UMIT
the health & life sciences university

# Vorlesung 2P: Dynamische Speicherverwaltung, Konstruktoren, Destruktoren, Operatoren, Vererbung

- Referenzen
- Dynamische Speicherverwaltung in C++
- Konstruktoren und Destruktoren
- Überladen von Operatoren
- Copy-Konstruktoren

UMIT
the health & life sciences university

## Call by Reference

*(Wiederholung)*

- Auch bei der Übergabe als Referenzparameter wird nicht der *Wert* der Parametervariablen, sondern ihre *Speicheradresse* übergeben.
- Analog zur Übergabe mittels Zeiger ist es auch hier nicht möglich, Literale (wie `3`) oder zusammengesetzte Ausdrücke (wie `a+b`) als Referenzparameter zu übergeben.
- **Vorteil gegenüber Zeigern:** Der Adressoperator beim Aufruf der Funktion entfällt. Damit muss der/die Programmierer/in beim Aufruf der Funktion sich in den meisten Fällen keine Gedanken darüber machen, ob die einzelnen Parameter als Werte oder Referenzen übergeben werden!
- Im Unterschied zu Zeigern müssen Referenzen aber stets auf gültige Variablen verweisen, es gibt keine „Null-Referenz“!
- Referenzparameter können auch *read-only* übergeben werden:

```
void print (const string &s);
```

UMIT
the health & life sciences university

## Referenzen

- Eine **Referenz** ist ein Aliasname für eine bestehende Variable.
- Referenzen können auch unabhängig von Funktionsparametern benutzt werden.
- Durch eine Referenz wird kein neuer Speicherplatz reserviert, sondern der vorhandene Speicherplatz benutzt.
- Beispiel:

```
double x; double &ref = x;
```

- Referenzen auf Konstanten müssen selbst durch das Schlüsselwort `const` als konstant deklariert werden.

```
const double x; const double &ref = x;
```

- Referenzen auf Variable können ebenfalls als konstant deklariert werden.

```
double x; const double &ref = x;
```

(Damit kann man `x` selbst lesend und schreibend benutzen, `ref` aber nur lesend!)

## Der Operator `new`

- Mit dem Operator `new` können in C++ Instanzen beliebiger Objektklassen oder elementarer Datentypen zur Laufzeit angelegt werden:

```
double *pd = new double; // alloziert ein double
Image  *pi = new Image;  // alloziert ein Bild
                         // ruft Default-Konstruktor Image() auf
```

- Durch das Schlüsselwort `new` geschieht zur Laufzeit Folgendes:
  - Es wird ein für den angegebenen Typ passender Speicherbereich reserviert.
  - Die Anfangsadresse des reservierten Speicherbereichs wird als Rückgabewert von `new` zurückgegeben (und sinnvollerweise einer Zeigervariablen zugewiesen).
  - Sofern der angegebene Typ über einen Konstruktor verfügt (wenn es sich also um eine Objektklasse handelt), wird dieser aufgerufen.

## Der Operator `new`

- Es können auch Initialisierungswerte übergeben werden:

```
double *pd1 = new double;           // keine Initialisierung
double *pd2 = new double (123.45); // mit Anfangswert
```

- Wird ein *Objekt* mittels `new` erzeugt, so wird nicht nur der benötigte Speicher erzeugt, sondern auch der passende Konstruktor aufgerufen:

```
class Image { ... };

Image* pi1 = new Image;
    // Aufruf des Default-Konstruktors Image()
Image* pi2 = new Image (30, 40); // mit Initialisierungswerten,
    // Konstruktor Image (const int, const int) wird aufgerufen
    // (ersatzweise Konstruktor mit anderen Parametern, in die
    // implizit umgewandelt werden kann)
```

UMIT
the health & life sciences university

## Der Operator `new`

- Ein Array mit mehreren Instanzen (Variablen) kann mittels `[]` alloziert werden:

```
double *pd3 = new double [nx * ny];
    // Feld fuer nx * ny double-Werte, ohne Initialisierung
Image  *pi3 = new Image [5];     // Bildserie aus 5 Bildern
                                 // Default-Konstruktor Image() wird 5mal
                                 // aufgerufen (fuer jede Image-Instanz)
```

- Mit der bisher verwendeten Syntax (Konstruktorargumente in runden Klammern) kann bei Array-Allokation (`new[]`) keine Initialisierung erfolgen. nur der Default-Konstruktor verwendet werden bzw. bei elementaren Datentypen keine Initialisierung vorgenommen werden.

  Mit den in C++11 eingeführten **Initialisiererlisten** können auch hier Initialisierungswerte übergeben werden, etwa

```
double *pd4 = new double [4] { 1.1, 2.2, 3.3, 4.4 };
Image  *pi4 = new Image [3] { { 8, 8 }, { 16, 12 }, { 12, 16 } };
```

## Der Operator `new`

- Auf die Variablen wird anschließend mittels des Zeigers zugegriffen, also beispielsweise

```
double *pd4 = new double [4] { 1.1, 2.2, 3.3, 4.4 };
Image  *pi4 = new Image [3] { { 8, 8 }, { 16, 12 }, { 12, 16 } };
...
cout << *pd4 << endl;                  // -> 1.1
cout << pd4[2] << endl;                // Zeigerarithmetik! -> 3.3
cout << pi4->pixel (1, 2) << endl; // Pfeiloperator+Methode
cout << pi4[1].pixel (2, 3) << endl; // Zeigerarithmetik!
```

## Der Operator `delete`

- Für mit Zeigern verwalteten Speicher in C++ gibt es keinen Garbage Collector.

  Die Lebenszeit von Variablen, die mit `new` zur Laufzeit erzeugt wurden, wird also nicht automatisch kontrolliert. Entwickelnde müssen selbst die Freigabe nicht mehr benötigten Speichers steuern.

- Im einfachsten Fall (schlecht!) verlässt man sich darauf, dass das Betriebssystem am Programmende die Reste wegräumt. Dann bleiben alle durch `new` erzeugten Speicherbereiche so lange blockiert, auch wenn dafür keine Notwendigkeit besteht.

- Besser ist es, den Speicherbereich dann frei zu geben, wenn er nicht mehr gebraucht wird – ein Muss für saubere Programmentwicklung!

- Dies geschieht durch den Operator `delete`, z. B.

  delete ruft destructor auf?

  ```
  delete ptr;
  ```

- Ein Speicherbereich, der mittels `new[]` alloziert wurde, muss mit

  ```
  delete[] ptr;
  ```

  freigegeben werden.

UMIT
the health & life sciences university

## Verschachtelte Objekte

- Oftmals werden Objekte als Datenelemente in anderen Objekten gebraucht
- Hierbei werden die Konstruktoraufrufe verkettet abgearbeitet *(chaining)*.
- *Beispiel:* Einträge für einen Geburtstagskalender könnten als Objekte einer Klasse wie folgt angelegt werden:

```
class Birthday { string name; CalendarDate date; ... }
```
*mit passenden Konstruktoren und Methoden.*

Instanziierung von `Birthday` durch `Birthday x;`

Erzeuge ein Objekt vom Typ `Birthday`
  Erzeuge Datenfeld `name`
    Konstruktoraufruf `string::string()` zur Initialisierung
  Erzeuge Datenfeld `date`
    Konstruktoraufruf `CalendarDate::CalendarDate()` zur Initialisierung
  Konstruktoraufruf `Birthday::Birthday ()`

UMIT
the health & life sciences university

## Verschachtelte Objekte

Die implizite Initialisierung beim Chaining kann sehr ineffizient sein, insbesondere wenn statt eines spezifischen Konstruktors der Default-Konstruktor aufgerufen und anschließend die Werte überschrieben werden.

- *Beispiel:* Zwar kann bei der Erzeugung einer Instanz von `Birthday` ein spezifischer Konstruktor `Birthday::Birthday (`*Initialwerte*`)` aufgerufen werden, aber dabei werden `name` und `date` mit den Default-Konstruktoren erzeugt.

- Wenn anschließend die Felder durch Zuweisungen mit spezifischen Werten, z. B. aus den Konstruktorargumenten von `Birthday::Birthday (...)`, gefüllt werden, so werden u. U. aufwändige Aktionen für die Teilobjekte doppelt ausgeführt.

UMIT
the health & life sciences university

## Verschachtelte Objekte

- Instanziierung von `Birthday` durch
  `Birthday x (einName, einTag, einMonat, einJahr);`

Erzeuge ein Objekt vom Typ `Birthday`
  Erzeuge Datenfeld `name`
    Konstruktoraufruf `string::string()` zur Initialisierung
  Erzeuge Datenfeld `date`
    Konstruktoraufruf `CalendarDate::CalendarDate()` zur Initialisierung
  Konstruktoraufruf `Birthday::Birthday (einName, einTag, einMonat, einJahr)`
    – trägt Werte in Datenfelder von `name` und `date` ein, zB mit

```
name = einName;
date.set(einTag,einMonat,einJahr);
```

UMIT
the health & life sciences university

## Elementinitialisierer

- Um dies zu vermeiden, gibt es die Möglichkeit, Konstruktoren um **Elementinitialisierer** zu erweitern:

```
Birthday::Birthday (string n, unsigned int d,
                    unsigned int m, unsigned int y)
    : name (n), date (d, m, y)
  { }
```

- Die Teilobjekte werden dann direkt mit den Werten initialisiert, ohne dass der Defaultkonstruktor aufgerufen wird. Dazu werden beim Chaining die entsprechenden qualifizierten Konstruktoren der Teilobjekte aufgerufen.
- Konstruktoren, deren einziger Zweck das Initialisieren von Datenfeldern ist, können regelhaft ausschließlich mit Elementinitialisierern und mit leerem Funktionskörper `{ }` realisiert werden.

UMIT
the health & life sciences university

## Elementinitialisierer

- Instanziierung von `Birthday` durch
  `Birthday x (einName, einTag, einMonat, einJahr);`

Erzeuge ein Objekt vom Typ `Birthday`
  Erzeuge Datenfeld `name`
    Konstruktoraufruf `string::string (einName)`
  Erzeuge Datenfeld `date`
    Konstruktoraufruf `CalendarDate::CalendarDate (einTag, einMonat, einJahr)`
  Konstruktoraufruf `Birthday::Birthday (einName, einTag, einMonat, einJahr)`

## Destruktoren

- Wenn ein Objekt Speicherbereiche über Zeiger verwaltet – die dann normalerweise durch Konstruktoren mittels `new` alloziert werden –, so muss dafür Sorge getragen werden, dass diese Speicherbereiche am Ende der Lebenszeit des Objektes auch wieder mittels `delete` freigegeben werden.

- Dafür benötigt man einen **Destruktor.**

  So wie ein Konstruktor dafür da ist, bei Erzeugung eines Objektes nötige Initialisierungen vorzunehmen, ist ein Destruktor dazu da, alles Erforderliche für eine ordnungsgemäße „Entsorgung“ des Objektes zu veranlassen.

- Der Name eines Destruktors ist stets `~`*Klassenname*, also etwa

  ```
  Image::~Image(){...}
  ```

  (quasi ein mit `~` negierter Konstruktor)

- Ein Destruktor hat keinen Rückgabewert und keine Parameter.

UMIT
the health & life sciences university

## Destruktoren

- Es kann pro Klasse nur einen Destruktor geben.

  Ist kein Destruktor programmiert, wird ein Default-Konstruktor vom Compiler erzeugt (der im Wesentlichen nichts tut).

- Der Destruktor wird automatisch aufgerufen, wenn die Existenz eines Objekts endet, i. d. R. am Ende eines Blocks oder des Programms.

- So wie Konstruktoren werden Destruktoren im Normalfall nicht explizit aufgerufen.

  *(Ein expliziter Aufruf des Destruktors ist möglich, aber nur in speziellen Situationen, die uns hier nicht beschäftigen.)*

## Der Operator `delete`

- Bei der Freigabe eines Objektes mit `delete` wird dessen Destruktor aufgerufen:

```
double *pd1 = new double;
Image  *pi1 = new Image;
...
delete pd1;    // elementarer Datentyp, kein Destruktor
delete pi1;    // ruft ~Image() auf
```

- Bei der Freigabe eines Arrays von Objekten mit `delete[]` wird der Destruktor für jede Instanz einmal aufgerufen:

```
double *pd3 = new double [nx * ny];
Image  *pi3 = new Image [5];
...
delete [] pd3;  // elementarer Datentyp, kein Destruktor
delete [] pi3;  // ruft ~Image() 5mal auf
```

Mit `delete pi3;` würde zwar der Speicher freigegeben, aber der Destruktor nur für das erste `Image`-Objekt aufgerufen!

## Anwendung von `delete`

- Wird eine Klasseninstanz mit `delete` freigegeben, so wird zuerst der Destruktor aufgerufen und dann der Speicherplatz freigegeben.
- Weiteres siehe Literatur, z. B. Beispielprogramm `DynObj.cpp` in *Kirch-Prinz/Prinz*.
- **Wichtig:** `delete` darf nur auf Zeiger angewendet werden, die zuvor mittels `new` erzeugt wurden!
- Für jede Speicheranforderung darf `delete` nur einmal aufgerufen werden.
- So wie `new` bei Allokation von Objektinstanzen automatisch den Konstruktor für jede Instanz aufruft, stellt `delete` sicher, dass pro Instanz einmal der Destruktor aufgerufen wird. Hierfür ist die Unterscheidung `delete` / `delete[]` allerdings wesentlich.
- Der Rückgabetyp von `delete` ist `void`, man kann also nicht testen, ob die Freigabe erfolgreich war.
- Es ist erlaubt, `delete` mit einem Null-Zeiger aufzurufen. Damit kann es auch für nicht initialisierte Objekte angewendet werden.
- Weiteres siehe Literatur (Hausübung), siehe auch Beispielprogramm `DynStd.cpp` in *Kirch-Prinz/Prinz*.

## Fehlerbehandlung

- Die korrekte Freigabe von Speicherplatz durch `delete` kann nicht überprüft werden.
- Beim Reservieren von Speicher mittels `new` ist es wichtig, zumindest einen Fehler kontrollieren und behandeln zu können: das Überschreiten des tatsächlich auf dem Rechner verfügbaren Speicherplatzes.
  - In diesem Falle wird eine so genannte *Exception* ausgelöst.
  - Standardmäßig führt die Exception zum Abbruch des Programms.
  - Es ist jedoch möglich, mittels Fehlerbehandlungsmethoden in die Fehlerbehandlung einzugreifen und den Fehler so „abzufangen“, dass das Programm weiterlaufen kann *(Vorlesung 11)*.

## `new/delete` versus `malloc/free`

- Die Allokation und Freigabe mit `malloc` und `free` ist in C und C++ möglich. Dabei „weiß“ `malloc` von dem Datentyp, für den der reservierte Speicher benutzt wird, nichts (und gibt deswegen auch den Typ `void*` zurück).
- *Zur Erinnerung:* In reinem C kann die explizite Umwandlung des Rückgabewertes von `malloc` entfallen; es erfolgt dann eine implizite Umwandlung. In C++ ist aber eine implizite Umwandlung von `void*` in andere Zeigertypen verboten.
- Die C++-spezifische Art, Speicher zu reservieren und freizugeben, ist durch die Operatoren `new` und `delete` gegeben.
- Für die Allokation von Feldern einfacher Datentypen haben `malloc` und `new` praktisch gleiche Wirkung.
- Für Objekte gibt es jedoch einen entscheidenden Unterschied: Nur `new` kennt den Typ der zu erzeugenden Objekte; damit kann nur `new` den passenden Konstruktur aufrufen.

  Analog kann nur `delete` vor der Freigabe des Speicherplatzes den Destruktor eines Objekts aufrufen.

## `new/delete` versus `malloc/free`

- Mit `malloc` allozierter Speicher *muss* mit `free` freigegeben werden. Mit `new` allozierter Speicher *muss* mit `delete` freigegeben werden.

  Beide Verfahren „über Kreuz“ zu kombinieren ist nicht möglich. Generell ist Mischen der beiden Verfahren im selben Programm unschön. Somit gehört `malloc`/`free` grundsätzlich ins C-Umfeld.

- *Zur Erinnerung:* In C/C++ sollte allozierter Speicher stets freigegeben werden; es gibt (anders als z. B. in Java) keine *Garbage Collection*.

  *Empfehlung:* Beim Design des Programms sollte darauf geachtet werden, dass (auf einer geeigneten Ebene des Programms) die Allokation und Freigabe „gepaart“ sind, das heißt zu jeder Allokation die zugehörige Freigabe klar zuzuordnen ist.

## Der Zeiger `this`

- Innerhalb einer Klassenmethode steht der Zeiger `this` bereit, der auf die aktuelle Objektinstanz verweist.
- Damit kann gezielt ein Element des aktuellen Objektes angesprochen werden.
- Dies hat z. B. Sinn, wenn Objektelemente und Methodenparameter gleiche Namen haben:

```
Image::Image (const int nx, const int ny)
{
  this->nx = nx;
  this->ny = ny;
  ...
}
```

(im Beispiel wurden stattdessen Elementinitialisierer verwendet)

## Dynamische Klasseninstanzen und Konstruktoren/Destruktoren

- *Erinnerung:* Wenn eine Klasse als Datenelemente wieder Objekte enthält wie

```
class Birthday { string name; CalendarDate date; ... }
```

(vgl. Vorlesung 9, Folie 28), dann wird bei *dynamischer* Instanziierung

```
Birthday *pb = new Birthday;
```

der Konstruktor von `Birthday` aufgerufen, beim `delete` dann der Destruktor.

Per Chaining werden dann auch die Konstruktoren/Destruktoren von `string` und `CalendarDate` aufgerufen.

UMIT
the health & life sciences university

## Dynamische Klasseninstanzen und Konstruktoren/Destruktoren

- Dank dynamischer Speicherreservierung kann man aber als Datenelemente auch *Zeiger* auf Objekte verwenden:

```
class Birthday { string *pname; CalendarDate *pdate; ... }
```

- In diesem Fall beschafft `pb = new Birthday` nur den Speicherplatz für das `Birthday`-Objekt (einschließlich der beiden Zeiger) und ruft den Konstruktor auf.

  Da keine `string`- und `CalendarDate`-Objekte automatisch erzeugt werden, erfolgt auch *kein Chaining.*

  Sinnvollerweise wird der Konstruktor von `Birthday` seinerseits durch

```
pname = new string;
pdate = new CalendarDate;
```

  die Datenobjekte anlegen – dies ist eine typische Aufgabe für einen Konstruktor. Dadurch werden dann auch die Konstruktoren für `string` und `CalendarDate` aufgerufen.

## Dynamische Klasseninstanzen und Konstruktoren/Destruktoren

- Beim `delete` gilt das Gleiche: `delete pb;` kümmert sich *nicht* um die Datenobjekte und ruft nur den Destruktor von `Birthday` auf.

  Deswegen muss `Birthday::~Birthday()` in diesem Fall durch

  ```
  delete pname;
  delete pdate;
  ```

  die Datenobjekte wegräumen, sonst bleibt ihr Speicher reserviert und unerreichbar!

  Dies ist eine typische Aufgabe des Destruktors!

UMIT
the health & life sciences university

## Beispiel: Quellcode verstehen

→2106-images-class *(aus Block 1)*

- Wir betrachten im Übungsbeispiel 2106-images-class nochmals die Art und Weise, wie das Rotieren von Bildern gelöst wurde:

  *Funktion:*

```
void rotateimage (Image &src, Image &dest) { ... }
```

  *Aufruf in* `main ()`*:*

```
Image rotatedimage (myimage.sizey(), myimage.sizex());
rotateimage (myimage, rotatedimage);
outputimage = &rotatedimage;
```

  Ist diese Lösung gut?

  Wenn nein: Was würde man sich stattdessen wünschen?

UMIT
the health & life sciences university

## Beispiel: Quellcode verstehen

→2106-images-class *(aus Block 1)*

- Zusatzfrage: Warum muss in `main` für den `Image`-Zeiger `outputimage` kein `delete` aufgerufen werden?

  *Vgl. Deklaration in* `main ()`*:*

```
Image *outputimage = &myimage;
```

UMIT
the health & life sciences university

## Beispiel: Quellcode verstehen

→2201-images-class

- Wir betrachten nun das Beispiel 2201-images-class-src.

  (Nur die Datei main.cpp ist gegenüber dem vorigen Beispiel geändert.)

- In der umgeschriebenen rotateimage-Funktion wird Zeigerübergabe benutzt:

  *Funktion:*

```
Image* rotateimage (Image &u) { ...
  Image *v = new Image (ny, nx); ...
  return v;
}
```

  *Aufruf in* main ():

```
Image *qimage = nullptr;
if (...) { qimage = rotateimage (myimage); pimage = qimage; }
if (qimage != nullptr) delete qimage;
```

- Ist diese Lösung zufriedenstellend?

## Zuweisungsoperator für Objekte

- Der einzige Standardoperator, der auf Objekte anwendbar ist, ist die Zuweisung mit `=`.
- Die Zuweisung funktioniert nur, wenn beide Operanden Objekte derselben Klasse sind!
- Mit dem Zuweisungsoperator werden alle Elementwerte des einen Objektes auf das andere Objekt kopiert.
- Zum Beispiel ist, wenn `date1` (vom Typ `CalendarDate`, egal in welcher der beiden Implementationen aus Block 1) den 15.3.2019 als Wert enthält, nach der Zuweisung `date2=date1;` auch in `date2` der 15.3.2019 gespeichert.

```
CalendarDate date1 (Initialisierungswerte);
CalendarDate date2;
date2 = date1;
```

- Dabei werden die Datenelemente von `date1` *binär* in `date2` kopiert.
- Das ist in Ordnung, solange es sich nur um einfache Datenvariablen handelt. Es ist nicht mehr in Ordnung, wenn dynamische Variablen im Spiel sind!

## Beispiel zum Kopieren von Objekten

- In unserer Klasse zur Darstellung von Grauwertbildern

```
class Image {
  double *p;
  ...
}
```

  ist `p` ein Zeiger auf ein Feld von Grauwerten.

- Mit

```
Image bild1 (30, 50);
Image bild2 (40, 70);
bild2 = bild1;
```

  am Ende des Programms: destructor von bild1 gibt speicher frei,, destructor von bild2 will speicher freigeben, dieser ist aber bereits freigegeben

  entsteht eine Kopie von `bild1` in `bild2`, in der *der Zeiger* `p` *binär identisch kopiert ist,* also auf denselben Speicher verweist.

  Änderungen von Pixeln in `bild2` würden dann auch auf `bild1` wirken und umgekehrt.

- Außerdem wird ggf. vorher in `bild2` allozierter Speicher dabei unzugänglich.
- *Das ist nicht das, was man mit* `bild2 = bild1;` *will!*

## Beispiel zum Kopieren von Objekten

- Als Ausweg könnte man eine Methode

  ```
  void Image::copy (const Image& source)
  ```

  definieren, die einen gleich großen Speicherbereich reserviert und den Inhalt des Bildes kopiert.

  Dann kann man schreiben

  ```
  bild2.copy (bild1);
  ```

  und hat nun tatsächlich zwei Exemplare des Bildes.

- **Umwandeln in einen Operator:** Wenn man die Methode `Image::copy` in `Image::operator=` umbenennt, dann führt

  ```
  bild2 = bild1;
  ```

  genau dieselbe Kopieroperation aus!

## Überladen von Operatoren

- Damit unser überladener =-Operator sich ganz wie sein Vorbild verhält, sollte der Rückgabetyp noch geändert werden:

```
Image& Image::operator= (Image& source);
```

  In der Implementation muss dafür als letzte Zeile

```
return (*this);
```

  stehen.

- Außerdem sollte man noch den Fall abfangen, dass das Quellbild und Zielbild identisch sind. Dazu fügt man ganz am Anfang der Methode ein

```
if (this == &source) return (*this);
```

UMIT
the health & life sciences university

## Beispiel: Quellcode verstehen

→2202-images-class

- Eine verbesserte Lösung für die Rotation von Bildern finden wir im Beispiel 2202-images-class.
- Wir sehen uns zunächst die Rotation von Bildern in main.cpp an.

*Funktion:*

```
Image rotateimage (Image &u) {
  ...
  Image v (ny, nx);
  ...
  return v;
}
```

*Aufruf in* `main ()`*:*

```
if (...) { myimage = rotateimage (myimage); }
```

## Beispiel: Quellcode verstehen

→2202-images-class

- Die Statements

```
return v; // in rotateimage (); v ist Image
myimage = rotateimage (myimage); // in main ()
```

  beruhen auf dem Kopieren von `Image`-Objekten mittels `=`.

- Dazu gehört in `image.h|cpp` die Methode `Image::operator=`.

```
Image& Image::operator= (const Image& source) {
  if (this == &source) return *this;
  if (p != nullptr) delete [] p;
  p = nullptr;
  nx = source.nx;
  ny = source.ny;
  p = new double [nx*ny];
  for (int i=0; i<nx*ny; ++i)
    p[i] = source.p[i];
  return *this;
}
```

UMIT
the health & life sciences university

## Überladen von Operatoren

*Warum funktioniert das?*

- Operatoren in C++ werden intern ebenfalls wie Funktionen oder Methoden behandelt, etwa

  | | | |
  |---|---|---|
  | `=` | entspricht | `operator= ()`, |
  | `+` | entspricht | `operator+ ()`, |
  | `[...]` | entspricht | `operator[] ()`, |
  | `(...)` | entspricht | `operator() ()`. |

- Für elementare Datentypen ist die Bedeutung der Operatoren fest vorgegeben.
- Für Klassentypen sind die Operatoren in der Regel nicht definiert, die Verwendung eines Objektes in einem Ausdruck wie `bild1 + 100` wird also zu einem Fehler führen, genau wie der Aufruf einer Funktion mit unpassenden Datentypen.
- Da Operatoren aber ebenfalls Funktionen sind, kann man sie für andere Datentypen überladen. Genau dies haben wir für `operator= ()` getan.
- Der Wertzuweisungsoperator ist der einzige Operator, der für Objekte auch ohne explizite Definition bereitgestellt wird, ähnlich wie ein Default-Konstruktor. Die eigene Definition überschreibt den Default-Wertzuweisungsoperator.

## Überladen von Operatoren

- *Wir kennen schon überladene Operatoren!*
  - Die Binär-Shift-Operatoren `<<` und `>>` sind für die Stream-Klassen überladen, sodass sie Ein- und Ausgabefunktionen wahrnehmen.
  - Auch Strings sind eine Klasse. Dafür kennen wir die überladenen Operatoren `=`, `+`, `+=`, `<`, `>`, `<=`, `>=`, `==` und `!=`.
- Unveränderlich sind:
  - Die Bedeutung der Operatoren für elementare Datentypen.
    → Deswegen funktioniert `string s = "Grau" + "kaese";` nicht.
  - Die Prioritätsfolge der Operatoren.
    → Deswegen funktioniert z. B. `cout << (a < b);` nur mit Klammern, weil Binärshifts höhere Priorität als Vergleiche haben.
- Einige Operatoren können nicht überladen werden (`?:` `.` `::` `.*`)

## Überladen von Operatoren mit Funktionen

- Bisher haben wir zum Überladen von Operatoren Methoden der Klasse des ersten Operanden benutzt. Manchmal ist das nicht zweckmäßig.
  - Auf diese Weise lassen sich keine Operatoren überladen, bei denen der erste Operand ein elementarer Datentyp ist.
  - Es lassen sich so auch keine Operatoren überladen, bei denen der erste Operand einer Klasse angehört, auf deren Deklaration kein Zugriff besteht.
    Zum Beispiel können wir so die Operatoren `<<` und `>>` nicht für die Ein- und Ausgabe eigener Datentypen aus/in Dateien definieren!
- Die Lösung besteht im **Überladen mit Funktionen.** Statt die Operation `Typ_A $ Typ_B` (wobei `$` für einen Operator steht) mit einer Methode

  ```
  Typ_A::operator$ (Typ_B& x)
  ```

  zu überladen, kann man dies auch mit einer Funktion

  ```
  operator$ (Typ_A& x, Typ_B& y)
  ```

  tun.
- Mittels Funktionen können überladene Operatoren auch unabhängig von der Klassendeklaration des ersten Operanden implementiert werden.

## Kopierkonstruktoren

- Mit der Klasse aus dem vorigen Block funktionierte auch die Initialisierung

  ```
  CalendarDate date2 = date1;
  ```

  ohne dass wir dafür etwas implementieren müssten.

- *Vorsicht:* Hier wirkt nicht der Wertzuweisungsoperator! Beispielsweise würde allein mit dem überladenen =-Operator

  ```
  Image bild2 = bild1;
  ```

  immer noch ein binärkopiertes Bild erzeugen!

- *Grund:* Bei der Initialisierung kommt ein spezieller Konstruktor zum Tragen, der **Kopierkonstruktor (Copy-Konstruktor):**

  ```
  CalendarDate::CalendarDate (const CalendarDate& source);
  Image::Image (const Image& source);
  ```

UMIT
the health & life sciences university

## Kopierkonstruktoren

- Wie der Default-Konstruktor und der Default-Wertzuweisungsoperator ist auch ein Default-Kopierkonstruktor ohne explizite Definition vorhanden.
- Wie der Default-Wertzuweisungsoperator überträgt der Default-Kopierkonstruktor die Feldinhalte binäridentisch.
- Anders als der Default-Konstruktor ist der Default-Kopierkonstruktor auch weiterhin verfügbar, wenn andere Konstruktoren explizit definiert werden. Er kann aber durch einen explizit definierten Kopierkonstruktor überschrieben werden.

  *Faustregel:* Immer wenn man für eine Klasse einen expliziten Wertzuweisungsoperator schreibt, sollte man auch einen Kopierkonstruktor definieren.

  *Beispiel:* Der Kopierkonstruktor der Klasse `Image` sollte analog dem Wertzuweisungsoperator überschrieben werden.

Immer zusammen schreibe: Operator-Überladung, Kopierkonstruktor und Destrukt (+ 2 weitere, nächstes mal mehr)

UMIT
the health & life sciences university

## Beispiel: Quellcode verstehen

→2202-images-class

- Finden Sie den Kopierkonstruktor der `Image`-Klasse und vollziehen Sie nach, wie dieser funktioniert.

  Worin unterscheiden sich die Implementationen von `=`-Operator und Kopierkonstruktor?

UMIT
the health & life sciences university

## Beispiel: Quellcode verstehen

→2202-images-class

- Wieso kann jetzt eigentlich statt `u.pixel (i, j)` etc. im Programm `u (i, j)` usw. geschrieben werden?

## Übungsfrage

Angenommen, `Image` hat einen Konstruktor, der ein Bild aus einer Datei einlesen kann, und eine Methode `writepgm ()`, die ein Bild in eine Datei schreibt.

Weiters angenommen, es wurde **kein** Kopierkonstruktor explizit definiert.

1. Was tut der folgende Code:

```
int main() {
  Image bild1 ("eingabe.pgm");
  Image bild2 = bild1;
  for (int i=0; i<bild2.sizex(); ++i)
    for (int j=0; j<bild2.sizey(); ++j) {
      bild2 (i, j) = 255.0 - bild2 (i, j);
    }
  bild1.writepgm ("ausgabe.pgm");
  return 0;
}
```

2. Wieso stürzt das Programm danach ab?