Programmieren in C++

SS 2011

Vorlesung 8, Mittwoch 29. Juni 2011

(Templates, valgrind, bitweise Operatoren)

Prof. Dr. Hannah Bast
Lehrstuhl für Algorithmen und Datenstrukturen
Institut für Informatik
Universität Freiburg

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Treffen mit Ihrem Tutor, bitte reagieren Sie jetzt!
  - Erfahrungen mit dem 7. Übungsblatt
- Templates
  - Was ist das und wofür ist das gut?
  - Deklaration, Instantiierung, Spezialisierung
  - Außerdem: die bitweisen Operatoren `&` `|` `^` `~` `<<` `>>`, das Speicherzugriffsfehlerentdeckungstool valgrind und SCOPE_TRACE
  - Übungsblatt: Machen Sie aus Ihrer String Klasse eine templatisierte Klasse und schreiben Sie eine Spezialisierung für Bitstrings

Erfahrungen mit dem 7. Übungsblatt

UNI FREIBURG

- Zusammenfassung / Auszüge Stand 29.6 15:27
  - Wieder schwierigeres Blatt, aber mit der Vorlage ging's
  - Sehr viele Testfälle diesmal, das kostet viel Zeit und ist nervig
  - Viele kleine Fehler, dadurch viel Zeit gekostet
  - Zeitaufwand: ca. 5h, Nervenverbrauch: ca. 90%
  - Stringvergleich in Tests char für char sehr umständlich
  - Probleme bei Ausgabe von strings ohne 0 am Ende
  - valgrind erklären, bei memory leaks / double-free / etc.
  - const_cast<char*> bitte erklären (keine Zeit in letzter VL)
  - Welche Allokationen werden automatisch wieder freigegeben?
  - Warum Parsen der Arg. in Grep Klasse und nicht in der Main?
  - Beim nächsten gleich alles in eine Klasse, nicht erst in die Main
  - Am Ende zu wenig Zeit für ASSERT_DEATH, const_cast, ...
  - Bitte weiter Übungsblätter mit Bezug zur Realität

Wie ausführlich sollen Unit Tests sein?

UNI FREIBURG

- Manchmal gibt es sehr viele "Grenzfälle"

  ... wobei hier mit "Grenzfälle" Aufrufe der zu testenden Funktion gemeint sind, wo potenziell etwas schief gehen kann

  – Typische Grenzfälle sind zum Beispiel ein leerer String, eine negative Position in einem String oder einem Feld, eine zu große Position, etc.
  – In Ihren Tests sollten Sie neben einem "normalen" Fall auch immer wenigstens ein paar Grenzfälle betrachten
  – Wenn es sehr viele gibt, müssen Sie aber nicht alle betrachten, dann reichen **drei**
  – Ihr Code sollte aber schon für alle Fälle funktionieren, auch für solche, die sie nicht getestet haben ...

Hilfsfunktionen für Tests 1/3

UNI FREIBURG

- Oft nützlich, zum Beispiel für die String Klasse

```
void assertStringEquality(const char* e, const String& s) {
  for (int i = 0; i < s.size(); i++) ASSERT_EQ(e[i], s[i]);
  ASSERT_EQ(eq[s.size()], 0);
}
...
TEST(StringTest, set) {
  String s;
  s.set("123");
  assertStringEquality("123", s);
}
```

  - Problem: schlägt ein ASSERT_EQ fehl, kommt eine Zeilennummer aus der **Hilfsfunktion**, anstatt der Zeilennummer aus dem Test, wo die Hilfsfunktion aufgerufen wurde

Hilfsfunktionen für Tests 2/3

UNI FREIBURG

- Lösung 1: SCOPED_TRACE

```
TEST(StringTest, set)
{
  String s;
  s.set("123");
  SCOPED_TRACE("assertStringEquality called from here");
  assertStringEquality("123", s);
}
```

  - Dann wird bei einem failure in assertStringEquality, das Argument aller SCOPED_TRACE Anweisungen im umgebenden Block ausgegeben

```
Google Test Trace:
StringTest.cpp:33: assertStringEquality called from here
```

Hilfsfunktionen für Tests 3/3

UNI FREIBURG

- Lösung 2: implementiere Operatoren == und <<
  - Am liebsten würde man ja einfach sowas schreiben wie

```
TEST(StringTest, set)
{
  String s;
  s.set("123");
  ASSERT_EQ("123", s);
}
```

  - Aber dazu braucht das ASSERT_EQ hier zwei Sachen
    - wie man ein const char* mit einem String vergleicht
    - wie man einen String ausgibt (falls hier "123" != s)
  - Wie das geht machen wir nächste Woche …

valgrind

UNI FREIBURG

- Zum Aufspüren von Speicherzugriffsfehlern
  - Einfach valgrind vor das ausführbare Programm setzen

    valgrind ./StringTest
  - Das "simuliert" den Code und schaut dabei, an welchen Stellen auf unerlaubten Speicher zugegriffen wird, und ob irgendwo Platz, der mit new alloziert wurde, nicht mehr freigegeben wurde
  - Gibt dann einen sehr aussagekräftigen Problembericht
  - Achtung: das Programm läuft durch diese Simulation um ein Vielfaches langsamer!
  - Installation einfach mit sudo apt-get install valgrind
  - Quick start / Doku / etc. siehe http://valgrind.org/

Type casting (noch von der letzten VL)

UNI FREIBURG

- Implizite Typkonvertierung

```
bool flag = true;
int x = flag;  // Will get the value 1 (false = 0).
```

- Explizite Typkonvertierung
  - manchmal muss man einen Typ in einen anderen, eigentlich imkompatiblen, konvertieren

```
const char* message = "Hi";
char** argv = new char*[2];
argv[1] = message;  // This will not compile.
argv[1] = const_cast<char*>(message);  // This will.
```

  - es gibt auch noch static_cast, dynamic_cast und reinterpret_cast aber die kommen später erst

Allokation mit new / delete vs. automatisch

UNI FREIBURG

- Automatische Allokation
  - Passiert wann immer man eine Variable deklariert
  - Der Platz wird am Ende des { ... } Blocks, in dem die Deklaration steht (= ihr "scope"), wieder freigegeben

```
{
  int x;  // Allocates 4 bytes for x.
  ...
}  // Here the 4 bytes for x are freed automatically.
```

  - Auch bei Argumenten einer Funktion, der Platz wird dann am Ende der Funktion wieder freigegeben → VL 6
- Allokation mit new / delete
  - Alles was man mit new alloziert, muss irgendwann mit delete bzw. delete[] wieder freigegen werden, da passiert nichts automatisch, siehe Vorlesung 5

Templates 1/3

UNI FREIBURG

- Wofür braucht man templates?
  - Oft hat man Klassen, die man fast genauso auch für einen anderen Typ braucht, zum Beispiel

    class ArrayOfInts { ... Elemente vom Typ int ... };

    class ArrayOfChars { ... Elemente vom Typ char ... };
  - Jetzt kann man natürlich den Code kopieren und in der Kopie überall int durch char ersetzen
  - Aber das ist sehr schlechter Stil und fehleranfällig, weil man Änderungen dann immer an zwei Stellen machen muss
  - Genau dafür hat man **templates**, dann hat man ein und denselben Code für einen beliebigen Typ

    template<class T> Array<T> { ... Elemente vom Typ T ... };
  - Details siehe Codebeispiel in Array.h und Array.cpp

Templates 2/3

UNI FREIBURG

- Wie benutzt man so eine template Klasse?
  - Indem man in spitzen Klammern angibt, für welches konkrete T man die Klasse haben will

    Array<int> arrayOfInts;
    Array<char> arrayOfChars;
  - Das <int> bzw. <char> ist dabei Teil des Klassennamens
  - In der Tat sind bei der Benutzung Array<int> und Array<char> wie zwei völlig **verschiedene** Klassen
  - Wir haben lediglich den Code dafür auf besondere Weise geschrieben(nämlich nur einmal, für beide Klassen zusammen)

Templates 3/3

UNI FREIBURG

- Das geht auch mit einzelnen Funktionen

```
template<class T> T cube(T x)
{
  return x * x * x;
}
...
int n = 3;
printf("n cubed = %d\n", cube<int>(n));  // Prints 27.
double pi = 3.14;
printf("pi cubed = %.2f\n, cube<double>(pi)); // Prints 30.96.
```

  - **Achtung:** erst beim Aufruf der jeweiligen Funktion wird geschaut, ob die Funktion für den Typ überhaupt kompiliert werden kann; im Beispiel oben, ob es überhaupt einen `operator*` für den Typ gibt

Template Fehlermeldungen

UNI FREIBURG

- ... haben eine charakteristische Form

  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]

  – Die Zeile mit dem error und die davor sind wichtig!
    - Die Zeile davor sagt, wo das template für einen konkreten Typ **benutzt** wurde und ein Fehler auftrat
    - die Zeile mit dem error sagt, wo dieser Fehler in der template **Deklaration** ist

Template Instantiierung 1/3

UNI FREIBURG

- Eine template Implem. erzeugt noch **keinen** Code!
  - Siehe nm –C Array.o
  - Um Code zu erzeugen, muss man sagen für welche konkreten T man die Klasse gerne hätte
  - Dafür gibt es zwei Alternativen
    - Alternative 1: Implementierung in der .h Datei
    - Alternative 2: explizite Instantiierung in der .cpp Datei
  - Beide Alternativen haben Vor- und Nachteile
  - Wir bevorzugen in dieser Veranstaltung **Alternative 2**

Template Instantiierung 2/3

UNI FREIBURG

- Alternative 1: Implementierung in der .h Datei
  - 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> arrayOfInts;  // Here Array<int> gets compiled.
...
Array<char> arrayOfChars;  // Here Array<char> gets compiled.
```

  - **Nachteil:** Wenn in 10 verschiedenen Dateien ein Array<int> benutzt wird, wird der Code dafür 10 mal kompiliert
  - Ok, wenn der Code relativ einfach / schnell zu kompilieren ist

Template Instantiierung 3/3

UNI FREIBURG

- Alternative 2: Explizite Instantiierung
  - Explizite Code-Erzeugung in der .cpp Datei

    ```
    template class Array<int>;  // Compile Array<int> here.
    template class Array<char>;  // Compile Array<char> here.
    ```

  - **Vorteil:** der Code für die entsprechenden Klassen muss jetzt nur einmal kompiliert werden und steht auch nur einmal irgendwo, nämlich in der entsprechenden .cpp Datei
  - **Nachteil:** man muss in der .cpp Datei explizit sagen, für welche T man den Code haben will
    - das ist unmöglich für Bibliotheken, wo der Schreiber der Bibliothek gar nicht wissen kann, für welches T jemand die templatisierte Klasse später mal benutzen will

      ```
      Array<MyPersonalAndStrangeObject> myArray;
      ```

Template Spezialisierung

UNI FREIBURG

- Manchmal möchte man für einen einzelnen Typ die Sache ganz anders implementieren
  - Zum Beispiel für ein Array<bool> 8 Bits in einen char packen (defaultmäßig braucht ein bool einen ganzen char)
  - Die Deklaration in der .h Datei schreibt man dann so

```
template<> Array<bool>
{
  hier jetzt die speziellen Deklaration für Array<bool>,
  die beliebig anders sein können als die in Array<T>.
}
```

  - Die Implementierung in der .cpp Datei entsprechend
  - Siehe Codebeispiel in Array.h und Array.cpp

Operatoren zur bitweisen Manipulation

- Hier anhand von ein paar Beispielen erklärt

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

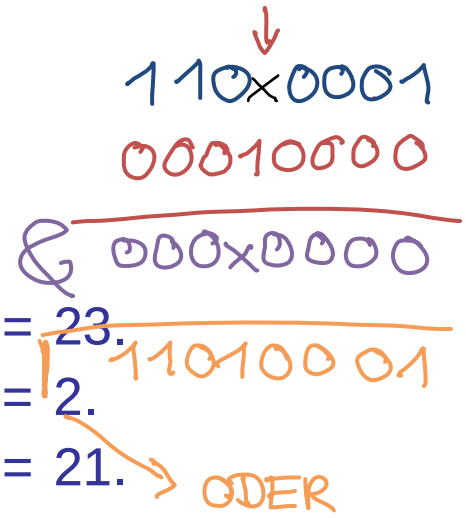
  - Bitweise Operatoren für AND, OR, XOR, NOT

```
char x_or_y = x | y;        // 00010111 in binary = 23.
char x_and_y = x & y;       // 00000010 in binary = 2.
char x_xor_y = x ^ y;       // 00010101 in binary = 21.
char not_x = ~x;            // 11111000 in binary = 249.
```

  - Bitschiebe-Operatoren << und >>

```
char x_shifted = x << 2; // 00011100 in binary = 28.
char y_shifted = y >> 3; // 00000010 in binary = 2.
```

  - Das geht genauso mit zum Beispiel int nur sind es dann entsprechend mehr Bits, typischerweise 32



UNI FREIBURG

- Templates, Instantiierung, Spezialisierung
  - http://www.cplusplus.com/doc/tutorial/templates
- Bitweise Operatoren
  - http://www.cplusplus.com/doc/tutorial/operators/
- Hilfsfunktionen für Tests / SCOPED_TRACE
  - http://code.google.com/p/googletest/wiki/AdvancedGuide#Using_Assertions_in_Sub-routines
- Valgrind
  - http://valgrind.org/
- const_cast
  - http://www.cplusplus.com/doc/tutorial/typecasting/