Speicherverwaltung

- Klassen mit Zeigern als Datenelementen

- Der Zuweisungsoperator `=`
    - Überladen des Zuweisungsoperators `=`

- Der `this`-Zeiger
    - Selbstzerstörung verhindern mit `this`
    - Assoziativität von Operatoren

- Der Kopierkonstruktor
    - Ähnlichkeiten zwischen Kopierkonstruktor und überladenem Zuweisungsoperator

Speicherverwaltung

Klassen mit Zeigern als Datenelementen

```
class Person
{
  public:
    // Konstruktoren und Destruktor
    Person();
    Person(char const *n, char const *a,
           char const *p);
    ~Person();

    // Zugriffsfunktionen
    void setname(char const *n);
    void setaddress(char const *a);
    void setphone(char const *p);

    char const *getname(void) const;
    char const *getaddress(void) const;
    char const *getphone(void) const;

  private:
    // Datenelemente
    char *name;
    char *address;
    char *phone;
};
```

Klassen mit Zeigern als Datenelementen

```
Person::~Person()
{
  delete [] name;
  delete [] address;
  delete [] phone;
}

int main()
{
  Person
    kk("Kurt Kunz", "Frankfurt",
       "069-5426044"),
    *george = new Person("George W. Bush",
                         "White House",
                         "001-202-456-1414");

  cout << kk.getname() << ", "
       << kk.getaddress() << ", "
       << kk.getphone() << ", "
       << endl;

  cout << george->getname() << ", "
       << george->getaddress() << ", "
       << george->getphone() << ", "
       << endl;

  delete george;
}
```

Speicherverwaltung

Der Zuweisungsoperator =

Variablen, die Strukturen oder Objekte einer Klasse sind, können in C++ direkt einander zugewiesen werden.

Die Semantik einer solchen Wertzuweisung ist als byteweise Kopie voreingestellt – wenn alle Attribute einfache Datentypen haben.

Beispiel:

```
void printperson(Person const &p)
{
  Person tmp;

  tmp = p;

  cout << "Name:    "
       << tmp.getname()
       << endl;
  cout << "Address: "
       << tmp.getaddress()
       << endl;
  cout << "Phone:   "
       << tmp.getphone()
       << endl;
}
```

Der Zuweisungsoperator =

Was geschieht beim Aufruf der Funktion `printperson()`?

- `printperson()` erwartet als Parameter `p` eine Referenz auf ein `Person`-Objekt.

- Die Funktion definiert ein lokales Objekt `tmp`.
  Der Standardkonstruktor der Klasse `Person` wird aufgerufen und setzt die Datenelemente von `tmp` alle auf `NULL`.

- `p` wird nach `tmp` kopiert, d.h. `sizeof(Person)` Bytes von `p` werden nach `tmp` kopiert.

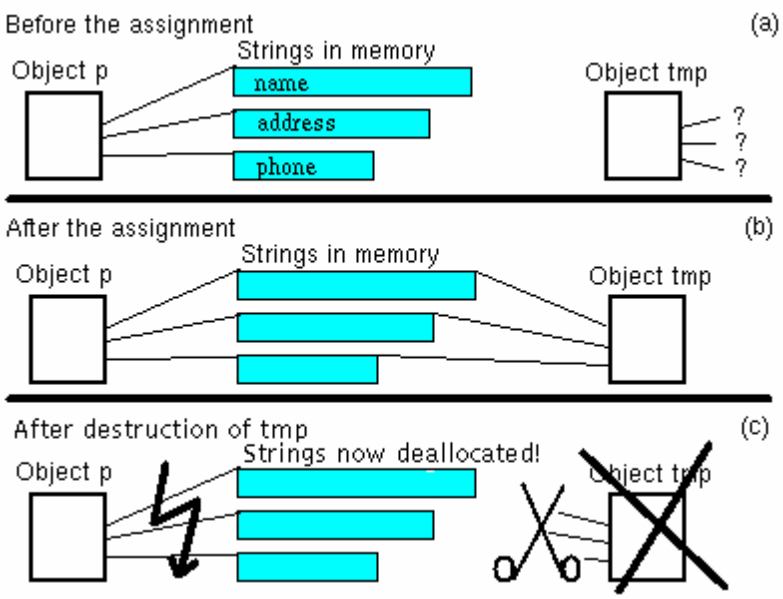
- Beim Verlassen von `printperson()` wird `tmp` zerstört.
  Der Destruktor der Klasse `Person` gibt den Speicherplatz frei, der von `name`, `address` und `phone` belegt wird.

  Bedauerlicherweise wird dieser Speicherplatz auch von `p` verwendet.

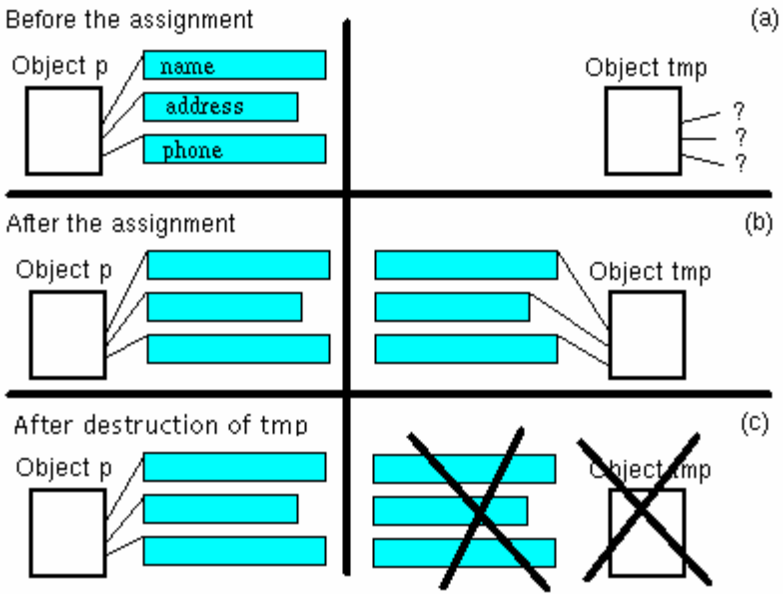
Nach Verlassen von `printperson()` enthält das Objekt, auf das `p` verweist, Zeiger auf bereits freigegebene Speicherbereiche!

Der Zuweisungsoperator =

Der Effekt der byteweisen Wertzuweisung `tmp = p;`



und der Effekt, den wir eigentlich erzielen wollten:



Der Zuweisungsoperator =

Überladen des Zuweisungsoperators =

Lösung des obigen Problems z.B. mit einer speziellen Funktion zur Wertzuweisung von `Person`-Objekten:

```
void Person::assign(Person const &other)
{
  // eigenen allokierten Speicherplatz freigeben
  delete [] name;
  delete [] address;
  delete [] phone;

  // Daten der Person other kopieren
  name = strdupnew(other.name);
  address = strdupnew(other.address);
  phone = strdupnew(other.phone);
}

void printperson(Person const &p)
{
  Person tmp;

  // eine Kopie von p anlegen,
  // dabei Duplikate der Zeichenketten anfertigen
  tmp.assign(p);

  cout << "Name:    " << tmp.getname() << endl;
  cout << "Address: " << tmp.getaddress() << endl;
  cout << "Phone:   " << tmp.getphone() << endl;

  // Zerstörung von tmp richtet kein Unheil mehr an
}
```

Der Zuweisungsoperator =

Überladen des Zuweisungsoperators =

Die Funktion `operator=()` löst das Problem auf elegantere Art:

```
// neue Deklaration der Klasse Person
{
  public:
    ...
    void operator=(Person const &other);
    ...
  private:
    ...
};

// Definition der Funktion operator=()
// Version 1
void Person::operator=(Person const &other)
{
  // alte Daten deallokieren
  delete [] name;
  delete [] address;
  delete [] phone;

  // Daten der Person other duplizieren
  name = strdupnew(other.name);
  address = strdupnew(other.address);
  phone = strdupnew(other.phone);
}
```

Der Zuweisungsoperator =

Überladen des Zuweisungsoperators =

Wie wird die Funktion `operator=()` aufgerufen?

```
Person
  pers("Frank", "Birkenweg 17", "402223"),
  copy;

// erste Möglichkeit
copy = pers;

// zweite Möglichkeit
copy.operator=(pers);
```

Speicherverwaltung

Der `this`-Zeiger

Innerhalb einer Elementfunktion bezeichnet das Schlüssel-wort `this` einen Zeiger auf das aktuelle Objekt und `*this` das Objekt selbst, für das die Elementfunktion aufgerufen wird.

Der `this`-Zeiger ist implizit in jeder Elementfunktion vereinbart.
Für die Klasse `Person` können wir uns vorstellen, dass jede Elementfunktion eine - versteckte - Deklaration enthält:

```
extern Person *this;
```

Beispiel:

```
// Alternative 1: implizite Verwendung von this
  void Person::setname(char const *n)
  {
    delete [] name;
    name = strdupnew(n);
  }

// Alternative 2: explizite Verwendung von this
  void Person::setname(char const *n)
  {
    delete [] this->name;
    this->name = strdupnew(n);
  }
```

Speicherverwaltung

Der `this`-Zeiger

Selbstzerstörung verhindern mit `this`

ein einfaches Beispiel:

```
void func(Person const &p)
{
  p = p;        // Selbstzerstörung durch
                // Zuweisung an sich selbst
}
```

ein etwas komplizierteres Beispiel:

```
Person one, two, *pp;

pp = &one;
...
*pp = two;
...
one = *pp;
```

Der `this`-Zeiger

Selbstzerstörung verhindern mit `this`

```
// Definition der Funktion operator=()
// Version 2
void Person::operator=(Person const &other)
{
  // Zuweisung genau dann, wenn die
  // Adressen der Objekte verschieden sind
  if (this != &other)
  {
    delete [] name;
    delete [] address;
    delete [] phone;

    name = strdupnew(other.name);
    address = strdupnew(other.address);
    phone = strdupnew(other.phone);
  }
}
```

Der `this`-Zeiger des Objekts, das auf der linken Seite der Zuweisung steht, wird mit der Adresse des zuzuweisenden Objekts, d.h. mit der Adresse des Parameters `other`, verglichen.

Wenn die Objekte identisch sind, müssen die Adressen gleich sein.

Die Zuweisung wird nur bei Ungleichheit ausgeführt.

Der `this`-Zeiger

Assoziativität von Operatoren

In C++ ist der Zuweisungsoperator rechtsassoziativ. D.h., eine Anweisung

```
a = b = c;
```

wird ausgewertet wie folgt:

```
a = ( b = c );
```

Bis jetzt erlaubt unsere Implementierung der Funktion `operator=()` diese Konstruktion nicht, da die Funktion den Rückgabetyp `void` hat.

```
a = b = c;
```

bedeutet

```
a.operator=(b.operator=(c));
```

was syntaktisch falsch ist.

```
b.operator=(c)
```

gibt den Typ `void` zurück. Und die Klasse Person besitzt keine Elementfunktion `operator=()` mit dem Prototyp

```
operator=(void)
```

Der this-Zeiger

Assoziativität von Operatoren

```
// Deklaration der Funktion in der Klasse
class Person
{
  public:
    ...
    Person& operator=(Person const &other);
    ...
};

// Definition der Funktion operator=()
// Version 3
Person& Person::operator=(Person const &other)
{
  // Selbstzerstörung verhindern
  if (this != &other)
  {
    // eigene Daten deallokieren
    delete [] name;
    delete [] address;
    delete [] phone;

    // Daten der Person other duplizieren
    name = strdupnew(other.name);
    address = strdupnew(other.address);
    phone = strdupnew(other.phone);
  }

  // Rückgabe des aktuellen Objekts
  return *this;
}
```

Speicherverwaltung

Der Kopierkonstruktor

Beispiel: eine Klasse zur Verwaltung von Zeichenketten

```
class String
{
  public:
    // Konstruktoren, Destruktor
    String();
    String(char const *s);
    ~String();

    // überladene Wertzuweisung
    String& operator=(String const &other);

    // Zugriffsfunktionen
    void set(char const *data);
    char const *get(void);

  private:
    // ein Datenelement: Zeiger auf
    // dynamisch allokierten Speicher
    char *str;
};
```

Der Kopierkonstruktor

Einige Anmerkungen zur Klasse `String`:

- Die Klasse enthält ein Datenelement `char *str`. Daher müssen für die Klasse sowohl Konstruktor als auch Destruktor definiert werden.

- Aus dem gleichen Grund benötigt die Klasse auch einen überladenen Zuweisungsoperator.

```
String& String::operator=(String const &other)
{
  if (this != &other)
  {
    delete [] str;
    str = strdupnew(other.str);
  }
  return *this;
}
```

- Neben dem Standardkonstruktor gibt es einen weiteren Konstruktor, der eine Zeichenkette als Argument hat und z.B. bei der folgenden Definition aufgerufen wird.

```
String a("Hello World!\n");
```

Der Kopierkonstruktor

Beispiele für die Definition von `String`-Objekten:

```
String
  a("Hello World\n"),                  // (1)
  b,                                   // (2)
  c = a;                               // (3)

int main()
{
  b = c;                               // (4)
  return 0;
}
```

- **(1)** Ein `String`-Objekt `a` wird erzeugt und initialisiert. Aufruf des Konstruktors
  `String(char const *s)`
  Die Definition ist identisch mit

  ```
  String a = "Hello World\n";
  ```

- **(2)** Ein `String`-Objekt `b` wird erzeugt. Aufruf des Standardkonstruktors `String()`

- **(3)** Ein `String`-Objekt `c` wird erzeugt und initialisiert. Aufruf des Konstruktors **?????**
  Die Definition ist identisch mit

  ```
  String c(a);
  ```

- **(4)** Eine Wertzuweisung; es wird *kein* Objekt erzeugt.

Der Kopierkonstruktor

Regel:

Wenn ein Objekt erzeugt wird,
wird ein Konstruktor aufgerufen.

Alle Konstruktoren haben die folgenden Eigenschaften:

- Konstruktoren haben keinen Rückgabewert.

- Konstruktoren werden definiert in Funktionen, die den gleichen Namen haben wie die zugehörige Klasse.

- Die Argumentliste des aufzurufenden Konstruktors kann aus dem Code der Objekt-Definition abgeleitet werden.

Der Kopierkonstruktor

Die Klasse `String` muss um einen Kopierkonstruktor erweitert werden.

```
// Klassendeklaration
class String
{
  public:
    ...
    String(String const &other);
    ...
};

// Definition des Kopierkonstruktors
String::String(String const &other)
{
  str = strdupnew(other.str);
}
```

Der Kopierkonstruktor

Ein Kopierkonstruktor wird immer dann benutzt, wenn ein Objekt **erzeugt** und mit einem anderen Objekt seiner Klasse initialisiert wird.

- Die Wert-Übergabe von Objekten an eine Funktion und
- die Wert-Rückgabe eines Ergebnisobjekts

werden ebenfalls als Initialisierung betrachtet, rufen also den Kopierkonstruktor implizit auf.

Beispiel: Wert-Übergabe eines Objekts an eine Funktion

```
void func(String s)     // call-by-value !!
{
  puts(s.get());
}

int main()
{
  String hi("hello world");

  func(hi);
  return 0;
}
```

Der Kopierkonstruktor wird aufgerufen, um eine Kopie des Objekts zu erzeugen, die dann als Argument übergeben wird.

Der Kopierkonstruktor

Beispiel: Wert-Rückgabe eines Objekts aus einer Funktion

```
String getline()
{
  // Puffer für Tastatureingabe
  char buf [100];

  gets(buf);            // Puffer lesen,
  String  ret = buf;    // in String umwandeln
  return ret;           // und zurückgeben
}
```

Im folgenden Beispiel wird der Kopierkonstruktor **nicht** aufgerufen:

```
String getline()
{
  // Puffer für Tastatureingabe
  char buf [100];

  gets(buf);        // Puffer lesen
  return buf;       // und zurückgeben
}
```

Der Kopierkonstruktor wird **nicht** aufgerufen bei Wertzuweisungen, also Änderungen von Objekten.

```
String a = "Hello", b = "World";

a = b;
```

Der Kopierkonstruktor

Ähnlichkeiten zwischen Kopierkonstruktor und überladenem Zuweisungsoperator

- (Private) Daten werden im Kopierkonstruktor und im überladenen Zuweisungsoperator dupliziert.

- Allokierter Speicher wird im Destruktor und im überladenen Zuweisungsoperator freigegeben.

Zwei private Funktionen `copy()` und `destroy()` können diese Aufgaben übernehmen.

```
// nur die hier relevanten Funktionen
// werden deklariert
class Person
{
  public:
    // Konstruktoren, Destruktor
    Person(Person const &other);
    ~Person();
    // überladene Wertzuweisung
    Person& operator=(Person const &other);
  private:
    // Datenelemente
    char *name, *address, *phone;

    // die beiden primitiven Funktionen
    void copy(Person const &other);
    void destroy(void);
};
```

Der Kopierkonstruktor

Ähnlichkeiten zwischen Kopierkonstruktor und überladenem Zuweisungsoperator

```
// copy(): Daten von other duplizieren
void Person::copy(Person const &other)
{
  name = strdupnew(other.name);
  address = strdupnew(other.address);
  phone = strdupnew(other.phone);
}

// destroy(): Daten freigeben
void Person::destroy ()
{
  delete [] name;
  delete [] address;
  delete [] phone;
}
```

Der Kopierkonstruktor

Ähnlichkeiten zwischen Kopierkonstruktor und überladenem Zuweisungsoperator

```
// Kopierkonstruktor
Person::Person (Person const &other)
{
  // Daten von other duplizieren
  copy(other);
}

// Destruktor
Person::~Person()
{
  // Speicher freigeben
  destroy();
}

// überladene Wertzuweisung
Person& Person::operator=(Person const &other)
{
  // Selbstzerstörung verhindern!!
  if (this != &other)
  {
    destroy();
    copy(other);
  }
  // Referenz auf das aktuelle Objekt zurück-
  // geben (Verkettung von Zuweisungen)
  return *this;
}
```

Speicherverwaltung

Regel 1:

Wenn eine Klasse Zeiger als Datenelemente verwendet, dann müssen

- **Destruktor**
- **Kopierkonstruktor**
- **überladener Zuweisungsoperator**

definiert werden.

Regel 2:

Wenn einer der drei benötigt wird, dann werden auch die anderen beiden benötigt.