universität freiburg

Programmieren in C++
SS 2024

Vorlesung 7: Move-Konstruktor und Move-Zuweisung, Überladung, Funktionen: Argumentübergabe & Ergebnisrückgabe

4. Juni 2024

Prof. Dr. Hannah Bast
Institut für Informatik
Universität Freiburg

Blick über die Vorlesung heute

- Organisatorisches
  - Erfahrungen mit dem Ü6 — String und StringSorter
  - Treffen mit dem Tutor — Sie bekommen dazu Mail
- Inhalt
  - Funktionsüberladung — Funktionen mit gleichem Namen
  - Move-Operationen — Verschieben statt Kopieren
  - Übergabe von Argumenten — Pass by value/pointer/reference
  - Rückgabe von Objekten — Return by value/pointer/reference

Ü7: Move-Operationen für die String-Klasse, Sortieren mit move statt copy, Vergleich der Laufzeiten

Erfahrungen mit dem Ü6

- Zusammenfassung und Auszüge
  - In der Vorlesung sehr gut erklärt, der Ansatz mit gegebener Testdatei hat den meisten gefallen und war sehr gut machbar

    "Tolles Blatt, hat super viel Spaß gemacht, nicht zu lange gedauert und ich habe viel gelernt, 10/10"

    "Implementierung nach der Testdatei hat viel Spaß gemacht"

    "Sehr hilfreich zum Verständnis der Funktionen einer Klasse"

    "Es wird anspruchsvoller, aber trotzdem machbar"

    "This went so well, I feel like I belong in this class again!"

    "Fehlermeldungen des address sanitizers zum Teil zermürbend"

    "Garbage Collection ist schon deutlich bequemer"

    "Habe die Fragen zu Gen Z, freier Wille, etc. vermisst"

Funktionsüberladung 1/3

- Worum geht es
  - In C++ darf es mehrere Funktionen mit dem selben Namen geben, wenn sich ihre **Signatur** unterscheidet
  - Zur Signatur gehören insbesondere die Typen der Argumente:

```
String& operator=(const char* str);      // Assign C-style string.

String& operator=(const String& rhs);    // Assign String object.
```

  - Der Compiler versucht dann, anhand der Typen die richtige Funktion auszuwählen das klappt aber nicht immer, siehe Folie 6

```
String s1;
String s2;
s1 = "Doof";      // Calls the first function above.
s2 = s1;          // Calls the second function above.
```

Funktionsüberladung 2/3

- Die "constness" ist auch Teil der Signatur
  - Das hier zum Beispiel kompiliert:

```
char& operator[](size_t i) { return data_[i]; }
const char& operator[](size_t i) const { return data_[i] };
```

  - Der Compiler sucht die Funktion anhand der Constness des Objekts aus

```
ObjectWithAllocatedMemory o1("Doof");
const ObjectWithAllocatedMemory o2("Blöd");

printf("%c\n", o1[0]);     // Calls the first function above.
printf("%c\n", o2[0]);     // Calls the second function above.

o1[0] = 'd';               // Calls the first function above
o2[0] = 'b';               // Does not compile!
```

Funktionsüberladung 3/3

- Grenzen
  - Der Rückgabewert einer Funktion ist nicht Teil der Signatur; das hier zum Beispiel gibt einen Kompilerfehler:

```
float String:size() { … }
double String::size() { … }
```

  - Wenn der Compiler den Typ in einem kompatiblen Typ umwandeln muss und mehrere Typen "gleich gut" passen, gibt es ebenfalls einen Kompilierfehler, zum Beispiel:

```
float square(float x) { return x * x; }
double square(double x) { return x *x; }
...
printf("The square of 42 is %f", square(42));
```

Move-Operationen 1/8

- Die Operation new und delete sind relativ teuer
  - Wir messen die Zeit für die Erzeugung von n Objekten

```
size_t n = atoi(argv[1]);      \\ Number of objects.
size_t k = atoi(argv[2]);      \\ Number of bytes per object.
clock_t start = clock();
for (size_t i = 0; i < n; ++i) {
   ObjectWithDynamicAllocation(k);
}
clock_t end = clock();
double elapsed_seconds = (end – start) / CLOCKS_PER_SEC;
```

  - Beobachtung: selbst für k = 0 ist es relativ teuer (im Vergleich dazu, wenn die Objekte nichts allozieren)
  - Übrigens: delete nullptr ist erlaubt und macht einfach **nichts**

Move-Operationen 2/8

- Copy-Konstruktor und Copy-Zuweisung
  - Signaturen (Wiederholung aus Vorlesung 6):

    ```
    String(const String& other);                // Copy constructor.
    String& operator=(const String& other);     // Copy assignment.
    ```

  - Wie wir gerade gesehen haben, kann das teuer werden für Objekte, die dynamisch (viel) Speicher allozieren
  - Vorteil: man bekommt eine echte ("tiefe") Kopie

    Das Argument **other** wird dabei nicht verändert, wie man ja auch an dem **const** sieht

Move-Operationen 3/8

- Oft braucht man aber gar keine Kopie
  - Zum Beispiel beim Tauschen ("swap") zweier Objekte:

```
String s1;
String s2;
String tmp = s1;     // After this, s1's value does not matter.
s1 = s2;             // After this, s2's value does not matter.
s2 = tmp;            // After this, tmp's value does not matter.
```

Bei allen drei Zuweisungen ist uns der Wert der Variablen auf der rechten Seite nach der Zuweisung egal, weil er sowieso danach überschrieben oder gar nicht mehr gebraucht wird

Move-Operationen 4/8

- Move-Konstruktor und Move-Zuweisung
  - Signatur ähnlich zu copy, aber mit && statt & (siehe nächste Folie) und ohne const (weil das Argument verändert wird)

    ```
    String(String&& other);                 // Move constructor.
    String& operator=(String&& other);      // Move assignment.
    ```

  - Typischerweise implementiert man das so, dass sich das Objekt den Inhalt von other "klaut" und other danach leer ist (auf jeden Fall muss es danach noch ein gültiges Objekt sein)

    Das implementieren wir jetzt für ObjectWithAllocatedMemory

  - Das && ist eine sogenannte **rvalue reference**

    Wurde mit C++11 eingeführt und gibt es in C nicht; der Zweck ist gerade, um so etwas wie "move" leicht zu ermöglichen

Move-Operationen 5/8

- Wann werden diese Funktionen aufgerufen?
  - Anwendungsfall 1: wenn man explizit nach && "castet", dafür gibt es extra **std::move** (dafür braucht man: #include <utility>)

```
String s1;                  // Calls the constructor for s1.
String s2 = s1;             // Calls the copy constructor for s2.
String s3 = (String&&)s1;   // Calls the move constructor for s3.
String s4 = std::move(s1);  // Just the same, with nicer syntax.
```

Wichtig zum Verständnis: das std::move selber macht nichts, es sorgt nur dafür, dass der Move-Konstruktor aufgerufen wird

  - Anwendungsfall 2: bei temporären Objekten, das hat man insbesondere bei der Übergabe von Argumenten (Folien 17+20)

```
String s5 = String("doof");     // Calls move constructor for s5
                                //  (and normal constructor for temporary object)
```

Move-Operationen 6/8

- Const und move passen nicht zusammen
  - Wenn das Objekt, aus dem heraus "gemoved" werden soll, const ist, wird stattdessen eine Kopie gemacht

```
const String s1("doof");
String s2 = std::move(s1);  // Calls the copy constructor
                            // because s1 is const.
```

Der Compiler sollte einen warnen, wenn man so etwas schreibt, tut er aber leider nicht

Es gibt aber spezielle "static analysis" Tools, die so etwas erkennen, und sollte man bei größeren Projekten sowieso verwenden, zum Beispiel:

```
clang-tidy-14 *.cpp –checks=performance-move-* --
```

Move-Operationen 7/8

- Implizite Move-Funktionen (analog zu Copy aus Vorlesung 6)
  - Wenn man sie nicht explizit selber schreibt, und auch keinen Destruktor und keine der copy-Funktionen, dann gibt es einen impliziten move constructor und move assignment operator
  - Die impliziten Funktionen machen Folgendes:
    - Membervariablen mit Basistypen werden einfach kopiert
    - Für Membervariablen mit Klassentypen wird der jeweilige move constructor bzw. move assignment operator aufgerufen

  Achtung: für Objekte, die selber low-level Speicher allozieren, ist das in der Regel **nicht**, was man möchte

Move-Operationen 8/8

- "Rule of Five" analog zur "Rule of Three" aus der V6
  - Regel: Wenn man **eine** der folgenden Memberfunktionen implementiert, sollte man **alle** fünf davon implementieren

    Destruktor
    Copy-Konstruktor
    Copy-Zuweisungsoperator
    Move-Konstruktor
    Move-Zuweisungsoperator
  - Wenn man das nicht tut, tun die impliziten Varianten dieser Funktionen vermutlich nicht das, was man möchte

    Siehe vorherige Folie
  - In Vorlesung 9 (STL) sehen wir dann noch die "Rule of Zero"

Übergabe von Argumenten 1/7

- Wiederholung Zeiger und Referenzen
  - Wir haben bisher die folgenden drei Möglichkeiten gesehen, mit einer Variable auf ein Objekt zu verweisen

```
String s = "doof";      // An object that owns it memory.
String* p = &s;         // A pointer to an object.
String& ref = s;        // A reference to (alias for) an object.
```

  - Alle drei gibt es auch mit const:

```
const String s = "doof";   // Value of s can't be changed anymore.
const String* p = &s;      // We can change p, but not *p.
const String& ref = s;     // We can neither change that ref is an
                           // alias for s, nor its value.
```

Auf den nächsten Folien schauen wir uns an, was diese Varianten bei der Übergabe von Argumenten an eine Funktion bedeuten

Übergabe von Argumenten 2/7

- Pass **by value** (mit copy)
  - Die Funktion benutzt ein neues Objekt, in das **kopiert** wird

    void doSomething(**String** arg) { … do sth with arg … }
  - Ein Aufruf der Funktion sieht dann so aus

    String s("doof");
    doSomething(s);

    und dabei passiert in der Funktion sinngemäß Folgendes

    String arg = s;
    … do something with arg …
  - Anwendungsfall: Wenn man das Argument im aufrufenden Code weiterhin braucht und es in der Funktion verändern möchte, es im aufrufenden Code aber **nicht** verändert werden soll

Übergabe von Argumenten 3/7

- Pass **by value** (mit move)
  - Die Funktion benutzt ein neues Objekt, in das **gemoved** wird

    void doSomething(**String** arg) { … do sth with arg … }
  - Ein Aufruf der Funktion sieht dann so aus

    String s("doof");
    doSomething(std::move(s));

    und dabei passiert in der Funktion sinngemäß Folgendes

    String arg = std::move(s);
    … do something with arg …
  - Anwendungsfall: Wenn man das Argument in der aufrufenden Funktion nicht mehr braucht + es spart die Kosten einer Kopie

Übergabe von Argumenten 4/7

- Pass **by pointer**
  - Es wird ein Zeiger auf das Objekt an die Funktion übergeben

    void doSomething(**String*** arg) { … do sth with arg or *arg … }
  - Ein Aufruf der Funktion sieht dann so aus

    String s("doof");
    doSomethingf(**&**s);

    und dabei passiert in der Funktion sinngemäß Folgendes

    String* arg = &s;
    … do sth with arg or *arg …
  - Anwendungsfall: Wenn man das Objekt im aufrufenden Code verändern möchte, ohne die Kosten einer Kopie des Objektes

    Stattdessen nur Kopie eines Zeigers (in der Regel 4 – 8 Bytes)

Übergabe von Argumenten 5/7

- Pass **by reference**
  - Ähnlich wie "pass by pointer", aber mit anderer Syntax

    ```
    void doSomething(String& arg) { … do sth with arg … }
    ```

  - Ein Aufruf der Funktion sieht dann so aus

    ```
    String s("doof");
    doSomething(s);
    ```

    und dabei passiert in der Funktion sinngemäß Folgendes

    ```
    String& arg = s;
    … do sth with arg …
    ```

  - Anwendungsfall: Wie bei "pass by pointer", aber man muss beim Aufruf nicht & schreiben und in der Funktion nicht *

    Achtung: man sieht dem Aufruf nicht an, das **s** verändert wird

Übergabe von Argumenten 6/7

- Pass **by rvalue reference**
  - Ähnlich wie Folie 17, aber es **MUSS gemoved** werden

    void doSomething(**String&&** arg) { … do sth with arg … }
  - Ein Aufruf der Funktion sieht dann so aus

    String s("doof");
    doSomething(std::move(s));

    und dabei passiert in der Funktion sinngemäß Folgendes

    String arg = std::move(s);
    … do sth with arg …
  - Anwendungsfall (selten): Wie auf Folie 17 (Objekt wird in der aufrufenden Funktion nicht mehr gebraucht + keine Kopie) und man kann die Funktion **nur** so benutzen (nicht auch mit Kopie)

Übergabe von Argumenten 7/7

- Const oder nicht const, das ist hier die Frage
  - Bei allen Aufrufen kann man auch const davor schreiben, je nach "pass by …" Art ist das aber mehr oder weniger nützlich

    Const value: doSomething(const String arg) { … }

    Selten: wenn schon Kopie, kann man sie auch verändern

    Const pointer: doSomething(const String* arg) { … }

    Häufig: die Funktion kann das Objekt lesen, aber nicht ändern

    Const reference: doSomething(const String& arg) { … }

    Häufig: die Funktion kann das Objekt lesen, aber nicht ändern

    Const rvalue ref: doSomething(const String&& arg) { … }

    Nie: wenn man dem aufrufenden Code schon das Objekt klaut, will man dann auch was damit machen

Rückgabe von Objekten 1/6

- Return **by value**
  - Wir geben ein neues Objekt zurück

    ```
    String computeString() { String r; … ; return r; }
    ```

  - Ein Aufruf der Funktion sieht dann so aus

    ```
    String result = computeString();
    ```

    und dabei passiert **sinngemäß** (nicht wirklich) Folgendes:

    ```
    String r;                          // Constructor called.
    ….
    String returnValue = r;            // Copy constructor called.
    String result = returnValue;       // Copy constructor called.
    ```

    Wird hier wirklich dreimal ein Konstruktor aufgerufen?

Rückgabe von Objekten 2/6

- Return **by value**, "copy elision"
  - In der Regel erkennt der Compiler, dass es hier nur mit einem Konstruktor geht, es wird dann für

    ```
    String computeString() { String r; … something … ; return r; }
    String result = computeString();
    ```

    effektiv ausgeführt

    ```
    String result; ... something … ;
    ```

  - Die entsprechenden Compileroptimierung heißt **copy elision**

    Vernünftige Kompiler haben das schon immer gemacht, seit C++17 schreibt es der Standard für bestimmte Fälle vor

  - Wir können sie mit –fno-elide-constructors ausschalten und dann sehen wir tatsächlich **drei** Aufrufe eines Konstruktors

Rückgabe von Objekten 3/6

- Return **by value**, Sonderfälle
  - Beobachtung: Mit –fno-elide-constructors wird der Move-Konstruktor benutzt, wenn wir ihn explizit definiert haben, sonst der Copy-Konstruktor
  - Wir können uns auf copy elision verlassen, wenn in der Funktion ein lokales Objekt erzeugt und mit return zurückgegeben wird

    Wie auf den Folien 22+23 … das ist auch der häufigste use case
  - In allen anderen Fällen brauchen wir ein explizites std::move, um eine unnötige Kopie zu vermeiden, zum Beispiel:

```
String s1, s2;
…
return drand48() < 0.5 ? std::move(s1) : std::move(s2);
```

Rückgabe von Objekten 4/6

- Return **by reference**
  - Das benutzt man of als getter/setter in Klassen

    ```
    String& StringSorter::operator[](size_t i) { return data_[i]; }
    ```

    Damit kann man dann im aufrufende Code z.B. schreiben

    ```
    StringSorter strings(10);        // Create array of ten strings.
    strings[0] = "doof";             // Access particular string.
    ```

  - **Wichtig:** das Objekt, auf das man eine Referenz zurückgibt, muss auch nach Ende des Funktionsaufrufes noch existieren

    ```
    String& doof() { String result("Doof"); return result; }
    ```

    warning: reference to stack memory associated with local variable 'result' returned

Rückgabe von Objekten 5/6

- Return **by pointer**
  - Analog zu "Return **by reference**", aber braucht man selten

    Man sieht es manchmal in älterem C++-Code oder wenn man mit C-Bibliotheken interagieren möchte oder muss

    Außerdem in bestimmte Fällen bei "Vererbung" (Vorlesung 10)

    Wie bei "Pass by value" muss man dann an mehr Stellen * schreiben, was nicht schön ist, aber dafür sieht man dem Code direkter an, dass Speicher verändert wird (siehe Folie 18+19)

Rückgabe von Objekten 6/6

- Return **via argument**
  - Man übergibt ein Argument (als Zeiger oder als Referenz), in das dann das Ergebnis geschrieben wird, zum Beispiel:

```
void computeString(String* result) {
   ... directly manipulate *result ...
}
```

  - Anwendungsfälle:
    - Typisch bei C-Bibliotheken (der Rückgabewert ist dann meistens der Fehlercode)
    - Wenn man mehrere Objekte zurückgeben möchte (und es nicht sinnvoll ist, die in einer Klasse zusammenzufassen)
    - Wenn das Argument sowohl Eingabe ist, als auch verändert wird (in-out parameter bzw. in-out argument)

Hinweise zum Ü7

- Machen Sie das Sortieren vom Ü6 effizienter
  - Zur Erinnerung: die Zeit beim Sortieren geht vor allem dafür drauf, Elemente zu vergleichen und dann ggf. zu tauschen

    Bubble Sort hat nicht nur quadratisch viele Vergleiche, sondern im schlechtesten Fall auch quadratisch viele Vertauschungen (deswegen haben wir das für das Ü6 genommen)
  - Für das Ü6 haben Sie Objekte (bewusst oder unbewusst) mittels "Copy" getauscht
  - Für das Ü7 sollen Sie auch die Move-Funktionen schreiben und die Objekte (bewusst) mittels "Move" tauschen + den Laufzeitunterschied zu "Copy" messen

    Zur Zeitmessung, siehe Vorlesungscode und Folie 7

    Für Zufallszahlen, siehe z.B. lrand48() dafür: #include <cstdlib>

Literatur / Links

- Move-Funktionen

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

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

- Copy-Elision und impliziter Move

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

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