Programmieren in C++

SS 2012

Vorlesung 9, Dienstag 3. Juli 2012

(STL, vector, string, sort, iostream, namespaces)

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 8. Übungsblatt (Templates)
  - Ein paar Vorab-Infos über das Projekt am Ende
- Die STL = Standard Template Library
  - Klassen für Sachen, die man oft braucht
  - std::vector = ähnlich zu unserer Array Klasse
  - std::string = ähnlich zu unserer String Klasse
  - std::iostream für Ein- und Ausgabe
  - std::stringstream zum "Zusammenbauen" von Strings
  - std::sort zum Sortieren von beliebigen Objekten
  - und was heißt dieses std::... ?
  - **9. Übungsblatt:** Anagramme finden

Erfahrungen mit dem Ü8 (Templates)

UNI FREIBURG

- Zusammenfassung / Auszüge Stand 3. Juli 14:00
  - Aufgabe war sehr gut machbar, weniger Zeit als sonst
  - Die TemplatedString<bool> hat aber bei einigen gedauert
    - das Bitgefriemel hat einigen keinen Spaß gemacht
  - Wobei man viel aus dem Code von der VL benutzen konnte
  - Wie erzeugt man denn sowas wie 11110111 ? 
  - Die vorgegebenen Tests haben nicht alles getestet
  - Freitag programmieren ist wirklich doof
  - gdb / valgrind waren nützlich
  - Wann werden die Abschlussprojekte vorgestellt?
  - Mehr Zeit als es 4 ECTS Punkte entspricht … 4 ECTS = 6h/Ü
  - Videoaufzeichnungen lassen sich nicht durchsuchen

Erfahrungen mit dem Ü8 (Templates)

UNI FREIBURG

- Fortsetzung ...
  - Wie hacke ich mich in den Server, um die Lösungen vorzeitig zu erhalten?
  - Programme mit GUI?

Das Projekt am Ende der Vorlesung 1/3

UNI FREIBURG

- Art des Projektes
  - Zwei Aufgaben zur Auswahl, von verschiedenem Schwierigkeitsgrad
  - Für die Erfahreneren / Unterforderten (aber nur für die) die Möglichkeit ein eigenes Projekt zu definieren
  - Sie schreiben dann ein C++ Programm nach der Art, wie wir es in den Übungsblättern gemacht haben, mit Klassen, Tests, und allem ... nur halt etwas größer
  - Umfang ca. 4 Übungsblätter
  - Siehe die Beispiele von der Vorlesung im SS 2011

    http://ad-wiki.informatik.uni-freiburg.de/teaching

Das Projekt am Ende der Vorlesung 2/3

UNI FREIBURG

- Zeitlicher Rahmen
  - Wir fangen in der vorletzten Vorlesungswoche damit an
  - Das Übungsblatt dazu wird sein, sich ein Design für ihr Programm zu überlegen und die .h Dateien zu schreiben
  - Dazu kriegen Sie dann in der Woche drauf (so schnell wie möglich) Feedback von Ihren Tutoren
  - Wenn Sie sich ranhalten, können Sie Ihr Programm dann in der letzten Vorlesungswoche und in der Woche drauf fertig schreiben … dann haben Sie es hinter sich!
  - Finale Deadline für die Abgabe des Projektes ist

    Dienstag, der 11. September 2012 um 14 Uhr

Das Projekt am Ende der Vorlesung 3/3

UNI FREIBURG

- Muss man das Projekt machen?
  - Ja!
  - Auch wenn man sonst fast alle Punkte in den Übungen hat?
  - Ja!
  - Wirklich?
  - Ja!
  - Zum Bestehen brauchen Sie mindestens 50% der Punkte aus den Übungen **und** mindestens 50% der Punkte aus dem Projekt

Die STL

UNI FREIBURG

- STL = Standard Template Library
  - Voller nützlicher Klassen, die man immer wieder braucht
    - vector (unser Array), string (unser String), etc.
  - Dank templates können alle diese Klassen für alle möglichen Arten von Objekten benutzt werden können
    - vector<int>, string<w_char>, ...
  - Alle Klassen in der STL stehen im std namespace
    - Deswegen muss man schreiben std::vector<int> etc.
    - Erklärung dazu nächste Folie

Namespaces 1/2

UNI FREIBURG

- Ein **namespace** ist
  - … ein gemeinsamer Namenspräfix für Klassen die zu einem bestimmten Projekt oder zu einer Bibliothek gehören
  - Konkret heißt das, dass einfach um den Code herum steht

```
namespace std {
  ...
}
```

  - Wenn man etwas aus diesem namespace benutzt, muss man dann den Namen des namespace gefolgt von :: davor schreiben, also z.B. std::vector<int>
  - Grund: Ich will ja vielleicht meine eigene Klasse vector schreiben (und sie genauso nennen)

Namespaces 2/2

UNI FREIBURG

- Muss man das std:: wirklich immer davor schreiben?
  - So ohne Weiteres ja
  - Aber wenn Sie am Anfang der Datei schreiben

    using namespace std;

    dann können Sie es in der Datei weglassen
  - Damit machen Sie aber den Schutz vor Namensgleichheit zunichte, weswegen es namespace überhaupt gibt
  - Deswegen wird der Linter meckern ... zu Recht!
  - Für spezielle Klassen können Sie aber schreiben

    using std::vector;
  - Dann können Sie in dem Rest der Datei schreiben

    vector<int> array;

std::vector

UNI FREIBURG

- Die Klasse **std::vector** ist
  - … für dynamische Felder von Objekten von einem beliebigen Typ, ähnlich zu unserer Klasse Array aus der 8. Vorlesung

```
std::vector<int> numbers;
numbers.push_back(1);
numbers.push_back(2);
for (size_t  i = 0; i < numbers.size(); i++)
  printf(%d\n, numbers[i]);
```

  - Häufig benutzte Methoden: size, push_back, pop_back, resize, reserve, clear, ...
  - Details dazu siehe Referenzen am Ende

std::string

UNI FREIBURG

- Die Klasse **std::string** ist
  - … ein komfortable Klasse für Zeichenketten, ähnlich zu unserer Klasse String vom 8. Übungsblatt

```
std::string s = "doov";
size_t pos = s.find("v");
if (pos != std::string::npos) { s[pos] = 'f'; }
printf("%s\n", s.c_str());  // Prints "doof".
```

  - Intern speichert die Klasse einen null-terminierten C-String, und den bekommt man mit der c_str() Methode
    - das ist nützlich z.B. wenn man printf benutzt, siehe oben
  - Häufig benutzte Methoden: size, append, erase, substr, find, find_first_of, …
  - Details dazu siehe Referenzen am Ende

std::sort 1/3

- Zum Sortieren von Objekten

```
#include <algorithm>
#include <vector>

std::vector<int> array;
array.push_back(2);
array.push_back(1);
array.push_back(3);
std::sort(array.begin(), array.end());
```

  - Ohne weiteres Argument wird einfach der Operator **<** auf dem Elementtyp benutzt
  - Im Beispiel ist das int ... und nach dem std::sort stehen dann in array die Elemente 1 2 3 in der Reihenfolge

std::sort 2/3

- Sortieren mit eigener Vergleichsfunktion

```
// My own comparison function for two ints.
class MyComparison {
  // Return true iff x comes before y in desired order.
  public: bool operator()(const int& x, const int& y) {
    return x > y;  // Larger number wins now.
  }
};
...
std::vector<int> array;
... // Fill the vector with some numbers.
MyComparison cmp;  // Object from the class above.
std::sort(array.begin(), array.end(), cmp);
```

std::sort 3/3

UNI FREIBURG

- Ein paar Hinweise zur internen Funktionsweise
  - In der std::sort Methode wird immer wenn zwei Elemente x und y verglichen werden sollen cmp(x, y) aufgerufen
    - das ruft gerade die Methode MyComparison::operator() mit den Argument x und y auf
  - Auf der vorherigen Folie ist die Klasse **inline** definiert, das heißt gleich bei der Deklaration in der .h Datei
  - Dann setzt der Compiler an die Stelle des Aufrufes cmp(x, y) gleich den Code aus der Funktion, in dem Fall x > y
  - Das spart im Maschinencode einen Funktionsaufruf, das heißt: das Hin- und Zurückspringen im Code
  - Da Sortieren aus nicht viel mehr als (vielen) Vergleichen besteht, spart das einen Faktor von etwa 2 in der Laufzeit

std::iostream 1/2

UNI FREIBURG

- Oberklasse für die Ein- und Ausgabe
  - std::ofstream für die Ausgabe in eine Datei
  - std::ifstream für das Lesen aus einer Datei
  - Sind von beschränktem Nutzen, weil fopen, fread, fwrite, fprintf, ... dasselbe genau so gut oder besser tun
  - Die Objekte für die Benutzereingabe und die Ausgabe auf dem Bildschirm sind aber manchmal ganz nützlich
    - std::cout für die Ausgabe nach stdout
    - std::cerr für die Ausgabe nach stderr
    - std::cin für die Eingabe von stdin
  - Für das "\n" schreibt man std::endl

std::iostream 2/2

UNI FREIBURG

- Beispiel für cin, cout und cerr

```
using std::cin;
using std::cout;
using std::cerr;
using std::endl;
...
int x, y;
cin >> x >> y;  // Reads two ints from stdin.
if (x < 0 || y < 0) {
  cerr << "x and y should be positive!" << endl;
}
cout << "x = " << x << "; y = " << y << endl;
```

  - Details dazu und zu iostream ... siehe Referenzen am Ende

std::stringstream 1/4

UNI FREIBURG

- Nützlich für das Zusammenbauen von strings
  - Insbesondere in folgendem Kontext
  - Nehmen wir an, wir haben unsere eigene Klasse, z.B. ArrayInt, und wollen uns den Inhalt eines Objektes davon anschauen
  - Bisher haben wir dafür eine print() Methode geschrieben
  - Vielseitiger ist aber eine asString() Methode, die ein Objekt in einen string schreibt, in menschen-lesbarer Form
  - Den string können wir dann so ausgeben

```
cout << array.asString() << endl;          // If you prefer cout.
printf("%s\n", array.asString().c_str());  // If you prefer printf.
```

  - Oder in einem Test auf Gleichheit testen

```
ASSERT_EQ("[1, 3, 7, 12]", array.asString());
```

std::stringstream 2/4

UNI FREIBURG

- Beispiel Implementierung (in der .cpp Datei)

```
#include <sstream>
...
// Return object as human-readable string.
string SetOfIntegers::asString() {
  std::ostringstream os;  // Stream for assembling a string.
  os  <<  "[";  // Can be used just like cout.
  for (size_t  i = 0; i < _size; i++) {
    os  <<  (i > 0 ? ", " : "")  <<  _elements[i];
  }
  os  <<  "]";
  return os.str();  // The assembled string;
}
```

std::stringstream 3/4

UNI FREIBURG

- Warum lassen wir asString einen string zurückgeben
  - Und nicht einfach einen const char* ?
  - Wir müssen in der Funktion asString Speicherplatz für die Zeichenkette, die wir da zusammenbauen, allozieren
  - Wenn wir das mit new char[...] machen, muss derjenige, der asString() aufruft, den Speicher wieder freigeben
    - Das ist doof ... siehe Vorlesung 6 (Funktionsaufrufe)
  - Wenn wir das mit einem string Objekt machen, wird die Allokation von dem Objekt übernommen
    - Sobald der scope des Objektes beendet ist, wird der Destruktor aufgerufen und der Speicher freigegeben

std::stringstream 4/4

UNI FREIBURG

- Man kann auch den << Operator überladen
  - Das heißt, eine Funktion **operator<<** mit geeigneten Argumenten schreiben, so dass man schreiben kann

    ```
    cout << "My array is " << array << endl;
    ```

    im Gegensatz zu

    ```
    cout << "My array is " << array.asString() << endl;    (*)
    ```
  - Für die string Klasse aus der STL brauchen wir das, sonst könnten wir (*) gar nicht schreiben
  - Ansonsten brauchen wir es nicht ... im Gegenteil, es führt oft zu schwer verständlichen Fehlermeldungen
    - die Variante mit asString ist vielseitiger und klarer
  - Wen's trotzdem interessiert ... siehe Referenzen

explicit

UNI FREIBURG

- Bei Konstruktoren mit **genau** einem Argument

  … möchte der Linter, dass wir **explicit** davor schreiben

  ```
  // Stupid constructor that creates array with one element.
  explicit ArrayInt(int x);
  ```

  – Tun wir das nicht, kann dieser Konstruktor auch zur automatischen Typkonvertierung verwendet werden

  ```
  // Method for appending an array to an array.
  ArrayInt::append(const ArrayInt& otherArrayInt) { ... }
  ...
  ArrayInt array;
  array.append(5);  // Will call the constructor above.
  ```

  – Das kann nett sein, kann aber vor allem zu schwer zu debuggenden Fehlern führen, deswegen meckert cpplint

Literatur / Links

UNI FREIBURG

- Standard Template Library (STL)
  - http://www.sgi.com/tech/stl
- Namespaces
  - http://www.cplusplus.com/doc/tutorial/namespaces
- vector, string, sort, iostream, stringstream
  - http://www.sgi.com/tech/stl/Vector.html
  - http://www.sgi.com/tech/stl/basic_string.html
  - http://www.sgi.com/tech/stl/sort.html
- Ein- und Ausgabe
  - http://www.cplusplus.com/doc/tutorial/basic_io/
  - http://www.cplusplus.com/doc/tutorial/files/