Programmieren in C++

SS 2011

Vorlesung 6, Mittwoch 8. Juni 2011

(call by value / reference, copy constructor, ...)

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
  - Ihr Tutor und Sie / Fragetermine
  - Plagiate und ihre Konsequenzen
- Themen heute
  - Der & und der -> Operator
  - Was genau passiert bei einem Funktionsaufruf
  - Call bei value, call by reference
  - Const-correctness
  - Copy constructor, assignment operator, this Zeiger
  - Übungsblatt: eine Klasse String mit allerlei nützlichen Methoden
    - copy, assign, insert, erase, find, replace, ...

Erfahrungen mit dem Ü5 (OO + new)

UNI FREIBURG

- Zusammenfassung / Auszüge Stand 8.6 15:39
  - Bestehenden Code objektorientiert zu machen war gut
  - Im Prinzip nicht schwer, aber (Zeiger sind) tricky
  - Viele Zeiger Fehler, seg faults, double free, ...
  - Doofe Fehler kosten manchmal viel Zeit
  - Probleme bei der Übergabe eines Objektes als Argument **!!!**
  - War es mit dem in der Vorlesung gezeigten lösbar?
  - Wann übergibt man Objekte, wann Zeiger, wann Referenzen?
  - Enttäuscht von Musterlösung für 1b
  - Java hat auch einen Defaultkonstruktor
  - Linter: was ist der Sinn der Whitespace-Regel?

Ihr Tutor und Sie

UNI FREIBURG

- Sie
  - … sollen sich mindestens einmal mit Ihrem Tutor treffen
  - Für eine halbe Stunde
  - Grund 1: wir wollen persönlich schauen, wie es Ihnen geht und ob es Probleme gibt
  - Grund 2: wir wollen nicht, das jemand schummelt
- Ihr Tutor
  - … korrigiert Ihre Abgaben und gibt Ihnen Feedback
  - Steht auch sonst für Fragen zur Verfügung
  - Wenn Sie nicht zufrieden sind, schreiben Sie mir bitte eine Mail
  - Wenn Sie besonders zufrieden sind auch gerne

Fragestunde

UNI FREIBURG

- Zur letzten Fragestunde kam keiner mehr
  - Von daher ab jetzt keine fixe Fragestunde mehr
  - Stattdessen nach Vereinbarung mit Ihrem Tutor
    - Schreiben Sie ihm / ihr bei Problemen, die über das hinausgehen was man sinnvoll über das Forum lösen kann, eine Mail

Plagiate und ihre Konsequenzen

UNI FREIBURG

- Schummeln verboten
  - Es gibt keine Klausur für diese Veranstaltung
  - Die Prüfungsleistung ergibt sich alleine aus den Punkten für die Übungsblätter (und dem Projekt am Ende)
  - Plagiate werden von daher geahndet wie Abschreiben bei einer Klausur

Der Adress Operator &

UNI FREIBURG

- 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

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

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

```
(*p).parseFromString("1 2 3");
printf("%d\n", (*p)._size);  // Assumes that _size is public.
```

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

```
p->parseFromString("1 2 3");
printf("%d\n", p->_size);
```

Wie ein Funktionsaufruf funktioniert 1/3

UNI FREIBURG

- 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 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

UNI FREIBURG

- Nehmen wir jetzt an wir haben die Funktion

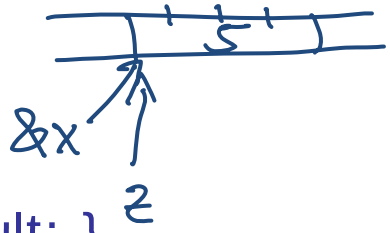
  ```
  int square(int* z) { int result = (*z) * (*z); return result; }
  ```

  – 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, wenn man den Wert braucht, sowas wie *z schreiben

Wie ein Funktionsaufruf funktioniert 3/3

UNI FREIBURG

- 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

UNI FREIBURG

- 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**
  - Bei Funktionsargumenten mit Basistypen (int, float, etc.), die by value übergeben werden, lässt man es in der Praxis allerdings oft weg

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;  // Return the size of the string.
   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

mutable

UNI FREIBURG

- const für einen Teil der Membervariablen
  - Oft sollen bestimmte Membervariablen vom dem const einer Methode ausgeschlossen sein
  - Vor die schreibt man dann **mutable** (engl.: veränderlich)

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

Wie wird ein Objekt kopiert 1/2

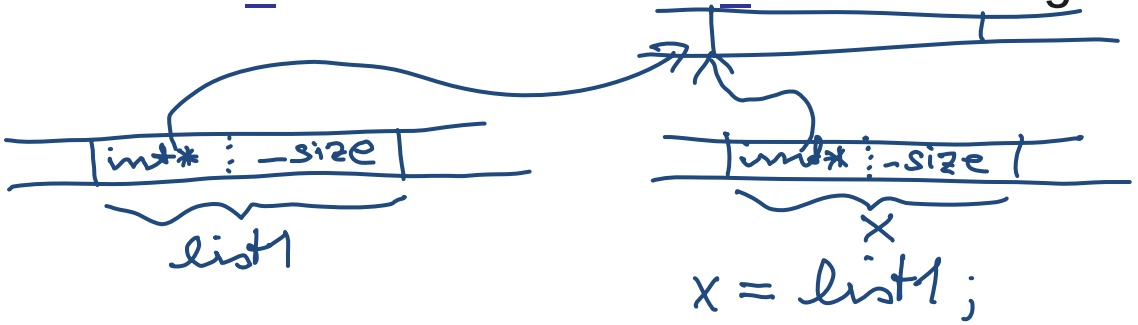
- Beispiel ListOfIntegers::getInterleaveOf
  - Nehmen wir an, wir haben kein & geschrieben

    computeIntersection(SetOfIntegers x, SetOfInteger y)
  - Dann werden bei einem Aufruf

    SetOfIntegers result;

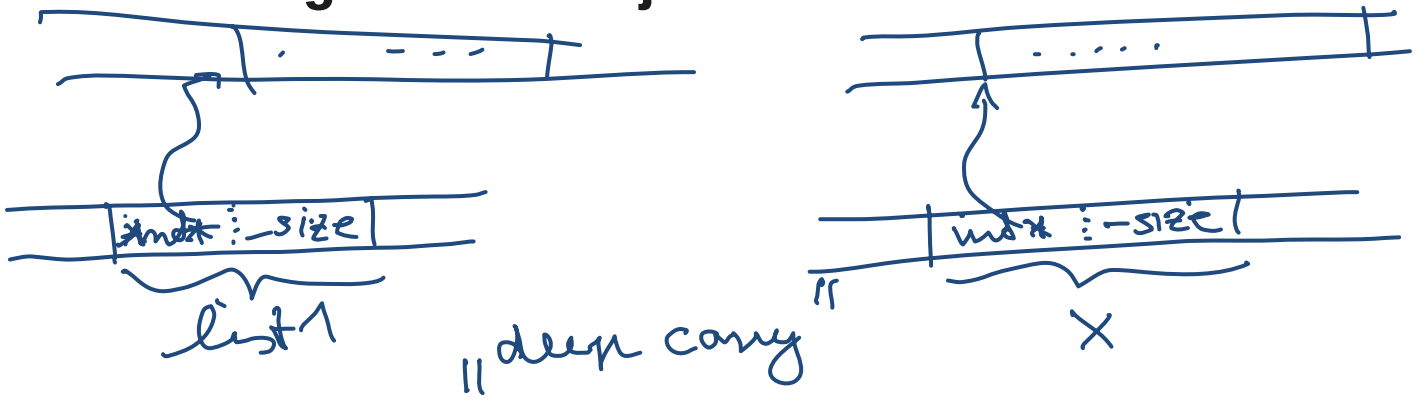
    result.computeIntersection(list1, list2)

    die Objekte list1 und list2 kopiert. Allerdings werden nur die Membervariablen kopiert, nicht der Speicherinhalt auf den list1._elements und list2._elements zeigen



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, wird der Speicherinhalt auf den die Zeiger zeigen **nicht** kopiert
  - Will man den auch kopieren — das wäre dann eine **deep copy** — muss man einen **copy constructor** schreiben
  - Das brauchen wir aber meistens nicht, als Faustregel gilt: **bei Übergabe von Objekten immer const &**



Copy constructor 1/3

UNI FREIBURG

- 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

    SetOfIntegers::SetOfIntegers(const SetOfIntegers& x);

Copy constructor 2/3

UNI FREIBURG

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

```
SetOfIntegers::SetOfIntegers(const SetOfIntegers& x)
{
   _size = x._size;
   _elements = x._elements;
}
```

  - 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

UNI FREIBURG

- Hier eine Implementierung
  - … die eine deep copy macht

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

  - Wir haben jetzt den Speicher, auf den _elements von dem alten Objekt zeigt, kopiert, und der Zeiger von dem neuen Objekt zeigt jetzt auf den neuen Speicher

Assignment operator 1/3

UNI FREIBURG

- Was passiert eigentlich bei dieser Anweisung

```
SetOfIntegers x;
x.parseFromString("{1, 6, 15}");
SetOfIntegers y = x;  // What happens here?
SetOfIntegers z;
z = x;  // And here?
```

  - Bei der zweiten Zuweiung 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, eine shallow copy

Assignment operator 2/3

- Man kann ihn auch selber implementieren
  - Für eine deep copy in der .cpp Datei zum Beispiel so:

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

  - Deklaration innerhalb der Klasse in der .h Datei analog:

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

Assignment operator 3/3

UNI FREIBURG

- Achtung bei Initialisierungen bei Deklarationen
  - Zum Beispiel hier

    ```
    String s1 = s2;  // Calls the copy constructor!
    ```

  - Hier wird der copy constructor aufgerufen, und nicht der assignment operator

this Zeiger

UNI FREIBURG

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

...

SetOfIntegers 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 process(const String& question)
{
   String answer = "42";
   return answer;
}
```

  - Aufruf

```
String question = "What is the meaning of life?";
String answer;
answer = process(question);
```

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

Rückgabe von Objekten 2/6

UNI FREIBURG

- Möglichkeit 2: Rückgabe einer Referenz
  - Implementierung

```
String& process(const String& question)
{
  String answer = "42";
  return answer;
}
```

  - Der Kompiler meckert, warum?
    - answer existiert nur lokal in der Funktion process, und einen Zeiger auf etwas zurückzugeben, was dann nicht mehr existiert, ist eine schlechte Idee

Rückgabe von Objekten 3/6

UNI FREIBURG

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

```
String* process(const String& question)
{
   String answer = "42";
   return &answer;
}
```

  - Kompiliert nicht aus demselben Grund

Rückgabe von Objekten 4/6

UNI FREIBURG

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

```
String* process(const String& question)
{
   String* answer = new String("42");
   return answer;
}
```

  - Das kompiliert ... und funktioniert auch
  - Aber sollte man in der Regel 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

UNI FREIBURG

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

```
void process(const String& question, String& answer)
{
   answer = "42";
}
```

  - Aufruf

```
String question = "What is the meaning of life?";
String answer;
process(question, answer);
```

  - Das gefällt dem Linter nicht
    - weil man dem Aufruf nicht ansieht, dass answer von der Funktion verändert wird

Rückgabe von Objekten 6/6

UNI FREIBURG

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

```
void process(const String& question, String* answer)
{
   *answer = "42";
}
```

  - Aufruf

```
String question = "What is the meaning of life?";
String answer;
process(question, &answer);
```

  - So hätten wir und der Linter das gerne!
  - Weiterer Vorteil gegenüber Rückgabe mit return: so kann man auch 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/