Programmieren in C++

SS 2022

Vorlesung 8, Dienstag 21. Juni 2022

(Templates, Bitweise Operationen)

Johannes Kalmbach
Lehrstuhl für Algorithmen und Datenstrukturen
Institut für Informatik
Universität Freiburg

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Erfahrungen mit dem Ü7 — StringSorter mit Move
  - Treffen mit Tutor*in — Ablauf
- Inhalt
  - Templates — Prinzip + Beispiel
  - Templates — Instanziierung
  - Templates — Spezialisierung
  - Bitweise Operatoren — & | ^ ~ << >>
  - **Ü8:** Eine templatisierte Klasse Set<T>, mit einer effizienteren Implementierung für **unsigned char**

Erfahrungen mit dem Ü7 1/2

UNI FREIBURG

- Rückmeldung zur Aufgabe

  "Bei mir war der Move-Konstruktor durchgängig ziemlich genau 58-mal so schnell wie der Copy-Konstruktor.„

  "Bei mir ist es gleich schnell, weil ich […] minSort verwendet habe"

  " Ich fand auch, dass das ÜB viel weniger Aufwand war als das letzte."

  "100 zufällige Strings zu erstellen mit der methode lrand48() war etwas zeitaufwändig, aber hat geholfen den Umgang mit valgrind, new[] und delete[] zu vertiefen."

  "Aufgabe 1 war easy, Aufgabe 2 war schmerzhaft"

  "ich habe die vorgegebenen Tests vermisst...."

Erfahrungen mit dem Ü7 2/2

UNI FREIBURG

- Rückmeldung zur Aufgabe

  "vielleicht könnte man ja in Zukunft beim Live-Programmieren die Kommentare schon in einer Datei im Vornherein verfassen. Die sind nämlich meistens eh immer eine Wiedergabe von dem was grad erklärt wurde„

  "Wie schon in der Vorlesung beprochen sind new/delete operationen teuer eine Verbesserung der Laufzeit mit "Move" erreicht werden kann.„

  "Obwohl ich dachte, es in der Vorlesung verstanden zu haben, habe ich eine Weile gebraucht, um zu begreifen[...]"

  "Bei der ersten Aufgabe hätte ich eigentlich gerne getestet, dass die Move-Methode auch für temporäre Objekte angewendet wird[...]"

Treffen mit der Tutor*in

UNI FREIBURG

- Ablauf
  - Bitte schauen Sie, dass Ihre Email in Daphne stimmt, kontaktieren Sie ansonsten Ihr*e Tutor*in
  - Sie bekommen in der kommenden Woche eine Einladung zu einem treffen, digital oder evtl. in Präsenz
  - Treffen bis spätestens Mitte Juli
  - Sicherstellung dass Sie der Mensch sind, der Ihre Übungen abgibt
  - Bringen Sie gerne Fragen mit
  - Kurze Ausweiskontrolle am Anfang

Live-Coding

UNI FREIBURG

- In dieser Vorlesung
  - Klassen für ein Feld fester Größe mit den Funktionen

    Erzeugen mit fester Größe, i-tes Elemente, …
  - Wir implementieren folgende Varianten:

    Klasse **ArrayInt** für Elemente vom Typ int

    Klasse **ArrayFloat** für Elemente vom Typ float

    Template-Klasse Klasse **Array<T>** für beliebigen Typ T

    Spezialisierung **Array<bool>**

Gelöschte Funktionen

UNI FREIBURG

- Syntax und Use Case
  - Wenn man hinter eine Funktionsdeklaration = delete schreibt, dann gibt es diese Funktion nicht.
  - Nützlich bei Funktionen, die der Compiler ansonsten automatisch erzeugt, z.B. Copy-Konstruktor, Copy-Assignment, etc.

```
Class S {
  // …
  public:
  S(const S& other) = delete;  // no copy constructor
  S& operator=(const S& other) = delete; // no assignment
};
```

Templates 1/8

UNI FREIBURG

- Motivation
  - Manchmal hat man Klassen, die man fast genauso auch für einen anderen Typ braucht, zum Beispiel

    class ArrayInt { ... methods … int* _elements; };

    class ArrayFloat { ... methods … float* _elements; };
  - Der Code ist fast identisch, außer dass an diversen Stellen in der einen Klasse int und in der anderen float steht
  - Solche "code duplication" ist immer schlechter Stil

    Wenn man in der einen Klasse was ändert, ist die Gefahr groß, dass man es in der anderen vergisst

    Außerdem ist es unnötige (nicht ganz) doppelte Arbeit

Templates 2/8

UNI FREIBURG

- Motivation
  - Mit Templates kann man einen oder mehrere Typen bei der Implementierung "offen lassen"

```
template<class T>
class Array { ... methods ... T* _elements; }
```

  - Erst bei der Deklaration gibt man dann den Typ an

```
Array<int> a1;          // Array of ints.
Array<float> a2;        // Array of floats.
Array<char> a3;         // Array of chars.
```

  - Das <...> ist dabei Teil des Klassennamens

Der Compiler erzeugt in der Tat für jedes T anderen Code

Templates 3/8

UNI FREIBURG

- Instanziierung
  - Bei Template-Klassen wird durch die Implem. in der .cpp Datei noch **kein** Code erzeugt … siehe nm –C Array.o
  - Um Code zu erzeugen, muss man sagen für welche Typen T man die Klasse gerne hätte
  - Dafür gibt es zwei Alternativen

    Alternative 1: **Header-Implementierung** in der .h Datei

    Alternative 2: **Explizite Instanziierung** in der .cpp Datei
  - Beide Alternativen haben Vor- und Nachteile

    Wir bevorzugen in dieser Veranstaltung Alternative 2

Templates 4/8

UNI FREIBURG

- Instanziierung, Alternative 1
  - Die Deklaration wie gehabt in der .h Datei
  - Man schreibt die Implementierung, die normalerweise in der .cpp Datei steht, AUCH in die .h Datei
  - **Vorteil:** Klasse wird erzeugt wenn sie gebraucht wird

```
#include "./Array.h"
...
Array<int> arrayInt;  // Here Array<int> gets compiled.
```

  - **Nachteil:** Wenn in 10 verschiedenen Dateien Array<int> benutzt wird, wird der Code dafür 10 mal kompiliert

    Das ist aber ok, wenn der Code schnell zu kompilieren ist

Templates 5/8

UNI FREIBURG

- Instanziierung, Alternative 2
  - Explizite Code-Erzeugung am Ende der .cpp Datei

```
template class Array<int>;      // Compile Array<int>
template class Array<float>;    // Compile Array<float>
```

  - **Vorteil:** Code für Array<int> und Array<float> wird jetzt nur einmal erzeugt und steht in der entsprechenden .o Datei
  - **Nachteil:** Man muss sich entscheiden, für welche T man den Code haben will und für welche nicht

    Insbesondere ist das unmöglich für Bibliotheken, wo man gar nicht wissen kann, für welchen Typ T jemand die Template-Klasse später mal benutzen will

Templates 6/8

UNI FREIBURG

- Template-Funktionen
  - Man kann auch nur einzelne Funktionen "templatisieren"

```
template<class T> T cube(T x) {
  return multiply(multiply(x, x), x);
}
...
int n = 3;
printf("n^3 = %d\n", cube<int>(n));  // Prints 27.
```

  - Erst beim **Aufruf** wird die Funktion für diesen Typ kompiliert
  - Und auch dann erst dann gibt es ggf. Fehlermeldungen
  - Man kann beim Aufruf statt cube<int> auch cube schreiben ... der Compiler findet den richtigen Typ T dann selber heraus

Templates 7/8

UNI FREIBURG

- Fehlermeldungen bei Templates ...
  - ... sind oft sehr lang und verwirrend

    FileX:123: instantiated from [some function]
    FileY:456: instantiated from [some other function]
    ...
    FileZ:789: instantiated from [yet another function]
    SomeFile.cpp:666: instantiated from here
    SomeFile.h:555: error: [some error message]

  - Am wichtigsten sind dabei meistens die **ersten** und **letzten** Zeilen
  - Die Zeile mit "instantiated from here" sagt, wo das Template zum ersten Mal konkret **benutzt** wird
  - Die Zeile mit "error" sagt, wo beim Kompilieren im Template Code der Fehler aufgetreten ist

Templates 8/8

UNI FREIBURG

- Spezialisierung
  - Für einzelne Typen möchte man die Klasse vielleicht ganz anders implementieren, oft aus Effizienzgründen
  - Deklaration in der .h Datei geht so:

    template<> Array<bool> { … }
  - Implementierung in der .cpp Datei **ohne** template:

    Array<bool>::get { ... }
  - Wichtig zu verstehen: die spezialisierte Klasse kann auch ganz andere Memberfunktionen- und variablen haben

    unsigned char _elements anstatt T* _elements

    Neue Memberfunktion int memory_used()

Bitweise Operatoren 1/4

UNI FREIBURG

- Beispiele

```
char  x = 7;      // In binary: 00000111
char  y = 18;     // In binary: 00010010
```

  - Oder, Und, Exklusiv-Oder, Negation

```
x | y             // In binary: 00010111
x & y             // In binary: 00000010
x ^ y             // In binary: 00010101
~x                // In binary: 11111000
```

  - Bits nach rechts oder links schieben

```
x << 2            // In binary: 00011100
y >> 3            // In binary: 00000010
```

Bitweise Operatoren 2/4

UNI FREIBURG

- Setzen einzelner Bits
  - "Maske" mit genau einer 1 oder 0 in der Binärdarstellung

```
unsigned char m0 = 1 << 0;    // 00000001
unsigned char m5 = 1 << 5;    // 00100000
~m0;                          // 11111110
~m5;                          // 11011111
```

  - Setzen und Löschen eines bestimmten Bits

```
unsigned char x = 19;          // 00010011

x | m5;                        //
x | m0;                        //
x & ~m0                        //
x & ~m5                        //
```

Bitweise Operatoren 3/4

UNI FREIBURG

- Achtung mit **signed char** vs. **unsigned char**
  - In C/C++ steht der Typ char für den Inhalt von einem einzigen **Byte** … also 256 verschiedene Werte
  - Es gibt eine Variante mit und eine ohne Vorzeichen:

    signed char: Werte im Bereich -128 .. 127
    unsigned char: Werte im Bereich 0 .. 255
  - Wichtig: char ohne Angabe ist auf vielen Plattformen (allerdings nicht allen, z.B. ARM) ein signed char

```
char x = 7;

printf("%d", x);                          // 7
printf("%d", ~x);                         // -8
printf("%d", (unsigned char)(~x));        // 248
printf("%u", ~x);                         //
```

Bitweise Operatoren 4/4

UNI FREIBURG

- Typen mit mehr Bits
  - Das geht genauso mit anderen Typen, z.B. int, dann mit entsprechend mehr Bits (bei int typischerweise 32 Bits)

    Achtung: wenn man sich auf die Anzahl Bits verlassen möchte, die folgenden (unsigned) Typen benutzen

    uint8_t, **uint16_t**, **uint32_t**, **uint64_t**, **...**
  - Es gibt sogar **128-Bit**, **256-Bit** und **512-Bit** Register

    Mit Bit-Operationen darauf (oft in Hardware gegossen) kann man einiges an Performance herausholen

    Dafür braucht man aber spezielle Bibliotheken (Stichwort: AVX Intrinsics) oder man hofft, dass der Compiler an geeigneten Stellen entsprechenden Code erzeugt

Literatur / Links

- Alles zu Templates
  - http://www.cplusplus.com/doc/tutorial/templates
- Bitweise Operatoren
  - http://www.cplusplus.com/doc/tutorial/operators/