Programmieren in C++

SS 2020

Vorlesung 7, Dienstag 14. Juni 2022

(Move-Semantik, Überladung, Funktionen: Argumentübergabe & Ergebnisrückgabe)

Johannes Kalmbach
Professur für Algorithmen und Datenstrukturen
Institut für Informatik, Universität Freiburg

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Erfahrungen mit dem Ü6 — String und StringSorter
- Inhalt
  - Move-Konstruktor und -Assignment
  - Überladene Funktionen
  - Übergabe von Argumenten — call by value / reference
  - Rückgabe von Objekten — copy elision / impliziter Move

Ü7: Move-Operationen für die String-Klasse, Performance-Vergleich von Copy- und Move-Operationen.

Erfahrungen mit dem Ü6 1/2

UNI FREIBURG

- Feedback zur Aufgabe
  - Aufgabe wurde größtenteils in einem Zug erledigt. Wortwitz beiseite (der Urlaub war apropos toll, ein Hoch auf das 9 Euro Ticket)
  - Das Blatt war nicht einfach, aber machbar. […] Existierende Forum-Beiträge haben […] aber sehr geholfen.
  - Eine schöne Aufgabe zum Üben. Die Null-terminated Strings haben mich etwas Zeit gekostet (…)
  - Fand es mit vorgegebenen Tests sehr viel angenehmer zu arbeiten
  - Bevor ich einschlafe höre ich: “invalid read segmention fault (core dumbed)”
  - Die strcmp()-Erklärung im Skript war falsch. Da haben Sie Recht, es ist >=1 und nicht ==1, das tut uns Leid.

Erfahrungen mit dem Ü6 1/2

UNI FREIBURG

- Feedback zur Aufgabe
  - Ich finde es nach wie vor eher schwer aus den Fehlermeldungen schlau zu werden (z.B. hat Valgrind 115 Errors ausgespuckt, weil ein einzelnes & gefehlt hat)
  - Der Stackoverflow-Hivemind mein, dass ich sowieso new and delete nicht verwenden sollte, sondern besser mit RAII oder so arbeiten sollte. (Recht haben die☺)
  - Dass man mich an “etwas ist faul” verzweifeln lassen hat, war schon bisschen gemein :D
  - Das Arbeitsblatt sollte ja eigentlich nett sein, aber für mich war es größtenteils frustrierend
  - Ich habe mir dieses mal nicht viel vorgenommen, habe dementsprechend aber auch nicht viel geschafft.
  - [Valgrind] zeig mir zwar an, dass ich einen memory leak habe, aber nicht wo genau der entsteht.

Dynamische Speicherallokation

UNI FREIBURG

- New / Delete sind teure Operationen

– Im Wesentlichen wird das Betriebssystem um einen Block Speicher gebeten.

– Sollte nicht unnötig gemacht werden (z.B. im Default-Konstruktor eines leeren Strings, siehe ML6 und Blatt 7)

– Wichtiger Trick: Den null-pointer darf man löschen (hat keinen Effekt, ist aber immer legal).

```
for (int i = 0; i < 1000; ++i) {
  delete nullptr;  // always legal
}
```

Funktionsüberladung 1/3

UNI FREIBURG

- Überladung mit unterschiedlichen Parametern
  - Gleicher Funktionsname, unterschiedliche Parameter:

```
String& operator=(const String& rhs);  // 1.

String& operator=(const char* str);  // 2.
```

  - Der Compiler sucht anhand der Parameter die passende Funktion aus:

```
String str1;
String str2;
str2 = str1; // Calls 1.
str = "Hallo"; // Calls 2.
```

Funktionsüberladung 2/3

UNI FREIBURG

- Überladung mit unterschiedlicher Constness
  - Gleicher Name + Parameter, unterschiedliche Constness: in der StringSorter-Klasse:

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

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

```
StringSorter sorter(1);
const StringSorter sorterConst(1);
// left: String&, right: const String&
sorter[0] = sorterConst[0];
// Error: writing to const String&:
sorterConst[0] = sorter[0];
```

Funktionsüberladung 3/3

UNI FREIBURG

- Mögliche Fehler
  - Funktionen, die sich nur im Rückgabewert unterscheiden sind verboten:

```
void a();
int a(); // Error
```

  - Wenn mehrere Funktionen gleich gut passen, gibt es auch einen Fehler (ambiguous overload)

```
void f(int x);
void f(char c);
// …
double d = 42.0;
f(d); // Fehler, beide Varianten passen gleich gut
```

Move-Operationen 1/6

UNI FREIBURG

- Wiederholung Copy-Konstruktor
  - Signaturen:

    String(const String& rhs); // Copy-Konstruktor

    String& operator=(const String& rhs); // Copy-Assignment
  - Teuer für Klassen mit dynamischem Speicher:
    Aufruf von new + Kopieren aller Unterobjekte
  - Das Originalobjekt rhs bleibt unverändert erhalten
    (rhs ist const !)

Move-Operationen 2/6

UNI FREIBURG

- Motivation
  - Häufiger Fall: Man will „kopieren“, braucht das Original hinterher aber nicht mehr, siehe z.B. swap:

```
String s1;
String s2;
String tmp = s1;  // s1 egal, wird gleich überschrieben
s1 = s2;  // s2 egal, wird gleich überschrieben
s2 = tmp;  // tmp egal, wird nicht mehr benutzt
```

  - Man könnte hier jeweils den dynamischen Speicher der rechten Seite “klauen”, genau das macht ein move in C++

Move-Operationen 3/6

UNI FREIBURG

- Move-Konstruktor und Move-Assignment
  - Signatur:
    String(String&& rhs); // Move-Konstruktor
    String& operator=(String&& rhs); // Move-Assignment
  - Das Objekt rhs muss nach der Operation immer noch gültig sein, darf aber einen anderen Wert als vorher haben. Typischerweise ist es dann leer.
  - Für Objekte mit dynamischem Speicher kann man moven oft effizienter implementieren als kopieren

Move-Operationen 4/6

UNI FREIBURG

- Wann werden Move-Operationen aufgerufen?

  String(String&& rhs);

  – String&& ist eine rvalue-Referenz (in etwa "Referenz auf ein Objekt, aus dem man moven darf")

  – Rvalue-Referenzen binden an:

  1. Temporäre Objekte ohne Namen
     ```
     String string1;
     // Move assignment, because rhs is temporary
     string1 = String("I will be moved");
     ```
  2. Explizit nach rvalue-Referenz gecastete Objekte:
     ```
     String string2((String&&) string1) // move constructor
     ```

Move-Operationen 5/6

UNI FREIBURG

- std::move
  - Statt dem expliziten Cast nach Typ&& schreibt man typischerweise:

```
#include <utility>  // for std::move
String s1("hallo");
String s2 = std::move(s1);  // same as (String&&)s1
```

  - Wichtig: std::move moved nichts, sondern erlaubt nur das moven mittels Move-Assignment oder –Konstruktor
  - Subtil: const + std::move ergibt eine Kopie

```
const String s3("hallo");
// Copy assignment!
String s4 = std::move(s3); // (const String&&) s3
```

Move-Operationen 6/6

UNI FREIBURG

- Rule of Five
  - Letze Woche: Rule of Three
    Destruktor, Copy-Konstruktor und Copy-Assignment sollte man immer alle drei implementieren, sonst ist meistens etwas verkehrt (siehe V5)
  - Dann sollte man oft auch aus Performance-Gründen Move-Konstruktor und Move-Assignment implementieren (Rule of Five)
  - Bei Klassen, die keinen dynamischen Speicher manuell mittels new/delete verwalten, reichen meistens die automatisch erzeugten Versionen (Rule of Zero, siehe kommende Vorlesungen)

Argumentübergabe 1/6

UNI FREIBURG

- Wiederholung Zeiger und Referenzen
  - Hier gibt es insgesamt nur ein String-Objekt:

```
String s("hallo");  // A string object that owns the bytes
String* sPtr = &s; // Pointer/address of s
const String* constSPtr = &s; // Pointer to const String.
String& sRef = s; // Reference/alias for s
const String& sConstRef = s; // Const
```

  - Frage: Wir schreiben eine Funktion, die einen String verarbeitet, welchen der obenstehenden Typen wählen wir für das Argument?

Übergabe von Argumenten 2/6

UNI FREIBURG

- Call by value

  ```
  void f(String arg) { /* do something with s */ }
  ```

  – Wenn wir jetzt einen Aufruf haben

  ```
  String s("hallo");    f(s);  // s is still the same!
  ```

  – Dann passiert sinngemäß Folgendes

  ```
  String s= 5;    {  String arg = s; /* do sth. With s */}
  ```

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

  – **Anwendung:** Entweder kopieren ist billig (z.B. bei Typ int) oder man braucht sowieso eine Kopie in der Funktion)

Übergabe von Argumenten 3/6

UNI FREIBURG

- Call by pointer

```
void f(String* sPtr) { /* do sth. with *sPtr;*/}
void fConst(const String* sPtr) { /* do sth. with *sPtr;*/}
```

  - Wenn wir jetzt einen Aufruf haben
    `String s("hallo");  f(&s);  // s might have changed, not const`
  - Dann passiert sinngemäß Folgendes
    `String s= 5;  { String* sPtr= &s; /* use *sPtr */}`
  - Die Speicheradresse der Variablen s wird in die lokale Zeigervariable sPtr kopiert
  - Eine Speicheradresse hat 4 – 8 Bytes, bei großen Objekten ist das also viel effizienter als call-by-value.

    Nachteil: man muss überall & und * schreiben (es sei denn, die Adresse ist das, was einen eigentlich interessiert)

Übergabe von Argumenten 4/6

UNI FREIBURG

- Call by reference

```
void f(String& arg) { /* use arg*/}
void fConst(const String& arg) { /* use arg*/}
```

  - Wenn wir jetzt einen Aufruf haben

```
String s("hallo");    f(s);  // s might have changed, not const
```

  - Dann passiert sinngemäß Folgendes

```
String s= 5;    {  String& arg = s; /* use arg */}
```

  - Die lokale Variable arg ist jetzt ein Alias für s, 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 5/6

UNI FREIBURG

- In-Parameter, Entscheidungshilfe
  - In-Parameter: Wird nur gelesen, nicht geändert:
    Call by value für kleine Typen: `void f(int i);`
    Call by value für große Typen, wenn man die Kopie braucht:
    `void f(String s);`
    Call by **const-**reference für große Typen, wenn man keine Kopie braucht:
    `void f (const String& s);`

Übergabe von Argumenten 6/6

UNI FREIBURG

- InOut-Parameter, Entscheidungshilfe
  - InOut-Parameter: Wird gelesen und geändert
  - Wir wählen in diesem Fall call by pointer:
    ```
    void f(int* i);
    void f(String* s);
    ```
  - Man könnte auch call by reference machen (macht z.B. die C++-Standardbibliothek), aber cpplint will Zeiger.
  - Man sieht dann dem Aufruf an, dass das Argument potenziell geändert wird:

    ```
    String s;
    f(&s); // s might be changed, why else do we take the address
    f2(s); // Does this change s or not? Need to look at declaration
    ```

Rückgabe von Objekten 1/6

UNI FREIBURG

- Möglichkeit 1: "Normale" Rückgabe
  - Funktion

    String reversed() { String result; … ; return result; }
  - Aufruf

    String s; String r = s.reversed(); // What happens here?
  - Theoretisch werden hier **zwei** temporäre Objekte erzeugt
    Die meisten Compiler vermeiden das allerdings ("copy elision" bzw. "return value optimization"), ab C++17 ist das teilweise Pflicht
  - Wenn keine copy elision, dann wird oft automatisch der Move-Konstruktor verwendet man schreibt meistens **nicht**

    return std::move(something)
    (verhindert die copy elision)

Rückgabe von Objekten 2/6

UNI FREIBURG

- Wann muss/ sollte man return std::move(…) schreiben
  - Copy elision oder impliziter move passieren nur wenn man eine einzelne Variable zurückgibt, die auf ein **vollständiges** Objekt zeigt, das **keine Referenz** ist

```
String f(bool b) {
   String s1, s2;
   // Ternary operator.
   return b ? std::move(s1) : std::move(s2);
}
String f2() {
   DataWithName dwn; // struct thas has a String member
   return std::move(dwn.name_);  // subobject
}
String f3() {
  StringSorter sorter(4);
  return std::move(sorter[3]); // reference
}
```

Rückgabe von Objekten 3/6

UNI FREIBURG

- Möglichkeit 2: Rückgabe einer Referenz
  - Funktion

    String& doof() { String r("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
  - Benutzt man aber oft als quasi-setter/getter in Klassen:
    String& StringSorter::operator[](size_t i) { return data_[i];}

Rückgabe von Objekten 4/6

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 5/6

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
  - Kommt in C-Bibliotheken häufig vor, etwas ähnliches braucht man in C++ manchmal bei Vererbung (VL 7+n)

Rückgabe von Objekten 6/6

UNI FREIBURG

- Möglichkeit 5: Rückgabe via Argument
  - Funktion

    ```
    void doof(String* r) { *r="doof"; }
    ```

  - Aufruf

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

  - Gleich wie InOut-Parameter (Folie TODO).
  - **Aber**: Bei reinen Out-Parametern (r wird nicht gelesen) besser als return value zurückgeben
  - Viele C-Bibliotheken verwenden Rückgabe via Argument, der Rückgabewert ist dann meist ein Fehlercode

Hinweise zum Ü7 1/3

UNI FREIBURG

- Effizienteres String-Sortieren mittels move
  - Bauen Sie die String- und StringSorter-Klassen so um, dass die Strings wo immer möglich hocheffizient gemoved warden.
  - Messen Sie, wie viel schneller das Sortieren geht, wenn man moved, anstatt zu kopieren.

Hinweise zum Ü7 2/3

UNI FREIBURG

- Zeitmessung
  - Im header <ctime> gibt es die Funktion
    clock_t clock();
  - Die gibt einem die aktuelle Zeit in CPU-Zyklen aus (clock_t ist ein integer-Typ.
  - Typischerweise ruft man das vor und nach einer Berechnung auf, die Differenz ist dann die vergangene Zeit.
  - Die Konstante CLOCKS_PER_SEC kann man verwenden, um CPU-Zyklen in Sekunden umzurechnen.
  - Ein int kann man mittels static_cast<double>(irgendeinInt) in ein double umwandeln, ohne dass Checkstyle meckert.

Hinweise zum Ü7 3/3

UNI FREIBURG

- Zufällige Zahlen
  - Im Header <cstdlib> gibt es die Funktion lrand48(). Die gibt eine Pseudozufallszahl zwischen 0 und 2^31 -1 zurück.

Literatur / Links

- Move-Semantik:
  - 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