# Programmieren in C++

## SS 2022

## Vorlesung 6, Dienstag 31. Mai 2022

(Dynamische Speicherallokation, Zeiger und Referenzen, Const-Korrektheit, Copy-Konstruktor und Copy-Zuweisung)

Prof. Dr. Hannah Bast
Professur 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 mit Klassen
  - Zeit für die … — Vorlesungen und Übungen
- Inhalt
  - Dynamische Allokation — new und delete
  - Zeiger und Referenzen — Mehr zu * und &
  - Const-Correctness — wo schreibt man überall const?
  - Copy-Konstruktor — und Copy-Zuweisung

**Ü6:** Klassen "String" und "StringSorter" implementieren, so dass die vorgegebenen Tests fehlerfrei durchlaufen

# Erfahrungen mit dem Ü5 1/3

UNI FREIBURG

- Feedback zur Aufgabe
  - Die meisten fanden die Aufgabe lehrreich und gut machbar

    "Weniger Aufwand als beim Ü4"

    "Aufteilung in Klassen hat Spaß gemacht. Tolle Übung. Nutzen und Sinn wurden in der Vorlesung super erklärt"

    "Inkrementelles Vorgehen war hier sehr hilfreich"

    "Hat sich zwar nicht ganz so produktiv angefühlt, dafür ist jetzt alles ein bisschen ordentlicher"

    "Lief eigentlich halbwegs gut, aber ich bin ultra-langsam"
  - Wunsch nach einer weniger langen Vorlesung

    Ja, selbst das abgespeckte Snake war noch zu umfangreich, beim nächsten Mal noch weiter vereinfachen

# Erfahrungen mit dem Ü5 2/3

UNI FREIBURG

- Game Of Life, Demo
  - Game of Life ist **Turing vollständig** = alles was man überhaupt berechnen kann, geht auch in Game Of Life
  - In der Tat kann man im Game Of Life zahlreiche (teilweise sehr) komplexe Strukturen bauen, zum Beispiel:

    Space X

    Abgefahren

    Tetris

    Computer mit Display
  - Und natürlich komplexe Lebewesen mit Selbstbewusstsein, so wie wir, siehe Ü5 Aufgabe 3

# Erfahrungen mit dem Ü5 3/3

UNI FREIBURG

- Wir sind nur eine Figur in einem zellulären Automaten … und?

  "Ich bin froh, dass mein zellulärer Automat 3D ist"

  "Hat keinerlei Auswirkungen auf mein Leben"

  "Nicht 'nur' eine Figur, sondern Bewusstsein, das über die Summe der Zellen der Figur entsteht"

  "Habe mich dahin entwickelt, die meisten Dinge hinzunehmen und darüber hinweg zu leben. Ändern kann ich daran sowieso nichts."

  "Schritt 278128: Der Tag der Erkenntnis. Die Erkenntnis ist beeng-end und macht doch frei. […] Es bleibt nichts außer der Hoffnung, dass jemand den Ausgangszustand mit Bedacht gewählt hat."

  "Dass unendliches Wachstum doch existiert ist be(un)ruhigend."

  "Ab jetzt laufe ich nur noch zu viert mit Leuten in Quadratform rum."

# Zeit 1/2

UNI FREIBURG

- Umfang der Übungsblätter
  - Das Ü3 und Ü4 waren etwas zu umfangreich
  - Wir werden das bei der nächsten Ausgabe dieser Vorlesung, im SS 2024, reduzieren, insbesondere für das Ü3
  - Damit Sie auch etwas davon haben, werden wir es aber auch in diesem Semester noch ausgleichen, indem wir:

    1. Bei den nächsten Übungsblättern verstärkt auf den Umfang achten, insbesondere schon beim Ü6 und Ü7
    2. Umfang des Projekts am Ende eher weniger als mehr
    3. Eine zusätzliche Vorlesungswoche frei, voraussichtlich die Woche mit Dienstag, dem 12. Juli

    Sie haben dann sowohl für das Ü6 als auch für das Ü10 jeweils zwei Wochen Zeit

# Zeit 2/2

UNI FREIBURG

- Vorlesungszeit
  - Ziel sind ≤ 1 h 45 min, die letzten beiden Vorlesungen haben aber jeweils ca. 2 Stunden gedauert

    Das empfinden viele als zu lang und ich auch, wobei es mit zwei Pausen ganz gut erträglich ist … aber letztes Mal waren selbst 2 h zu wenig, weil das Programm so umfangreich war
  - Ich tue, was ich kann, es ist aber nicht so einfach, die vielen Anforderungen und Wünsche unter einen Hut zu bringen:

    Organisatorisches, ausführliche Erklärungen, live vormachen, interessanter Stoff, auf Fragen eingehen, Pausen, nicht hetzen, sehr viele Fragen in den erfahrungen.txt, ob man xyz nicht noch mal genauer erklären könnte, …

    Alternative: ein zweiter Termin, aber das will niemand wirklich

# Dynamische Speicherallokation 1/7

UNI FREIBURG

- Statische Speicherallokation
  - Bisher wurde aller Speicherplatz bei der 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 2/7

UNI FREIBURG

- Dynamische Speicherallokation
  - Zur Laufzeit bekommt man Speicher mit new

    char* bytes = **new** char[n]; // Pointer to n bytes.
  - Das funktioniert auch mit Objekten

    String* str = **new** String(); // Pointer to String object.

    Erzeugt ein Objekt vom Typ String und ruft dessen Default-Konstruktor auf … die () kann man auch weglassen

    String* strs = **new** String[n]; // Pointer to String object**s**.

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

# Dynamische Speicherallokation 3/7

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[] bytes;  // 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
  - Typische Fehlermeldungen in diesem Zusammenhang:

    ```
    double free or corruption
    free(): invalid pointer
    segmentation fault
    ```

# Dynamische Speicherallokation 4/7

UNI FREIBURG

- Speichermanagement in anderen Sprachen
  - In Sprachen wie **Java**, **C#** oder **Python** wird Speicher nach einer Weile "von selber" wieder frei gegeben, wenn er nicht mehr benötigt wird (garbage collection)

    Das ist einerseits komfortabel, bringt aber in der Praxis auch einige Problem mit sich, insbesondere:

    Speicherverbrauch schwer vorherzusagen und zu messen

    Laufzeit einzelner Codeteile schwer zu messen, wenn an unvorhersagbaren Stellen die garbage collection läuft

  - In **C++** muss man sich selber darum kümmern, hat aber so auch die volle Kontrolle über Zeitpunkt und Umfang

    https://www.youtube.com/watch?v=s-LDAr5Y5SQ

# Dynamische Speicherallokation 5/7

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 6/7

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 7/7

UNI FREIBURG

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

    valgrind ./SomeDataMain
  - Das findet vor allem 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:

    definitely lost: 100 bytes in 1 blocks
  - Details mit valgrind –leak-check=full ./SomeDataMain
  - Wie beim Debuggen den Code mit –g kompilieren, dann bekommt man Hinweise mit Zeilennummern

# Wiederholung Zeiger 1/2

UNI FREIBURG

- Zur Erinnerung
  - In C++ ist es immer wichtig zu verstehen:

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

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

int* p = &x;     // The value of p is now the address of x.

int* q = &x;     // The same address, now assigned to q.
                 // We say: p and q point to the same address.
```

  - Eine mit * deklarierte Variable heißt **Zeiger**

# Wiederholung Zeiger 2/2

UNI FREIBURG

- Der & Operator
  - Auf den vorherigen Folien hat einem der & Operator die Speicheradresse einer Variablen gegeben

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

  - Das & hat in C++ aber auch noch eine andere Bedeutung:

    ```
    int& y = x;  // y is now another name (an "alias") for x.

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

  - Eine mit & deklarierte Variable heißt **Referenz**

    So wie in dem Beispiel oben benutzen wir das eher selten, aber bei der Übergabe von Argumenten an oder aus einer Funktion werden wir das häufig sehen

    Siehe Folien 24 und 27 und die nächste Vorlesung

# Const 1/6

UNI FREIBURG

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

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

  - Vor einem Funktionsargument heißt es entsprechend, 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 2/6

- Const bei Zeigern
  - **const** bei einer Zeigerdeklaration heißt, dass man den Inhalt nicht verändern darf, die Adresse aber schon

```
const char* p = "Doof";
*p = 'B';          // Will not compile, neither will p[i] = …
p = "Bloed";       // This is fine.
```

  - Für den sehr seltenen Fall, dass man es umgekehrt haben möchte, schreibt man das const nach dem *

```
char s[5] = { 'D', 'o', 'o', 'f' , 0 };
char * const p = s;
*p = 'B';          // This is now allowed.
p = p + 1;         // But this is not.
```

  - Es geht auch beides: `const char* const doof = "Doof";`

# Const 3/6

UNI FREIBURG

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

```
class SomeData {
   size_t size() const;
   size_t size_;
}

size_t SomeData::size() const { return size_; }
```

  - Ändert man trotzdem etwas, meckert der Compiler:

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

# Const 4/6

UNI FREIBURG

- Constexpr
  - Zur Erinnerung, auf Folie 17 stand:

    **const** bei einer Variablendeklaration heißt, dass man die Variable nach der initialen Zuweisung nicht mehr ändern darf
  - Seit C++11 gibt es auch **constexpr**

    **constexpr** heißt, dass der Wert des Ausdrucks bereits zur Zeit des Kompilierens vollständig feststeht
  - Brauchen wir bis auf Weiteres nicht, Details siehe

    https://en.cppreference.com/w/cpp/language/constexpr

# Const 5/6

UNI FREIBURG

- Const-Korrektheit
  - Ein Programm ist **const-korrekt**, wenn …

    1. Alle Variablen, die nach der initialen Zuweisung nicht mehr verändert werden "sollten", als **const** deklariert sind

    2. Alle Memberfunktionen, die das Objekt der Klasse nicht verändern "sollten", als **const** deklariert sind

    3. Alle Rückgabewerte, die der aufrufende Code nicht verändern können "sollte", als **const** deklariert sind

  - Das "sollten" bezieht sich dabei auf den **Sinn** des Programms
  - Ausnahme: Basistypen wie int, float, double, … müssen in Funktionsargumenten nicht const deklariert werden, siehe V7

    ```
    int square(int x) { return x * x; }   // OK without const.
    ```

# Const 6/6

UNI FREIBURG

- Mutable
  - Manchmal hat man Methoden, die eigentlich const sind, außer dass sie einige "Hilfsvariablen" der Klasse ändern

    Selten, aber trotzdem manchmal nützlich
  - 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 wieder Sache des Programmierenden … es sollte halt **Sinn** ergeben

# Copy-Konstruktor 1/7

UNI FREIBURG

- Spezielle Memberfunktionen
  - Jede Klasse hat **sechs** Memberfunktionen, die implizit da sind, auch wenn man Sie nicht explizit implementiert:

| | |
|---|---|
| Default-Konstruktor | siehe letzte Vorlesung (V5) |
| Destruktor | siehe letzte Vorlesung (V5) |
| Copy-Konstruktor | nächste Folien |
| Copy-Zuweisungsoperator | nächste Folien |
| Move-Konstruktor | siehe nächste Vorlesung (V7) |
| Move-Zuweisungsoperator | siehe nächste Vorlesung (V7) |

Für das Ü6 müssen Sie die ersten vier implementieren

# Copy-Konstruktor 2/7

UNI FREIBURG

- Copy-Konstruktor
  - Der **Copy-Konstruktor** heißt wie die Klasse und hat keinen Rückgabewert (so wie alle anderen Konstruktoren auch)
  - Er hat genau ein Argument, und zwar eine const Referenz auf ein Objekt der Klasse

    SomeData::SomeData(const String& s) { … }
  - Wenn man ihn nicht explizit implementiert, gibt es einen impliziten Copy-Konstruktor, der Folgendes macht:

    Er kopiert alle Membervariablen aus s

    Für jede Membervariable, die ein Objekt ist (also keinen Basistyp wie int, double, … hat), wird dabei der Copy-Konstruktor des betreffenden Objekts aufgerufen

# Copy-Konstruktor 3/7

UNI FREIBURG

- Copy-Konstruktor, Achtung
  - In unserer Klasse SomeData und in der Klasse String vom Ü6 haben wir einen Zeiger als Membervariable und dieser Zeiger zeigt auf Speicher, den wir mit new selber alloziert haben
  - Wenn wir beim Kopieren einfach nur die Adresse des Zeigers kopieren, haben wir hinterher zwei Objekte, deren Zeiger auf dieselbe Adresse zeigen

    Das ist böse und geht insbesondere dann schief, wenn nacheinander auf beiden Objekten der Destruktor aufgerufen wird ("double free", siehe Folie 10)

    Wir müssen darauf achten, in diesem Fall eine sogenannte "tiefe Kopie" aka "deep copy" zu erzeugen

# Copy-Konstruktor 4/7

UNI FREIBURG

- Copy-Konstruktor, Aufruf
  - Der Copy-Konstruktor wird immer dann aufgerufen, wenn es das Objekt, in das kopiert wird, vorher noch nicht gab:

```
String s2(s1);          // Calls the copy constructor on
                        // s2 with argument s1

String s2 = s1;         // Same as before, despite the =
```

# Copy-Konstruktor 5/7

UNI FREIBURG

- Copy-Zuweisungsoperator
  - Wenn man in ein Objekt kopieren will, das es vorher schon gab, braucht man den **Copy-Zuweisungsoperator**

```
SomeData s1(…);          // Create object s1.
SomeData s2(…);          // Create object s2.

s2 = s1;                 // Does not call the copy constructor,
                         // but the assignment operator.
```

  - Man definiert ihn wie folgt:

```
SomeData& operator=(const SomeData& s) { … }
```

Die Implementierung ist in der Regel sehr ähnlich zu der des Copy-Konstruktors, außer dass man sich noch angemessen um den bisherigen Inhalt des Objekts "kümmern" muss

# Copy-Konstruktor 6/7

UNI FREIBURG

- "Rule of Three"
  - Wenn man **eine** der folgenden drei speziellen Memberfunktionen implementiert, sollte man **alle** drei davon implementieren

    Default-Konstruktor
    Copy-Konstruktor
    Copy-Zuweisungsoperator
  - Grund: Wenn man das nicht tut, tun die implizit vorhandenen Varianten dieser Funktionen vermutlich nicht, was man möchte

    Zum Beispiel, wenn der Copy-Konstruktor Speicher alloziert und der implizite Destruktor diesen nicht wieder freigibt

    In Vorlesung 7 sehen wir dann die zugehörige "Rule of Five"

# Copy-Konstruktor 7/7

UNI FREIBURG

- explicit
  - Bei einem "normalen" Konstruktor (nicht Copy oder Move) mit einem Argument möchte cpplint.py gerne **explicit** davor
  - Der Grund ist die automatische Typkonversion von C++

```
// Constructor with one argument, without "explicit".
SomeData::SomeData(int size) { … }

// Function that takes object of type SomeData as argument.
someFunction(SomeData data) { … };

// Call someFunction with argument of type int.
someFunction(100);
```

Ohne "explicit" kompiliert das und der letzte Aufruf konvertiert über den o.g. Konstruktor 100 zu SomeData(100) … was oft nicht das ist, was man erwartet, sondern ein Versehen im Code

# Hinweise zum Ü6 1/2

UNI FREIBURG

- Sie sollen eine einfache String Klasse schreiben
  - Realisierung mit C-style Zeichenketten vom Typ const char*

    Ein Feld von **char**s mit 0 (nicht '0') am Ende "null-terminiert"
  - Sie dürfen #include <cstring> machen und die folgenden beiden Funktionen daraus benutzen, aber sonst keine:

    strlen(const char*) → zählt die Anzahl Zeichen bis zur 0

    strcmp(const char*, const char*) → vergleicht die beiden Zeichenketten und gibt 1 zurück, wenn die erste größer ist
  - Die vorgegebenen Tests benutzen teilweise ASSERT_STREQ → prüft die Gleichheit der Zeichenketten von zwei const char *

    Das Ü6 ist **sehr** nett, bitte diese Woche noch anfangen !!!

# Hinweise zum Ü6 2/2

UNI FREIBURG

- Der [] operator
  - Die Klasse StringSorter vom Ü6 enthält ein Feld von Strings
  - Für den Zugriff auf den i-ten String könnte man Funktionen get und set schreiben, wir wollen aber lieber sowas schreiben:

```
StringSorter strings(2);
strings[0] = "doof";
strings[1] = "bloed";
```

  - Dazu muss man den **Operator []** als Memberfunktion implementieren:

```
String& StringSorter::operator[](size_t i) { … }
```

Man beachte den Rückgabetyp String& : dadurch, dass es eine **Referenz** ist und **nicht const**, kann strings[0] oben auch auf der linken Seite einer Zuweisung stehen → mehr dazu in V7

# Literatur / Links

- New und delete
  - http://www.cplusplus.com/doc/tutorial/dynamic
- Zeiger nochmal
  - http://www.cplusplus.com/doc/tutorial/pointers
- Const-Korrektheit
  - http://en.wikipedia.org/wiki/Const-correctness
- Rule of Three
  - https://en.wikipedia.org/wiki/Rule_of_three_(C++_programming)