Programmieren in C++

SS 2018

Vorlesung 6, Dienstag 5. Juni 2018

(Dynamische Speicherallokation, Funktionen: Argumentübergabe & Ergebnisrückgabe, Const)

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 Ü5 — Game of Life OO
  - Treffen mit Ihrerm Tutor*in — siehe Folie 6
- Inhalt
  - Dynamische Allokation — new und delete
  - Übergabe von Argumenten — call by value / reference
  - Konstanten — const
  - Rückgabe von Objekten — sechs Varianten
  - **Ü6:** Implementierung einer einfachen "String" Klasse

Erfahrungen mit dem Ü5 1/3

UNI FREIBURG

g++ -std=c++11

- Zusammenfassung / Auszüge
  - Deutlich weniger Arbeit als das Ü4
  - Man musste nur sehr wenig neuen Code schreiben
  - Viele sind von der Musterlösung für das Ü4 ausgegangen
  - Wenn dann haben Flüchtigkeitsfehler Zeit gekostet
  - "Blatt hat keinen Spaß gemacht, aber OO muss sein"
  - Bitte den Debugger erklären, den wir in Vorlesung 4 aus Zeitgründen übersprungen haben … machen wir heute!
  - In C++11 Initialisierung **in** der Klassendeklaration möglich:

    class X { … int a[2][2] = { { 1, 2 }, {3, 4} } … };

    Bei Feldern praktisch, weil = { … } danach nicht mehr geht

Erfahrungen mit dem Ü5 2/3

UNI FREIBURG

- Lebenseinstellung in einem zellulären Automaten
  - Ich hab Angst vor Glidern
  - Wille Teil des Automaten, aber wir wissen es nicht
  - Kein schöner Gedanke, aber gute Begründung für Faulheit
  - Kann man dann nicht für seine Taten bestraft werden?
  - Würde Profibankräuber werden, da ich für mein Handeln ja keine Schuld trage, dürfte mich niemand verurteilen.
  - Superposition der Quantenobjekte evtl. nur Rechenleistungs-sparmaßnahme der Simulation, in der wir leben
  - Sehr komischer Gedanke, determiniert zu sein
  - Alles OK … kenne nur meinen Nutzen dabei nicht

Erfahrungen mit dem Ü5

Dazu gibt es eine perfekte
Szene in Westworld S01E06

- Wie passt Determinismus und freier Wille zusammen?
  - Experiment 1: man manipuliert ein Gehirn, so dass die Versuchsperson z.B. Ihren Arm bewegt und fragt sie hinterher, warum sie das gemacht hat

    Typische Antwort: "weil ich es wollte"
  - Experiment 2: man misst den Zeitpunkt der Entscheidung im Bewusstsein und im Unterbewusstsein

    Ergebnis: die Entscheidung wird erst im Unterbewusstsein getroffen, danach im Bewusstsein
  - Es spricht sehr viel dafür, dass unser Bewusstsein nicht die Entscheidungen trifft, sondern im Nachhinein eine zusammenhängende Geschichte bastelt von dem, was in uns und um uns herum passiert

Treffen mit Ihrem Tutor / Ihrer Tutorin

UNI FREIBURG

- 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, sonst bitte kurze Nachricht an Ihre/n Tutor/in
  - Alles Weitere in der Mail … Treffen dauert max. 30 Minuten

Dynamische Speicherallokation 1/6

UNI FREIBURG

- Statische Speicherallokation
  - Bisher wurde aller Speicherplatz bei den Deklaration allokiert, z.B.

```
int x;        // Reserves 4 bytes for an int.
int a[5];     // Reserves 20 bytes for array of five ints.
```

  - Das hier funktioniert **nicht**

```
int n = atoi(argv[1]);  // First command line argument.
int array[n];  // Will not compile, n must be a constant.
```

Dynamische Speicherallokation 2/6

UNI FREIBURG

- Dynamische Speicherallokation
  - Zur Laufzeit bekommt man Speicher mit new

```
int n = atoi(argv[1]);
int* array = new int[n];  // Array of n ints.
```

  - Das funktioniert auch mit Objekten

```
String* str = new String(); // Is a pointer!
```

Erzeugt ein String Objekt und ruft dabei den Default-Konstruktor auf… die Klammern sind dabei **Pflicht!**

```
String* strs = new String[n];  // Also a pointer!
```

Erzeugt ein Feld mit n String Objekten und ruft für jedes davon den Default-Konstruktor auf

Dynamische Speicherallokation 3/6

UNI FREIBURG

- Freigeben von dynamisch allokiertem Speicher
  - Bei einzelnen Objekten mit delete, z.B.

    delete str; // Free memory for single object.
  - Bei Feldern (von was auch immer) mit delete[], z.B.

    delete[] array; // Free memory for array of ints.

    delete[] strs; // Free memory for array of objects.
  - Danach darf man nicht mehr auf den Speicher, auf den diese Zeiger zeigen, zugreifen

    In **Java** wird Speicher von selber wieder frei gegeben, wenn er nicht mehr benötigt wird (garbage collection)

    In **C++** muss man sich selber drum kümmern, hat aber so auch Kontrolle über den genauen Zeitpunkt

Dynamische Speicherallokation 4/6

UNI FREIBURG

- Der -> Operator
  - Wenn man einen Zeiger auf ein Objekt hat

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

  - Geht der Zugriff auf die Members im Prinzip so

    ```
    (*str)._contents = "Doof";  // Access member variable.
    (*str).set("Doof");         // Access method.
    ```

  - Alternativ kann man dafür auch schreiben

    ```
    str->_contents = "Doof";   // Access member variable.
    str->set("Doof");          // Access method.
    ```

    Das bedeutet **exakt** dasselbe, es ist einfach nur lesbarer

Dynamische Speicherallokation 5/6

UNI FREIBURG

- valgrind
  - Mit **new** und **delete** kann folgendes schnell passieren:

    Man greift auf Speicher zu, den man noch nicht mit new alloziert oder schon wieder mit delete freigegeben hat

    Schwer zu finden, weil sich der Fehler oft nicht an der Stelle äußert, wo er passiert, sondern erst (viel) später

    Man hat Speicher, auf den man keinen Zugriff mehr hat, nicht mit delete freigegeben hat

    Das nennt man **Speicherleck** bzw. memory leak
  - Solche Fehler findet man gut mit **valgrind**

Dynamische Speicherallokation 6/6

UNI FREIBURG

- Benutzung von valgrind
  - Einfach vor das ausführbare Programm schreiben, z.B.

    valgrind ./StringMain
  - Das findet allerdings nur Zugriffsfehler auf Speicher, der dynamisch (mit new) alloziert wird, auf dem sog. **Heap**

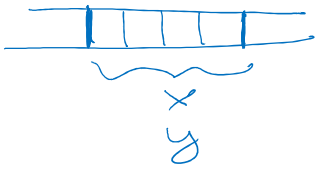
    Alle anderen Variablen liegen auf dem sog. **Stack**
  - Achten Sie auf die LEAK SUMMARY, insbesondere etwa:

    definitely lost: 4 bytes in 1 blocks
  - Details mit valgrind –leak-check=full ./StringMain

Übergabe von Argumenten 1/5

UNI FREIBURG

- Wiederholung Zeiger



  - In C++ wichtig zu verstehen:

    Wann hat man es mit dem **Wert** einer Variablen zu tun, und wann mir der **Adresse** dieser Variablen im Speicher

```
int x = 4;  // 4 is the value of x.

int* p = &x;  // p now holds the address of x in memory.

int* q = &x;  // Another pointer to the same address.
```

  - Man kann & auch noch so verwenden:

```
int& y = x;  // y is now an alias for x.

y = 5; printf("%d\n", x);  // Will print 5.
```

    Braucht man selten, aber gut um das Folgende zu verstehen

Übergabe von Argumenten 2/5

UNI FREIBURG

- Call by value

  int square(int z) { int res = z * z; return res; }

  – Wenn wir jetzt einen Aufruf haben

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

  – Dann passiert sinngemäß Folgendes

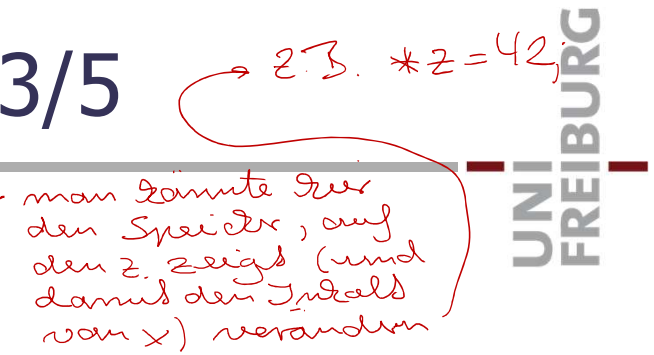
  int x = 5; { **int z = x**; int res = z * z; y = res; }

  – Der Wert der Variablen x wird in die für die Funktion lokale Variable z kopiert, das nennt man **call by value**

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

  Aber bei einem Objekt mit Hunderten oder Millionen von Bytes kostet Kopieren richtig Zeit

Übergabe von Argumenten 3/5



- Call by value, mit Zeigervariablen

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

  – Wenn wir jetzt einen Aufruf haben

  int x = 5; y = square(**&**x); // Need to pass int* now.

  – Dann passiert sinngemäß Folgendes

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

  – Die Speicheradresse der Variablen x wird in die lokale Zeigervariable z kopiert

  – Eine Speicheradresse hat 4 – 8 Bytes, bei großen Objekten ist das also viel effizienter als wie auf der Folie vorher

  Nachteil: man muss überall & und * schreiben

Übergabe von Argumenten 4/5

UNI FREIBURG

- Call by reference

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

  – Wenn wir jetzt einen Aufruf haben

  ```
  int x = 5;    y = square(x);  // No need for & here.
  ```

  – Dann passiert sinngemäß Folgendes

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

  – Die lokale Variable z ist jetzt ein Alias für x, das wird intern genauso realisiert wie mit den Zeigern auf der Folie vorher

  Das nennt man dann **call by reference**

  Vorteil: es wird beim Aufruf nur eine Adresse kopiert, aber man muss nur einmal & schreiben, bei der Deklaration

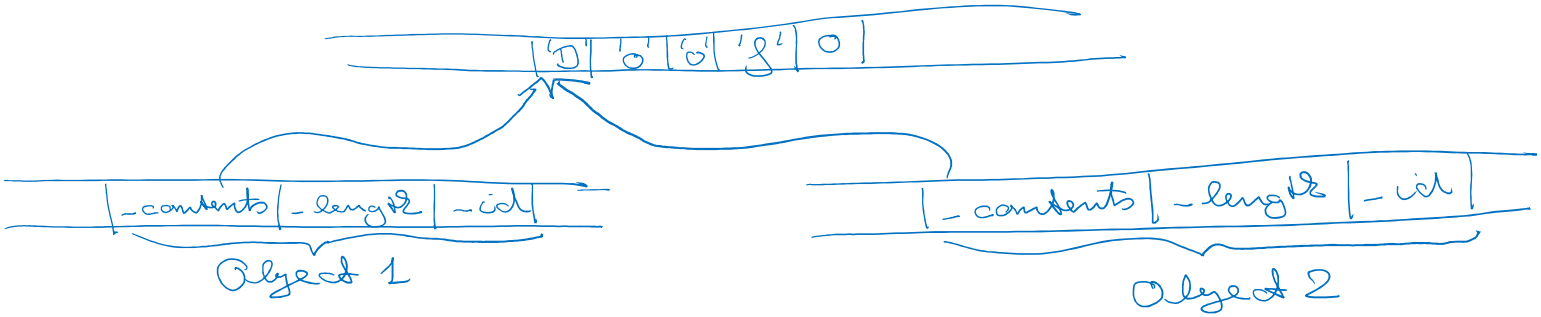
Übergabe von Argumenten 5/5

UNI FREIBURG

- Call by value, mit Objekten
  - Wie gesagt, wird ohne & das Objekt kopiert, z.B.

    void String::append(String s);

    Ohne Weiteres werden dabei aber einfach die Member-Variablen kopiert, hier ein char* und ein int, aber nicht, worauf der char* zeigt … das nennt man **shallow copy**



    Will man eine **deep copy**, muss man dafür eine extra Methode schreiben

    Faustregel: bei Objektargumenten fast immer **const … &**

Const 1/4

UNI FREIBURG

- Const bei Variablen
  - **const** bei einer Variablendeklaration heißt, dass man die Variable nach der Initialisierung nicht mehr ändern darf

```
const float pi = 3.1459;
pi = pi * 2;  // Will not compile.
```

  - Vor einem Funktionsargument heißt es entsprechende, dass man diese lokale Variable nicht ändern darf

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

Const 2/4

UNI FREIBURG

- Const bei Methoden
  - const bei einer Methodendeklaration heißt, dass diese Methode keine Membervariablen ändern darf

```
class String {
   int size() const;
   int _size;
}

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

  - Ändert man trotzdem etwas, meckert der Compiler:

```
error: assignment of member '...' in read-only object
```

Const 3/4

UNI FREIBURG

- Const-Correctness
  - Alles was von der Logik des Programms her nicht verändert werden darf, soll const deklariert werden

    Das hilft bei der Vermeidung von doofen Fehlern

    Es hilf auch beim Verständnis des Codes

    Außerdem kann es dem Compiler helfen, effizienteren Maschinencode zu erzeugen (weil er so weiß, welche Variablen sich mit Sicherheit nicht ändern)
  - Einzige Ausnahme ist call by value mit Basistypen wie int, float, double, …

    Da ist so was wie const int& x nur schwerer lesbar als einfach int x ohne etwas zu sparen oder viel zu helfen

Const 4/4

UNI FREIBURG

- Mutable
  - Manchmal hat man Methoden, die eigentlich const sind, außer dass sie einige "Hilfsvariablen", die eigentlich gar nichts mit der Klasse zu tun haben, ändern
  - Wenn man diese "Hilfsvariablen" mutable deklariert, kann die Methode trotzdem const sein

```
class String { … mutable _numCallsToSize; … }

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

  - Was dabei als "Hilfsvariable" zählt ist Sache des Programmierenden … man sollte halt den Sinn verstanden haben

Rückgabe von Objekten

Das ruft den sogenannten **Copy-Constructor** auf, siehe Code aus der Vorlesung

- Möglichkeit 1: "Normale" Rückgabe
  - Funktion

    ```
    String doof() { String r; r.set("Doof"); return r; }
    ```

  - Aufruf

    ```
    String x = doof();  // Same as String x(doof());
    ```

  - Nachteil: das Objekt wird bei der Rückgabe eventuell **kopiert** … das wäre sehr ineffizient bei großen Objekten
  - Allerdings: die meisten Compiler erzeugen Code, der eine solche Kopie vermeidet, diese Optimierung nennt sich **copy elision** oder **return value optimization**

    Klappt aber nicht immer, z.B. bei: return … ? r1 : r2

    Explizit ausschalten mit g++ -fno-elide-constructors

Rückgabe von Objekten 2/6

UNI FREIBURG

- Möglichkeit 2: Rückgabe einer Referenz
  - Funktion

    String& doof() { String r; r.set("Doof"); return r; }
  - Das gibt Mecker vom Kompiler:

    warning: reference to local variable 'r' returned

    Einen Zeiger auf etwas zurückzugeben, was dann nicht mehr existiert, ist eine **sehr** schlechte Idee

Rückgabe von Objekten 3/6

UNI FREIBURG

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

    String* doof() { String r; r.set("Doof"); return **&r**; }
  - Compiler meckert schon wieder

    warning: address of local variable 'r' returned

    Und zwar zurecht und aus demselben Grund wie bei Möglichkeit 2

Rückgabe von Objekten 4/6

UNI FREIBURG

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

```
String* doof() {
   String* r = new String();
   r->set("Doof"); return r;
}
```

  - Das kompiliert und funktioniert auch
  - Aber sollte man trotzdem nicht tun: in der Funktion wird Speicher alloziert, der außen freigegeben werden muss

    Die Gefahr ist groß, dass man das vergisst … oder den Speicher doppelt freigibt

Rückgabe von Objekten 5/6

UNI FREIBURG

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

    ```
    void doof(String& r) { r.set("doof"); }
    ```

  - Aufruf

    ```
    String x; doof(x);
    ```

  - Das kompiliert ... aber gefällt cpplint.py / checkstyle nicht:

    Man sieht dem Aufruf nicht an, dass r verändert wird

Rückgabe von Objekten 6/6

UNI FREIBURG

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

    ```
    void doof(String* r) { r->set("doof"); }
    ```

  - Aufruf

    ```
    String x; doof(&x);
    ```

  - So hätte cpplint.py / checkstyle das gerne!

    Das & vor dem x macht deutlich, dass das Objekt in der Funktion verändert werden kann

    Außerdem kann man so (anders als bei Möglichkeit 1) auch leicht mehrere Objekte übergeben (auch bei Variante A)

Literatur / Links

UNI FREIBURG

- New und delete
  - http://www.cplusplus.com/doc/tutorial/dynamic
- Wiederholung Zeiger
  - http://www.cplusplus.com/doc/tutorial/pointers
- Call by value / call by reference
  - http://www.cplusplus.com/doc/tutorial/functions2
- Const correctness
  - http://en.wikipedia.org/wiki/Const-correctness