse für Felder: Sieb des Eratosthenes



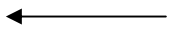
```
int main()
{
  // input of n
  std::cout << “Compute prime numbers in [2, n) for n = ?”;
  unsigned int n;
  std::cin >> n;

  // definition and initialization: provides us with
  // Booleans crossed_out[0],..., crossed_out[n-1]
  array crossed_out (n); // constructor call
  ...
  ...
  delete[] crossed_out;
  return 0;
}
```

Fehler: **`crossed_out`** ist kein Standard-Feld, **`delete`** ist nicht verfügbar!

Eine Klasse für Felder: Sieb des Eratosthenes



```
int main()
{
  // input of n
  std::cout << “Compute prime numbers in [2, n) for n = ?”;
  unsigned int n;
  std::cin >> n;

  // definition and initialization: provides us with
  // Booleans crossed_out[0],..., crossed_out[n-1]
  array crossed_out (n); // constructor call
  ...
  ...

  return 0;
}
```



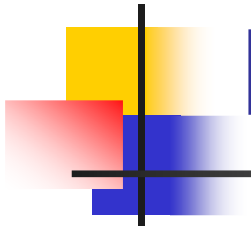
`delete` weglassen geht, aber dann entsteht ein Speicherleck

Eine Klasse für Felder: Sieb des Eratosthenes

```
int main()
{
  // input of n
  std::cout << “Compute prime numbers in [2, n) for n = ?”;
  unsigned int n;
  std::cin >> n;

  // definition and initialization: provides us with
  // Booleans crossed_out[0],..., crossed_out[n-1]
  array crossed_out (n); // constructor call
  ...
  ...

  return 0;
}
```


Diese Variable hat *automatische* Speicherdauer; dynamischer Speicher sollte mit der Variable automatisch “verschwinden”!

Eine Klasse für Felder: Der Destruktor



- spezielle Mitglieds-Funktion (*Anti-Kon-struktor*), die automatisch für jedes Klassen-Objekt aufgerufen wird, dessen Speicherdauer endet

Eine Klasse für Felder: Der Destruktor

- spezielle Mitglieds-Funktion (*Anti-Kon-struktor*), die automatisch für jedes Klassen-Objekt aufgerufen wird, dessen Speicherdauer endet

```
// --- Destructor -----------------------------------
~array ()
{
  delete[] ptr;
}
```

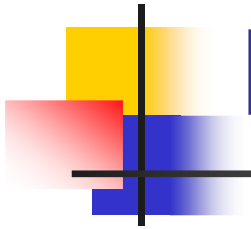
Destruktorname: ~*T* (keine Parameter)

Eine Klasse für Felder: Sieb des Eratosthenes

```
int main()
{
  // input of n
  std::cout << “Compute prime numbers in [2, n) for n = ?”;
  unsigned int n;
  std::cin >> n;

  // definition and initialization: provides us with
  // Booleans crossed_out[0],..., crossed_out[n-1]
  array crossed_out (n); // constructor call
  ...
  ...

  return 0;
}
```

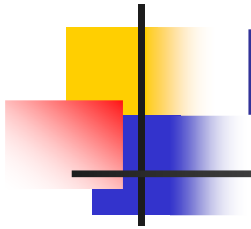
Hier wird der Destruktor für die Klassenvariable **`crossed_out`** automatisch aufgerufen: das **`delete[]`** passiert “von selbst”!

Eine Klasse für Felder: Das Kopieren



- Spezieller Kopier-Konstruktor, der automatisch aufgerufen wird, wenn ein Wert der Klasse mit einem anderen Wert der Klasse initialisiert wird

Eine Klasse für Felder: Das Kopieren

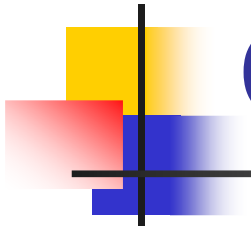


- Spezieller Kopier-Konstruktor, der automatisch aufgerufen wird, wenn ein Wert der Klasse mit einem anderen Wert der Klasse initialisiert wird

```
array a = b;
```

vom Typ array

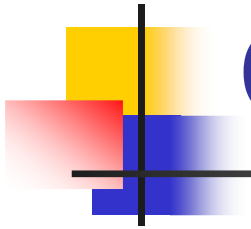
Eine Klasse für Felder: Der Copy-Konstruktor



- ...der Klasse *T* ist der eindeutige Konstruktor mit Deklaration

```
T(const T& x);
```

Eine Klasse für Felder: Der Copy-Konstruktor

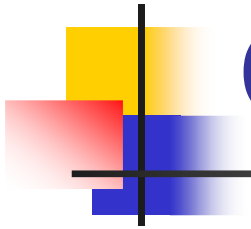


- ...der Klasse *T* ist der eindeutige Konstruktor mit Deklaration

  T **`(const`** *T* **`& x);`**

- wird automatisch aufgerufen, wenn Werte vom Typ *T* mit Werten vom Typ *T initialisiert* werden

Eine Klasse für Felder: Der Copy-Konstruktor

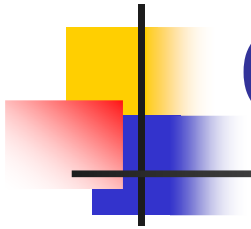


- ...der Klasse *T* ist der eindeutige Konstruktor mit Deklaration

  ```
  T(const T& x);
  ```

- wird automatisch aufgerufen, wenn Werte vom Typ *T* mit Werten vom Typ *T initialisiert* werden
  - `T x = t; // t vom Typ T`

Eine Klasse für Felder: Der Copy-Konstruktor

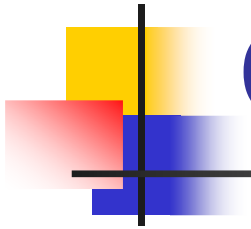


- ...der Klasse *T* ist der eindeutige Konstruktor mit Deklaration

  T`(const T& x);`

- wird automatisch aufgerufen, wenn Werte vom Typ *T* mit Werten vom Typ *T initialisiert* werden
  - `T x (t); // t vom Typ T`

Eine Klasse für Felder: Der Copy-Konstruktor

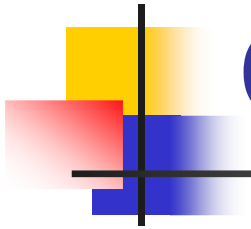


- ...der Klasse *T* ist der eindeutige Konstruktor mit Deklaration

  T **`(const`** *T* **`& x);`**

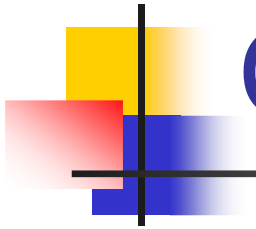
- wird automatisch aufgerufen, wenn Werte vom Typ *T* mit Werten vom Typ *T initialisiert* werden
  - Initialisierung von formalen Funktions-parametern und Rückgabewerten

Eine Klasse für Felder: Der Copy-Konstruktor



```
// --- Copy Constructor --------------------------------
// POST: *this is initialized from a
array (const array& a)
  : ptr (new T[a.size]), size (a.size)
{
  // now copy the elements of a into *this
  for (int i = 0; i < size; ++i)
    ptr[i] = a[i];
}
```

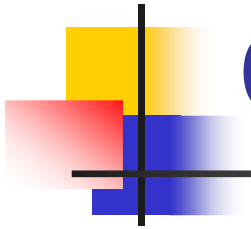
Eine Klasse für Felder: Der Copy-Konstruktor



```
// --- Copy Constructor -------------------------------
// POST: *this is initialized from a
array (const array& a)
  : ptr (new T[a.size]), size (a.size)
{
  // now copy the elements of a into *this
  for (int i = 0; i < size; ++i)
    ptr[i] = a[i];
}
```

Warum ist **`const array& a`** hier zwingend notwendig, während **`array a`** nicht geht ?

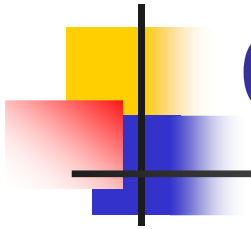
Eine Klasse für Felder: Der Copy-Konstruktor



```
// --- Copy Constructor -------------------------------
// POST: *this is initialized from a
array (const array& a)
  : ptr (new T[a.size]), size (a.size)
{
  // now copy the elements of a into *this
  for (int i = 0; i < size; ++i)
    ptr[i] = a[i];
}
```

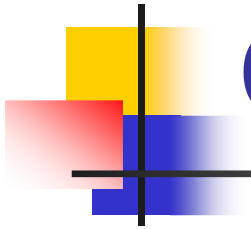
Aufruf eines Copy-Konstruktors mit Deklaration `array (array a);` müsste den Copy-Konstruktor aufrufen (Initialisierung des formalen Parameters!); unendliche Rekursion!

Eine Klasse für Felder: Der Copy-Konstruktor



- Falls kein Copy-Konstruktor deklariert ist, so wird er automatisch erzeugt (und initialisiert mitgliedsweise)

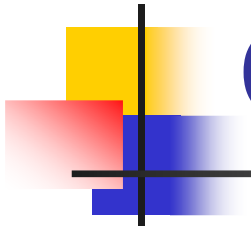
Eine Klasse für Felder: Der Copy-Konstruktor



- Falls kein Copy-Konstruktor deklariert ist, so wird er automatisch erzeugt (und initialisiert mitgliedsweise)

Das ist für unsere Klasse **`array`** nicht das, was wir wollen (es wird nur der Zeiger **`ptr`** kopiert, *nicht* jedoch das dahinterliegende Feld – wir erhalten Alias-Semantik)!

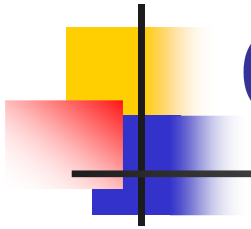
Eine Klasse für Felder: Der Copy-Konstruktor



Nun können wir zum Beispiel schreiben:

```
array a(3);
a[0] = true;
a[1] = false;
a[2] = false; // a == {true, false, false}


array b(a);   // b == {true, false, false}
b[2] = true;  // b == {true, false, true}
```

Eine Klasse für Felder: Der Copy-Konstruktor



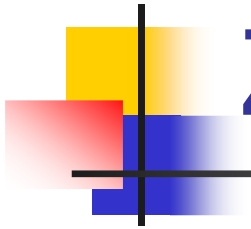
Nun können wir zum Beispiel schreiben:

```
array a(3);
a[0] = true;
a[1] = false;
a[2] = false; // a == {true, false, false}
```

Initialisierung von b durch a

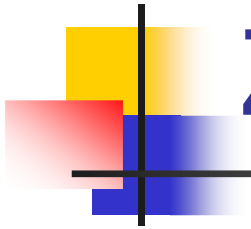
```
array b(a);   // b == {true, false, false}
b[2] = true;  // b == {true, false, true}
```

Eine Klasse für Felder: Der Zuweisungsoperator



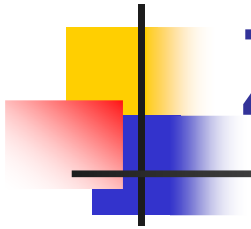
- Überladung von **`operator=`** als Mitglieds-Funktion

Eine Klasse für Felder: Der Zuweisungsoperator



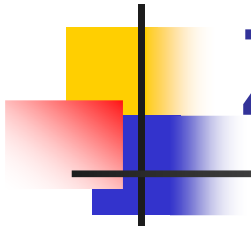
- Überladung von **`operator=`** als Mitglieds-Funktion
- ähnlich wie Copy-Konstruktor ohne Initialisierer, aber zusätzlich
  - Freigabe des Speichers für den “alten” Wert
  - Vermeidung von Selbstzuweisungen

Eine Klasse für Felder: Der Zuweisungsoperator



```
array& operator= (const array& a)
{
  // avoid self-assignments
  if (this != &a) {
    // free old memory, and get new memory of
    // appropriate size
    delete[] ptr;
    ptr = new T[size = a.size];
    // copy the elements of a into *this
    for (int i = 0; i < size; ++i)
       ptr[i] = a[i];
  }
  return *this;
}
```

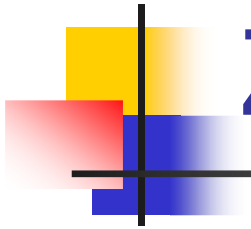
Eine Klasse für Felder: Der Zuweisungsoperator



```
array& operator= (const array& a)
{
  // avoid self-assignments
  if (this != &a) {
    // free old memory, and get new memory of
    // appropriate size
    delete[] ptr;
    ptr = new T[size = a.size];
    // copy the elements of a into *this
    for (int i = 0; i < size; ++i)
       ptr[i] = a[i];
  }
  return *this;
}
```

Test auf Selbstzuweisung ist wichtig, um versehentliches Löschen des zuzuweisenden Wertes zu vermeiden.

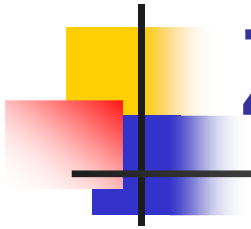
Eine Klasse für Felder: Der Zuweisungsoperator



- Falls kein Zuweisungsoperator deklariert ist, so wird er automatisch erzeugt (und weist mitgliedsweise zu)

Das ist für unsere Klasse **`array`** nicht das, was wir wollen (es wird nur der Zeiger **`ptr`** kopiert, *nicht* jedoch das dahinterliegende Feld – wir erhalten Alias-Semantik)!

Eine Klasse für Felder: Der Zuweisungsoperator

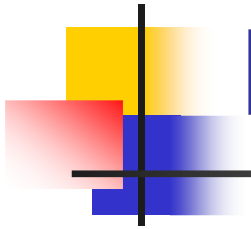


Nun können wir zum Beispiel schreiben:

```
array a(3);
a[0] = true;
a[1] = false;
a[2] = false; // a == {true, false, false}
```

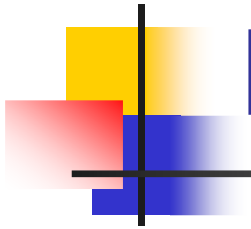
Zuweisung von b an a

```
array b(2);
b = a;        // b == {true, false, false}
b[2] = true;  // b == {true, false, true}
```



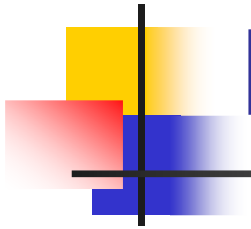
Dynamischer Datentyp

- Typ, der dynamischen Speicher verwaltet (z.B. unsere Klasse für Felder)



Dynamischer Datentyp

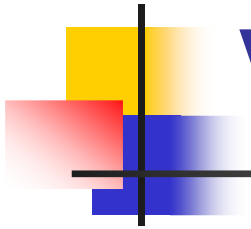
- Typ, der dynamischen Speicher verwaltet (z.B. unsere Klasse für Felder)
- andere typische Anwendungen:
  - Listen
  - Stapel
  - Bäume
  - Graphen



Dynamischer Datentyp

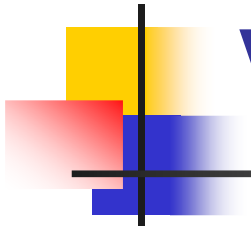
- sollte immer mindestens
  - Konstruktoren
  - Destruktor
  - Copy-Konstruktor
  - Zuweisungsoperator

  haben.



Vektoren

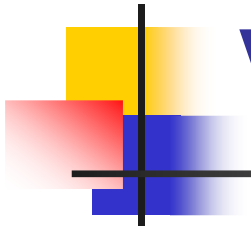
- sind die Felder der Standardbibliothek



Vektoren

- sind die Felder der Standardbibliothek
- **`std::vector<`*`T`*`>`** für zugrundeliegenden Typ *T*

```
#include<vector>
```

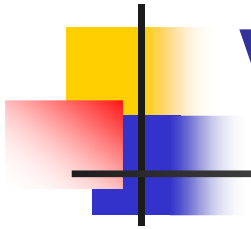


Vektoren

- sind die Felder der Standardbibliothek
- **`std::vector<`*T*`>`** für zugrundeliegenden Typ *T*

```
#include<vector>
```

- können noch mehr, z.B:
  - Elemente am Ende anhängen und löschen

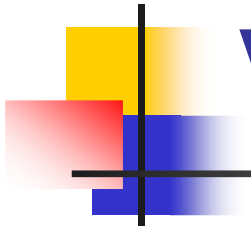


Vektoren

- sind die Felder der Standardbibliothek
- **`std::vector<`*T*`>`** für zugrundeliegenden Typ *T*
- können noch mehr, z.B:
  - Elemente am Ende anhängen und löschen

```
#include<vector>
```

können wir für unsere Klasse array auch machen (siehe Challenge 65)



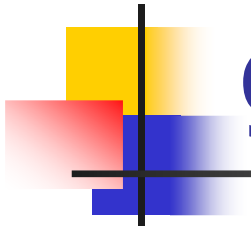
Vektoren

- sind die Felder der Standardbibliothek
- **`std::vector<`***T*`>`** für zugrundeliegenden Typ *T*

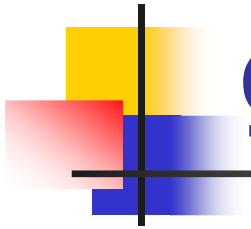
```
#include<vector>
```

- können noch mehr, z.B:
  - Elemente am Ende anhängen und löschen
- repräsentieren damit die Menge T^* aller endlichen Folgen über *T*

Das war’s! Was wir nicht gemacht haben:

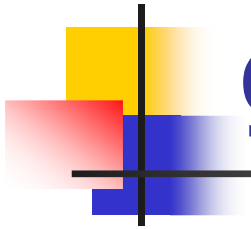


Das war's! Was wir nicht gemacht haben:



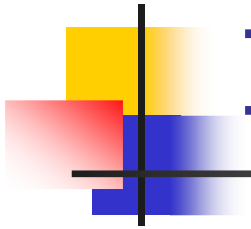
- Vererbung (Klassenhierarchien)

Das war's! Was wir nicht gemacht haben:



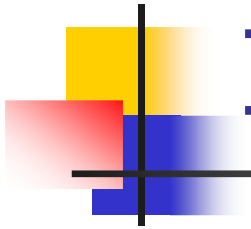
- Vererbung (Klassenhierarchien)
- Templates (Generisches Programmieren) – es gibt aber eine eigene Vorlesung dazu

Wie es weitergeht mit der Informatik



- Im zweiten Bachelor-Jahr (Mathematik):
  - VL Algorithmen und Komplexität: Theorie der Algorithmen; wichtige Konzepte der Programmierung (Funktionen, Felder, Zeiger) werden vorausgesetzt

Wie es weitergeht mit der Informatik



- Im zweiten Bachelor-Jahr (Mathematik):
  - VL Algorithmen und Komplexität: Theorie der Algorithmen; wichtige Konzepte der Programmierung (Funktionen, Felder, Zeiger) werden vorausgesetzt
- Für Physiker:
  - leider kommt hier erstmal nichts mehr...