Programmieren in C++

SS 2018

Vorlesung 8, Dienstag 19. Juni 2018

(Templates, Bitweise Operationen)

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 Ü7 — Reich werden mit Bitcoin
  - Prüfungsanmeldung — Erinnerung + Erklärung
- Inhalt
  - Templates — Prinzip + Beispiel
  - Templates — Instanziierung
  - Templates — Spezialisierung
  - Bitweise Operatoren — & | ^ ~ << >>
  - **Ü8:** Eine templatisierte Klasse Set<T>, mit einer effizienteren Implementierung für den Typ **char**

Erfahrungen mit dem Ü7 1/3

UNI FREIBURG

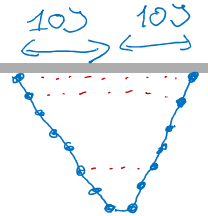
- Zusammenfassung / Auszüge
  - Sehr schöne, interessante, kreative Aufgabe
  - Klang erst schwieriger als es am Ende dann tatsächlich war
  - Aufwändiger als das vorherige Blatt, aber hat Spaß gemacht
  - Tests waren sehr hilfreich, aber nicht leicht zu verstehen, beim nächsten Mal besser dokumentieren, was sie tun
  - Den meisten gefällt das testgetriebene Programmieren
  - Panikattacke durch die schreckliche trading stragegy auf dem Ü7 (im Tief verkaufen, auf dem Hoch kaufen)
  - Die Strategie vom Ü7 ist schlecht: sie macht bei 1000 € Startkapital **nur** 2.2M € Gewinn … dabei gingen 755M €

Erfahrungen mit dem Ü7 2/3

UNI FREIBURG

- Kauft man Aktien wenn Sie steigen oder fallen?
  - Kaufen wenn sie steigen **werden**, verkaufen wenn sie fallen **werden**
  - Beste Strategie: bei schlechten Nachrichten alles verkaufen, bei guten Nachrichten alles kaufen (bevor es andere tun)
  - Vernünftige Menschen kaufen wertstabile Aktien nach einer Rabattaktion ("Crash"), die ANDERE Firmen betrifft
  - Man kauft sie am Tiefpunkt. Um den zu bestimmen, lädt man einfach die entsprechende CSV Datei herunter. Das sollte kein Problem sein, da ja eh alles deterministisch ist
  - Einer unserer Teilnehmer ist Tutor am Lehrstuhl für Finanzwesen, Rechnungswesen und Controlling :o

Erfahrungen mit dem Ü7 3/3



UNI FREIBURG

- Etwas Mathematik zum Aktienkauf
  - Angenommen folgendes Kurs- und Kaufverhalten einer Aktie:
    - fällt 10 Jahre stetig auf die Hälfte des ursprgl Wertes
    - steigt dann wieder 10 Jahre stetig auf den ursprgl Wert
    - Jeweils fixer Prozentsatz pro Monat
    - Man kauft jeden Monat für einen festen Betrag

$\ln 2 \approx 0.69\ldots$
$1/\ln 2 \approx 1.44\ldots$

  - Wie groß ist der Gewinn oder Verlust nach 20 Jahren?
  - Monatliche Rate k, $2 \cdot N$ Raten, Gesamtkapital $K = 2k \cdot N$
  - Prozentsatz pro Monat $q = 2^{1/N} = e^{\ln 2/N} \approx 1 + \ln 2/N$ ($q^N = 2$)
  - Nach 20 Jahren: $2k \cdot (q^0 + \ldots + q^{N-1}) = 2k \cdot (q^N - 1)/(q - 1)$ ($2-1=1$, $\ln 2/N$)

Bei $q = 3^{1/N} \Rightarrow K \cdot \frac{2}{\ln 3} \approx K + 82\%$

$\approx 2k \cdot N / \ln 2 = K / \ln 2$
$\approx K + 44\%$

Anmeldung Prüfung

UNI FREIBURG

- Sie müssen sich anmelden!
  - Auch wenn die Veranstaltung "nur" eine Studienleistung ist
  - Die Prüfung war allerdings bisher vom Prüfungsamt nicht korrekt eingetragen im HISinOne, das ist aber jetzt korrigiert

    Man kann (und muss) sich anmelden für "11LE13SL-840-1 Programmierung in C++ - Studienleistung"
  - Fakultätsfremde sollen sich bei Problemen an das Prüfungsamt der Technischen Fakultät wenden
  - Für die BOK-Teilnehmenden ist es eine andere Veranstaltung

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> here.
template class Array<float>;    // Compile Array<float> here.
```

  - **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

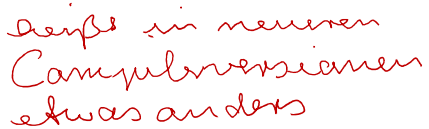
UNI FREIBURG

- 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



- 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 **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
  - Die Deklaration in der .h Datei schreibt man dann so

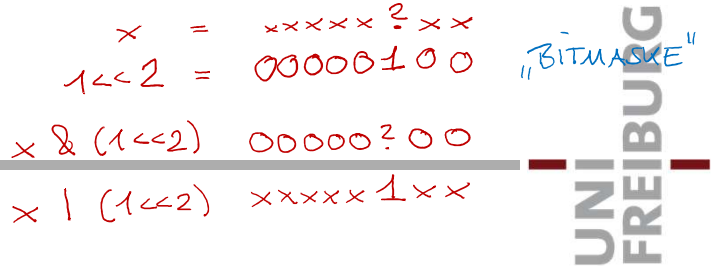
```
template<> Array<bool> {
  // Different implementation, packing 8 Bits in 1 Byte.
}
```

  - Die Implementierung in der .cpp Datei dann **ohne** template

```
Array<bool>::get { ... }
```

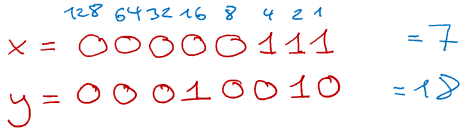
  - Wichtig zu verstehen: die spezialisierte Klasse kann auch ganz andere Memberfunktionen- und variablen haben

Bitweise Operatoren 1/2



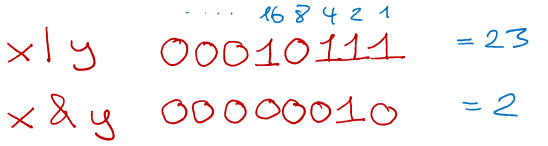
- Beispiele

```
char  x = 7;      // 00000111 in binary
char  y = 18;     // 00010010 in binary.
```



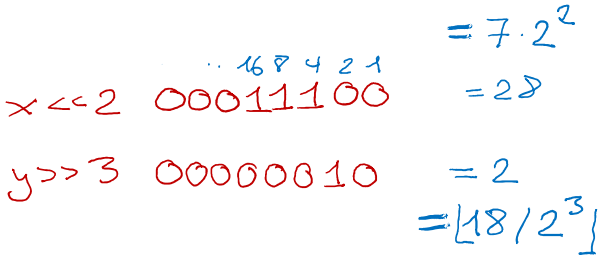
  - Oder, Und, Exklusiv-Oder, Negation

```
x | y      // 00010111 in binary = 23.
x & y      // 00000010 in binary = 2.
x ^ y      // 00010101 in binary = 21.
~x         // 11111000 in binary = 248.
```



  - Bits nach rechts oder links schieben

```
x << 2     // 00011100 in binary = 28.
y >> 3     // 00000010 in binary = 2.
```



Bitweise Operatoren 2/2

UNI FREIBURG

- Fortsetzung
  - Das geht genauso mit anderen Typen, z.B. int
  - Es werden dann entsprechend mehr Bits auf einmal manipuliert, beim int z.B. typischerweise 32 Bits
  - Moderne Rechner haben auch 256-Bit oder sogar 512-Bit Register
  - Mit Bit-Operationen darauf (oft in Hardware gegossen) kann man einiges an Performance herausholen

Für die (optionale) Zusatzaufgabe: uint64_t wäre ü8

Hinweis zum Übungsblatt

UNI FREIBURG

- Der Typ 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 auf ARM) ein signed char
  - Wenn man den Wert eines char als Index für ein Feld benutzen will, sollte man ihn als unsigned char casten

```
char c = …;
unsigned char i = static_cast<unsigned char>(c);
```

Literatur / Links

UNI FREIBURG

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