Programmieren in C++

SS 2010

Vorlesung 6, Mittwoch 2. Juni 2010

(const and &, call by value / reference, Operatoren)

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
  - Tutor Kennenlernen
- Ein paar neue C++ Konzepte
  - Was genau bedeutet const und wofür ist es gut
  - Was genau bedeutet & und wofür ist es gut
  - Wie werden eigentlich die Argumente bei einem Funktionsaufruf übergeben → call by value / call by reference
  - Was sind Operatoren und wofür sind sie gut

  - das neue Übungsblatt ist vom Prinzip wie das letzte, eher etwas einfacher, und eine andere (neue) Aufgabenstellung

Erfahrungen mit dem 5. Übungsblatt

UNI FREIBURG

- Zusammenfassung / Auszüge
  - leichter als das Übungsblatt vorher
  - aber trotzdem recht zeitintensiv
  - Übungsblatt immer viel schwerer als die Vorlesung
  - was genau bedeutet dieses const und dieses &
  - FRIEND_TEST nochmal in der Vorlesung erklären
  - bitte präzisere Aufgabenstellung
  - bitte ein neues Projekt, kein altes mehr abändern
  - bitte Forum statt Wiki
  - warum push_back und nicht pushBack?
  - warum _ für Membervariablen?
  - Editor XYZ ist cooler als Vim
  - wie debuggt man?

Tutor Kennenlernen

- Sie sollten Ihren Tutor kennenlernen ...
  - ... und ihr Tutor Sie
  - Auf diesem Wege noch einmal Feedback wie es Ihnen mit den Übungsblättern ergeht
  - Aber auch Kontrolle für uns, ob Sie die Übungsblätter auch wirklich selber gemacht haben
  - Ihr Tutor wird Sie ab und an (nicht jede Woche) anschreiben und zu einem Termin bitten (ca. 15 Minuten)
  - Bitte teilen Sie ehrlich mit, was Sie selber gemacht haben und wo Sie sich Hilfe geholt haben
  - Sich helfen lassen, auch sehr viel, wenn Sie Probleme haben, ist in Ordnung
  - Einfach Sachen von jemand anderem kopieren oder machen lassen, ohne es selber zu verstehen, ist Betrug

Const 1/2

- Const steht vor Deklarationen einer Variable

```
const int Pi = 3;
```

  - Das const bedeutet, dass man den Wert dieser Variablen nicht mehr verändern darf
  - Wann immer Variablen nur zum Lesen gedacht sind, sollte man const davor schreiben, als Schutz
  - const vor einem Zeiger bedeutet, dass man zwar den Wert des Zeigers verändern darf, aber nicht den Speicher auf den der Zeiger zeigt

```
const char* message = "Do not change me";
message[0] = 'd';  // Will produce a compiler error.
message = "Some other string";  // This is fine though.
```

Const 2/2

- Das Umgekehrte gibt es auch

```
char * const fixedPointer = "You belong to me";
fixedPointer[0] = y;  // This is fine now.
fixedPointer = "Not to me";  // This will not compile.
```

  - aber das braucht man äußerst selten
- Const nach einer Methodendeklaration

```
void ListOfIntegers::print() const;
```

  - Das bedeutet, dass der Aufruf dieser Methode keiner der Membervariablen des Objektes verändern darf
  - Wenn eine Methode ein Objekt nicht verändert bzw. verändern soll, immer const dahinter schreiben, auch wieder zum Schutz vor Programmierfehlern

Der Adress Operator &

- Hat zwei ganz verschiedene Anwendungen
  - in einer Deklaration wird die entsprechende Variable zu einem Alias, z.B.

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

  - Vor einer Variablen gibt einem & die Adresse der ersten Speicherzelle dieser Variablen im Speicher

```
int x = 5;
int* xPointer = &x;
xPointer[0] = 4;
printf("%d\n", x);  // This will print 4.
```

Wie ein Funktionsaufruf funktioniert 1/3

UNI FREIBURG

- Nehmen wir an wir haben die Funktion

```
void print(int x) { printf("%d\n", x); }
```

  - wenn wir jetzt einen Aufruf haben

```
int y = 5;
print(y);
```

  - dann passiert sinngemäß Folgendes

```
int y = 5;
{  int x = y;  printf("%d\n", x);  }
```

  - das heißt der Wert der Variablen y wird in die für die Funktion lokale Variable x kopiert → **call by value**
  - ein int hat nur 4 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

```
void print(int* x) { printf("%d\n", *x); }
```

  - wenn wir jetzt einen Aufruf haben

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

  - dann passiert sinngemäß Folgendes

```
int y = 5;
{  int* x = &y;  printf("%d\n", *x);  }
```

  - 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, wenn man den Wert braucht, sowas wie *x schreiben

Wie ein Funktionsaufruf funktioniert 3/3

- Nehmen wir jetzt noch an wir haben die Funktion

```
void print(int& x) { printf("%d\n", x); }
```

  - wenn wir jetzt einen Aufruf haben

```
int y = 5;
print(y);
```

  - dann passiert sinngemäß Folgendes

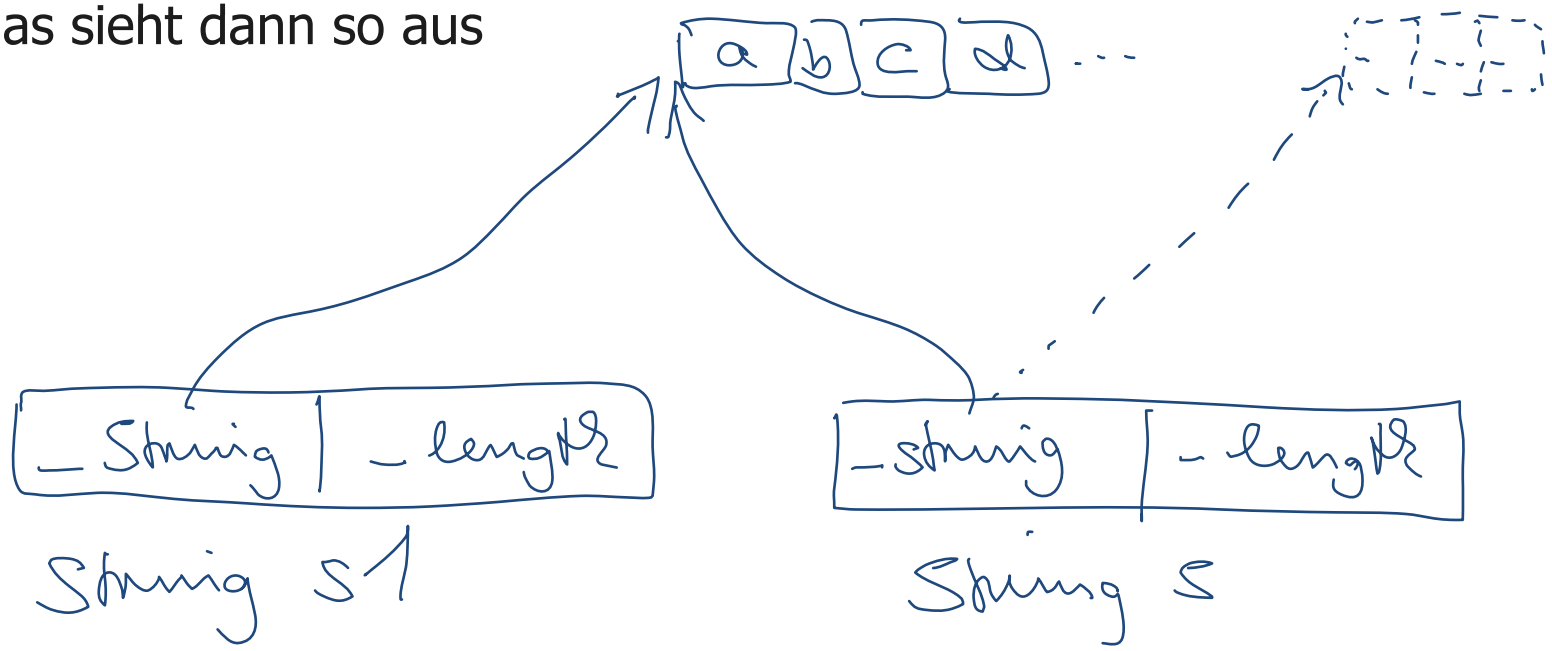
```
int y = 5;
{  int& x = y;  printf("%d\n", x);  }
```

  - 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 *x → **call by reference**
  - Und eigentlich sollte die Funktion natürlich lauten

```
void print(const int& x) { printf("%d\n", x); }
```

Wie wird ein Objekt kopiert 1/2

- Es wird defaultmäßig eine shallow copy gemacht
  - das heißt, es werden einfach die Membervariablen kopiert
  - Falls da Zeiger dabei sind, wird der Speicherinhalt auf den die Zeiger zeigen **nicht** kopiert
  - das sieht dann so aus



Wie wird ein Objekt kopiert 2/2

- Beispiel ListOfIntegers::getInterleaveOf
  - nehmen wir an, wir haben kein & geschrieben

    getInterleaveOf(ListOfIntegers x, ListOfInteger y)
  - dann werden bei einem Aufruf

    ListOfIntegers result;

    result.getInterleave(list1, list2)

    die Objekte list1 und list2 kopiert. Allerdings werden nur die Membervariablen kopiert, nicht der Speicherinhalt auf den list1._elements und list2._elements zeigen
  - will man den auch kopieren — das wäre dann eine deep copy — muss man einen copy constructor schreiben
  - das brauchen wir aber noch lange nicht, Faustregel bis dahin: **bei Übergabe von Objekten immer const &**

Operatoren

- Ein Operator ist eine ganz normale Methode

```
// Return the character at position i.
char String::operator[](int i);
```

  - man kann sie auch so aufrufen wie eine normale Methode

```
String s;
s.set("abc");
char x = s.operator[](2);
```

  - Aber man kann sie auch über den für den jeweiligen Operator typische Syntax aufrufen

```
char x = s[2];
```

  - es gibt auch noch operator(), operator+, operator*, operator-, operator/, operator<<, operator>>, usw.

FRIEND_TEST

UNI FREIBURG

- Wofür braucht man das?
  - Nur die Methoden einer Klasse haben Zugriff auf die privaten Membervariablen
  - Wenn man eine Methode testet, ist es aber sehr oft sehr praktisch, Zugriff auf die Membervariablen zu haben
  - Das geht mit dem FRIEND_TEST Makro, dazu schreibt man in der .h Datei über die Deklaration der Methode

    ```
    FRIEND_TEST(ListOfIntegersTest, getInterleaveOf)
    void getInterleaveOf(const ListOfIntegers& list1,
                         const ListOfIntegers& list2);
    ```

  - Dann kann man in der ...Test.cpp Datei in dem Test

    ```
    TEST(ListOfIntegersTest, getInterleaveOf)
    ```

    auf die Membervariablen des Objektes zugreifen

Debugging

UNI FREIBURG

- Wie debuggt man am besten
  - Regel #1: Fehler von Anfang an vermeiden.
  - Regel #2: Immer möglichst wenig Code auf einmal schreiben und dann compilieren und testen.

    (Auch wenn man dafür Umwege geht und Sachen hinschreibt die man dann später wieder löscht.)
  - Regel #3: Wenn es nicht funktioniert, erstmal auf den Code gucken und schauen ob das auch wirklich Sinn macht, was da steht.
  - Regel #4: Dann erst an strategischen Stellen mit printf die Werte von zum Problem gehörigen Variablen ausgeben, um zu schauen was da schief läuft
  - Regel #5: Debuggen mit low-level Werkzeugen wie dem gdb, das lernen wir später aber nicht jetzt

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
- Operatoren
  - http://www.cplusplus.com/doc/tutorial/operators