Programmieren in C++

SS 2012

Vorlesung 6, Dienstag 12. Juni 2012

(Funktionen: Argumentübergabe & Ergebnisrückgabe, copy constructor, assignment operator)

Prof. Dr. Hannah Bast
Lehrstuhl für Algorithmen und Datenstrukturen
Institut für Informatik
Universität Freiburg

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Erfahrungen mit dem 5. Übungsblatt
  - Treffen mit Ihrem Tutor / Ihrer Tutorin
- Themen heute
  - Der & und der -> Operator
  - Was genau passiert bei einem Funktionsaufruf
  - Übergabe von Argumenten: call bei value, call by reference
  - Was ist const-correctness
  - copy constructor, assignment operator, this Zeiger
  - Rückgabe von (komplexen) Ergebnissen
  - Übungsblatt: eine String Klasse mit allerlei nützlichen Methoden
    - insert, erase, find, findAndReplace, ...

Erfahrungen mit dem Ü5 (OO-Wurm)

UNI FREIBURG

- Zusammenfassung / Auszüge Stand 12.6 14:00
  - die meisten gut machbar / wieder einfacher als das Ü4
  - "Auch für C++ Anfänger gut zu schaffen"
  - Bei einigen länger gedauert als erwartet
  - Aber einige haben sich schwer getan / lange gebraucht
  - Endlich objekt-orientiert
  - Übergang anhand von Beispiel fanden viele gut, einige nicht
  - Unterschiede zwischen Java und C++ interessant
  - Studenten sollen in der VL mal die Klappe halten
  - Vorlesung auch für Fortgeschrittene nicht langweilig
  - Erstaunt, wie einfach die Musterlösung war
  - Musterlösung war nicht direkt nach der VL online

Erfahrungen mit dem Ü5 (OO-Wurm)

UNI FREIBURG

- Fortsetzung ...
  - Tests besser am Anfang schreiben und nicht am Ende ... genau!
  - gdb war bei Vielen hilfreich zum Debuggen
  - Fachbücher, weil man sonst von der VL nicht viel mitnimmt ... ??
  - FRIEND_TEST wurde in der VL nicht erklärt ... doch, sogar mit Bsp
  - Methoden mit Zufall kann man nicht testen ... doch
  - this hätte noch erklärt werden können / sollen ... diese VL
  - cpplint.py ist **noch** pingeliger geworden ... nur bez. Einrückung
    - vorrübergehend Probleme mit else ... sind jetzt gefixt
  - Zusätzliche Variablen eingeführt für kürzere Zeilen ... nicht nötig
  - Jetzt bitte keine Würmer mehr ... keine Sorge

Treffen mit Ihrem Tutor / Ihrer Tutorin

UNI FREIBURG

- Sie …
  - müssen sich einmal mit Ihrem/r Tutor/in treffen
  - Für maximal eine halbe Stunde
  - Grund: wir wollen persönlich schauen:
    - … wie es Ihnen geht und ob es Probleme gibt
    - … ob Sie ein Mensch sind und insbesondere der Mensch, der die Übungsblätter in Ihrem Namen macht
- Ablauf
  - Sie bekommen dazu eine Mail von Ihrem/r Tutor/in
    - an Ihre Mail Adresse aus Daphne
    - also stellen Sie bitte sicher, dass die auch stimmt
  - In der Mail steht dann alles Weitere

Der Adress Operator &

- Hat zwei ganz verschiedene Anwendungen
  - Vor einer Variablen gibt einem & die Adresse der ersten Speicherzelle dieser Variablen im Speicher

```
int x = 5;
int* p = &x;  // Pointer to the value of x.
*p = 4;
printf("%d\n", x);  // This will print 4.
```

  - In einer Deklaration wird die entsprechende Variable zu einem sogenannten **Alias**, z.B.

```
int x = 5;
int& y = x;  // Now y is like another name for x.
y = 4;
printf("%d\n", x);  // This will print 4.
```

Der -> Operator

- Nützlich bei Zeigern auf Objekte
  - Nehmen wir an wir haben einen Zeiger auf ein Objekt

```
String* p;
p = new String();
```

  - Will man da jetzt eine Methode aufrufen oder eine Membervariable zuweisen kann man schreiben

```
// Call size() method of String object pointed to by p.
printf("%d\n", (*p).size());
```

  - Weil das so häufig vorkommt, gibt es dafür eine Abkürzung

```
printf("%d\n", p->size());  // Exactly the same as above.
```

Wie ein Funktionsaufruf funktioniert 1/3

- Nehmen wir an wir haben die Funktion

  ```
  int square(int z) { int result = z * z; return result; }
  ```

  – Wenn wir jetzt einen Aufruf haben

  ```
  int x = 5;
  y = square(x);
  ```

  – Dann passiert sinngemäß Folgendes

  ```
  int x = 5;
  {  int z = x; int result = z * z; y = result; }
  ```

  – Das heißt der **Wert** der Variablen x wird in die für die Funktion lokale Variable z **kopiert** → **call by value**

  – Ein int hat nur 4 – 8 Bytes, da ist das Kopieren kein Problem

  – Aber bei einem größeren Objekt mit hunderten oder Millionen von Bytes kostet Kopieren richtig Zeit

Wie ein Funktionsaufruf funktioniert 2/3

- Nehmen wir jetzt an wir haben die Funktion

```
int square(int* z) { int result = (*z) * (*z); return result; }
```

  - Wenn wir jetzt einen Aufruf haben

```
int x = 5;
y = square(&x);  // We need to pass an int* now.
```

  - Dann passiert sinngemäß Folgendes

```
int x = 5;
{  int* z = &x; int result = (*z) * (*z); y = result; }
```

  - Hier wird nur die **Speicheradresse** der Variablen **kopiert**, das sind 4 – 8 bytes je nach Rechnerarchitektur
  - Bei einem int egal, bei großen Objekten großer Unterschied
  - Nachteil: man hat überall in der Funktion Zeiger und muss bei der Übergabe & und in der Funktion * schreiben

Wie ein Funktionsaufruf funktioniert 3/3

- Nehmen wir jetzt noch an wir haben die Funktion

  ```
  int square(int& z) { int result = z * z; return result; }
  ```

  – Wenn wir jetzt einen Aufruf haben

  ```
  int x = 5;
  y = square(x);
  ```

  – Dann passiert sinngemäß Folgendes

  ```
  int x = 5;
  { int& z = x; int result = z * z; y = result; }
  ```

  – Das wird intern genauso realisiert wie das mit den Zeigern von der Folie vorher, aber jetzt braucht man weder im Aufruf das & noch in der Funktion z → **call by reference**

  – Und eigentlich sollte die Funktion natürlich lauten

  ```
  int square(const int& z) { int result = z * z; return result; }
  ```

Funktionsaufruf — const Argumente

- Ein const vor einem Argument
  - … heißt genau dasselbe, wie das const bei einer Deklaration, nämlich dass der Wert dieser Variablen in der Funktion nicht verändert werden darf

```
int square(const int x) {
  x = x * x; // This will not compile, because x is const.
  return x;
}
```

  - Wann immer eine Variable nicht verändert werden soll, sollte man const davor schreiben → **const correctness**
  - Wenn allerdings ein Argument by value übergeben wird (wie typischerweise bei den Basistypen int, float, char, …), lässt man das const in der Praxis oft weg … warum?

const Methoden

- Ein const **hinter** einer Methodendeklaration
  - … bedeutet, dass die Methode keine Membervariable des Objektes verändert; in der .h Datei:

```
class String {
   int size() const;  // Must not change any member var's.
   int _size;
}
```

  - In der .cpp Datei

```
int String::size() const { return _size; }
```

  - Methoden, die das Objekt nicht verändern, sollen const deklariert werden, auch das gehört zur const correctness
  - Die typische (etwas kryptische) Fehlermeldung dazu:

```
assignment of data member ... in read-only structure
```

mutable

- Einzelne Membervariablen von dem const ausschließen
  - Typischerweise Variablen, die nicht direkt etwas mit der Kernfunktionalität des Objektes zu tun haben
  - Vor die schreibt man dann **mutable** (engl.: veränderlich)
  - Hier ein (unrealistisches, aber immerhin) Beispiel:

```
class String {
  int _size;
  mutable int _numCallsToSize;
  void size() const {
    _numCallsToSize++;  // Ok, since declared mutable.
    return _size;
  }
}
```

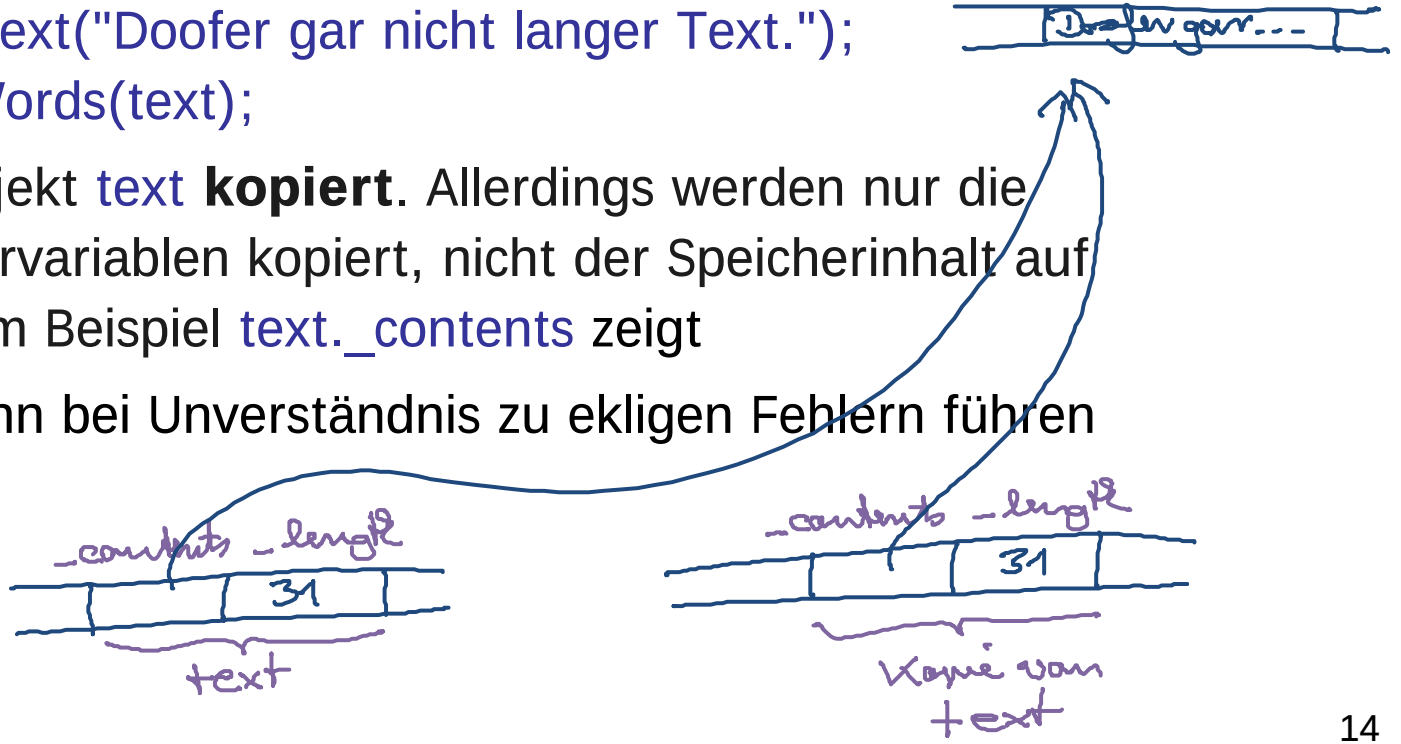
Wie wird ein Objekt kopiert 1/2

- Beispiel
  - Nehmen wir an, wir machen hier call by value (ohne &)

    countWords(String text); // Maybe a very long text.
  - Dann wird bei einem Aufruf

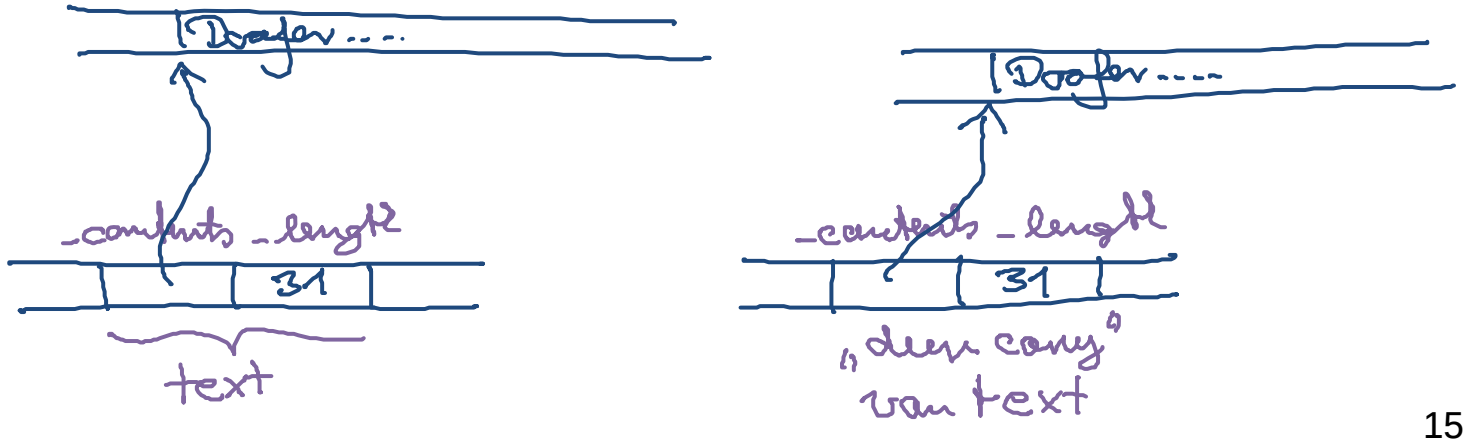
    String text("Doofer gar nicht langer Text.");
    countWords(text);

    das Objekt text **kopiert**. Allerdings werden nur die Membervariablen kopiert, nicht der Speicherinhalt auf den zum Beispiel text._contents zeigt
  - Das kann bei Unverständnis zu ekligen Fehlern führen



Wie wird ein Objekt kopiert 2/2

- Man nennt das eine shallow copy
  - Das heißt, es werden einfach die Membervariablen kopiert
  - Falls da Zeiger dabei sind, werden einfach die Zeiger kopiert, aber **nicht** das worauf Sie zeigen
  - Will man das auch kopieren, spricht man von einer **deep copy** … dafür muss man einen copy constructor schreiben
  - Das brauchen wir aber meistens nicht
  - Faustregel: bei Übergabe von Objekten (fast) immer const &



Copy constructor 1/3

- Beim Aufruf einer Funktion mit einem Objekt
  - … als Argument mit call by value wird das Objekt **kopiert**
  - Und zwar macht das der copy constructor
  - Den gibt es auch, ohne dass man ihn implementiert, dann macht er die erwähnte shallow copy
  - Man kann aber auch selber einen schreiben
  - Der copy constructor heißt, wie auch der default constructor, wie die Klasse und hat **keinen** return type
  - Als Argument ein const& auf ein Objekt der Klasse

    String::String(const String& x);

Copy constructor 2/3

- Hier eine Implementierung
  - … die eine shallow copy macht … so wie sie der default copy constructor auch schon gemacht hätte

```
String::String(const String& x) {
   _size = x._size;
   _contents = x._contents;
}
```

  - Es wird einfach Membervariable für Membervariable von dem einem Objekt in das andere kopiert
  - **Wichtig (nochmal):** in dem Beispiel wird nur der Zeiger kopiert, nicht der Speicher auf den der Zeiger zeigt!

Copy constructor 3/3

- Hier eine Implementierung
  - … die eine deep copy macht

```
String::String(const String& x) {
   _size = x._size;
   _contents = new char[_size];
   for (int i = 0; i < _size; i++) {
      _contents[i] = x._contents[i];
   }
}
```

  - Wir haben jetzt den Speicher, auf den _contents von dem alten Objekt zeigt, kopiert
  - Und das _contents von dem neuen Objekt zeigt jetzt auf den neuen Speicher

Assignment operator 1/2

- Was passiert eigentlich bei dieser Anweisung

```
String message1("Alles doof");
String message2 = message1;  // What happens here?
String message3;
message3 = message2;  // And what happens here?
```

  - Bei der ersten Zuweisung wird tatsächlich der copy constructor aufgerufen (obwohl es nicht so aussieht)
  - Bei der zweiten Zuweisung wird der sogenannte **assignment operator** aufgerufen
  - Den gibt es auch immer, ohne dass man ihn explizit implementiert, und dann macht er, wie der default copy constructor, auch nur eine shallow copy

Assignment operator 2/2

- Man kann ihn auch selber implementieren
  - Deklaration innerhalb der Klasse in der .h Datei:

```
String& operator=(const String& x);
```

  - Für eine deep copy in der .cpp Datei zum Beispiel so:

```
String& String::operator=(const String& x) {
   _size = x._size;
    _contents = new char[_size];
   for (int i = 0; i < _size; i++) {
      _contents[i] = x._contents[i];
   }
   return *this;  // "this" is explained on next slide.
}
```

this Zeiger

- Der this Zeiger
  - … kann nur innerhalb einer Methode aufgerufen werden und zeigt dort einfach auf das Objekt von dem aus diese Methode aufgerufen wurde

```
void String::printAddress() {
   printf("%p\n", this);  // Print the address of this object.
}

...

String x;
x.printAddress();  // Will print the address of x.
printf("%p\n", &x);  // Will print exactly the same.
```

  - Entsprechend ist ***this** das aktuelle Objekt selber

Rückgabe von Objekten 1/6

UNI FREIBURG

- Möglichkeit 1: "Normale" Rückgabe
  - Implementierung

```
String answerQuestion(const String& question) {
   String answer = "doof";
   return answer;
}
```

  - Aufruf

```
String question = "Würmer sind ... ?";
String answer;
answer = answerQuestion(question);
```

  - Nachteil: das Objekt wird bei der Rückgabe **kopiert**
    - Das ist **ineffizient** bei großen Objekten

Rückgabe von Objekten 2/6

- Möglichkeit 2: Rückgabe einer Referenz
  - Implementierung

```
String& answerQuestion(const String& question) {
  String answer = "doof";
  return answer;
}
```

  - Da meckert der Kompiler meckert ... warum?
    - answer existiert nur lokal in der Funktion answerQuestion
    - Einen Zeiger auf etwas zurückzugeben, was dann nicht mehr existiert, ist eine schlechte Idee

Rückgabe von Objekten 3/6

- Möglichkeit 3: Rückgabe eines Zeigers, Versuch 1
  - Implementierung

```
String* answerQuestion(const String& question) {
   String answer = "doof";
   return &answer;
}
```

  - Kompiliert nicht aus demselben Grund

Rückgabe von Objekten 4/6

- Möglichkeit 4: Rückgabe eines Zeigers, Versuch 2
  - Implementierung

```
String* answerQuestion(const String& question)
{
   String* answer = new String("doof");
   return answer;
}
```

  - Das kompiliert … und funktioniert auch … aber sollte man in der Regel trotzdem nicht tun
    - weil in der Funktion Speicherplatz alloziert wurde
    - der dann später irgendwann, außerhalb der Funktion, wieder freigegeben werden muss
    - die Gefahr ist groß, dass man das vergisst

Rückgabe von Objekten 5/6

- Möglichkeit 5: Rückgabe via Argument, Variante A
  - Implementierung

```
void answerQuestion(const String& question, String& answer) {
  answer = "doof";
}
```

  - Aufruf

```
String question = "Würmer sind ... ?";
String answer;
answerQuestion(question, answer);
```

  - Das kompiliert ... aber gefällt cpplint.py / checkstyle nicht
    - weil man dem Aufruf nicht ansieht, dass answer von der Funktion verändert wird

Rückgabe von Objekten 6/6

- Möglichkeit 6: Rückgabe via Argument, Variante B
  - Implementierung

```
void answerQuestion(const String& question, String* answer) {
    answer = "doof";
}
```

  - Aufruf

```
String question = "Würmer sind ... ?";
String answer;
process(question, &answer);
```

  - So hätten wir und cpplint.py / checkstyle das gerne!
  - Weiterer Vorteil gegenüber Rückgabe mit return:
    - so kann man auch leicht **mehrere** Objekte zurückgeben

Literatur / Links

- Const und die const correctness
  - http://www.parashift.com/c++-faq-lite/const-correctness.html
- Adressoperator &
  - http://www.cplusplus.com/doc/tutorial/pointers
- Call by value / call by reference
  - http://www.cplusplus.com/doc/tutorial/functions2/
- Copy constructor
  - http://www.cplusplus.com/doc/tutorial/classes/