

18.911: Kompaktkurs C++ für Java-Programmierer

4./5. Tag

Reinhard Zierke

Universität Hamburg, MIN-Fakultät, Informatik-Rechenzentrum

8.-12. Oktober 2007

Übersicht 4./5. Tag

- 1 Templates, STL, Iteratoren, Exceptions, E/A
 - Templates
 - C++ Standard-Bibliothek (STL)
 - Fehlerbehandlung
 - Ein- und Ausgabe: Streams

Fehlerbehandlung mit abort und exit

- Statt im Fehlerfall eine Exception zu werfen, die ignoriert oder erst in `main()` abgefangen wird, kann man wie in C das Programm vorzeitig beenden:
- `void abort()`;
beendet das Programm sofort; es werden keine Destruktoren aufgerufen.
- `void exit(int status)`;
beendet das Programm sofort; es werden nur Destruktoren für statische Objekte aufgerufen, nicht für lokale Objekte in offenen Funktionsaufrufen
- Nie `exit()` im Destruktor eines statischen Objekts aufrufen
- `int atexit(void f(void))` : Registrieren von Funktionen zum Ausführen beim `exit`-Aufruf

Assertions

- `assert` ist ein Präprozessor-Makro; verfügbar nach `#include <cassert>`
- Wenn Präprozessor-Makro `NDEBUG` definiert ist, werden `assert`-Aufrufe ignoriert.
- Anderfalls, sofern der Parameter von `assert` Null ist, wird das Programm mit `abort` abgebrochen; der Ausdruck, Programmdatei und Zeilennummer werden ausgegeben.

```
void foo(string *s) {  
    assert(s != NULL);  
    ...  
}
```

Assertion failed: s!= NULL, file prog.cc, line 17

Werfen von Exceptions

- Beliebige Typen können als Exception geworfen werden,
- aber auch vordefinierte Exception-Klassen nutzbar
- Eine Methoden-Deklaration kann eine Liste von zu werfenden Exceptions aufführen
- Reiner Kommentar, wird vom Compiler nicht geprüft
- Schlüsselwort ist `throw`, nicht `throws`

```
void foo() throw(UnderflowException, OverflowException);
```

- `throw()` : keine Exceptions zu werfen
- kein `throw` : beliebige Exceptions möglich (Rückwärtskompatibilität)
- Nicht angekündigte Exception geworfen:
`std::unexpected` und damit `abort` wird aufgerufen
- Angekündigte Exception geworfen, aber nicht abgefangen:
`std::terminate` wird aufgerufen

Fangen von Exceptions

- Eine Folge von `catch`-Blöcken wird nacheinander auf passende Exception geprüft
- Der erste passende `catch`-Block wird verwendet
- `catch (...)` fängt beliebige Exceptions ab, das Exception-Objekt ist hier aber nicht zugreifbar
- Mit `what()` kann man weitere Informationen bekommen
- Exceptions können „weitergeworfen“ werden:

```
catch (domain_error e) {  
    ...           // lokale Fehlerbehandlung  
    throw;       // erneut werfen  
}
```

Standard-Exceptions in <stdexcept>

exception

- runtime_error
 - range_error
 - overflow_error
 - underflow_error
- logic_error
 - out_of_range
 - invalid_argument
 - length_error
 - domain_error
- bad_alloc
- bad_cast
- bad_typeid
- bad_exception
- ios_base::failure

Beispiel: Exception in der STL abfragen

```
#include <iostream>
#include <string>
#include <stdexcept>

string s = "1234";
try {
    cout << s.at(4) << endl;
}
catch (out_of_range &e) {
    cout << "Index-Exception:␣" << e.what() << endl;
}
catch (...) {
    cout << "allgemeine␣Exception" << endl;
    throw;
}
```

Beispiel: Standard-Exception benutzen (1)

```
#include <iostream>
#include <string>
#include <cmath>
#include <stdexcept>
using namespace std;

double myexp(double x) throw(out_of_range)
{
    double result = exp(x);
    if (result == HUGE_VAL)
        throw out_of_range("exp too large");
    else
        return result;
}
```

Beispiel: Standard-Exception benutzen (1)

```
int main() {
    try {
        cout << myexp(100000.) << endl;
    }
    catch (out_of_range &e) {
        cout << "out_of_range-Exception:_" << e.what() << endl;
    }
    catch (...) {
        cout << "Allg._Exception:_" << e.what() << endl;
    }
}
```

Beispiel: Exception mit einfachen Datentypen

```
string s="1234";
char c;
int i = 4;
try {
    if (i > s.size()-1) throw i;
    cout << s[i] << endl;
}
catch (int &ii) {
    cout << "Index out of bounds: " << ii << endl;
}
```

Beispiel: eigene Exception-Klasse

```
class illegal_name {
public:
    illegal_name(string name) throw() : name_(name) {}
    const string what() {return "Illegal_␣Name:␣"+name_ ;}
private:
    string name_;
};

int main () {
try {
    string name = "G.␣Köpke";
    if (...) throw illegal_name(name);
}
catch (illegal_name &n) {
    cout << n.what() << endl;
}
}
```

Exception-Probleme

- Keine Exceptions in Konstruktoren benutzen!
- Keine Exceptions in Destruktoren benutzen!
- Keine Exceptions in Templates benutzen!
- Vorzeitiges Verlassen einer Funktion: Mit `new` erzeugte Daten werden nicht gelöscht:

```
void foo () {  
    Person *p = new Staff("Reinhard_Zierke", 1958, 4711);  
    p->print();  
  
    if(...) throw(...);  
  
    delete p;  
}
```

auto_ptr

- auto_ptr verpackt Zeiger-Variable in ein Objekt; überlädt Operatoren wie * und ++ so, dass sie transparent benutzt werden können
- Variable ist nun ein Objekt: beim Verlassen der Funktion wird Destruktor aufgerufen

```
#include <memory>
```

```
void foo () {  
    auto_ptr<Person> p(new Staff("Reinhard_Zierke", 1958, 4711  
    p->print());
```

```
    if(...) throw(...);
```

```
    // delete p; // nicht mehr nötig
```

auto_ptr (2)

- auto_ptr nur mit Heap-Variablen erzeugen!
- Keine zwei auto_ptr mit der gleiche Heap-Variablen erzeugen!
- Vorsicht beim Kopieren von auto_ptr-Variablen
- auto_ptr-Variablen nicht in Container packen

Ein- und Ausgabe

- in C mit `printf()`, `scanf()`
- in C++ mit der Streams-Klassenhierarchie

Header	Klasse	Verwendung
<code><iostream></code>	<code>istream</code> <code>ostream</code> <code>iostream</code>	Eingabe Ausgabe Ein-/Ausgabe
<code><iomanip></code>		E/A-Formatierung
<code><fstream></code>	<code>ifstream</code> <code>ofstream</code> <code>fstream</code>	Datei-Eingabe Datei-Ausgabe Datei-Ein-/Ausgabe
<code><sstream></code>	<code>istringstream</code> <code>ostringstream</code> <code>stringstream</code>	Eingabe aus string Ausgabe in string String-Ein-/Ausgabe

Ausgabeformatierung (3)

- Beispiel Tabellenausgabe:

```
#include <iostream>
#include <iomanip>
int main () {
    cout << setw(8) << -128 << setw(8) << 127 << endl;
    cout << setw(8) << 0 << setw(8) << 255 << endl;
    cout << setw(8) << -32768 << setw(8) << 32767 << endl;
    cout << setw(8) << 0 << setw(8) << 65535 << endl;
}
```

Unformatierte Ausgabe

- einzelnes Zeichen ausgeben (out = beliebiger Ausgabekanal)

```
out.put(char c);
```

- n Zeichen ab gegebener Adresse ausgeben:

```
out.write(const char* a, int n);
```


Unformatierte Eingabe

- 1 Zeichen (in = beliebiger Eingabekanal)

```
int in.get();  
in.get(char& c);
```

- Maximal n Zeichen oder bis zum Trennzeichen

```
in.getline(char* str, int n, char trenn='\n');
```

- Binäre Eingabe

```
in.read(char* a, int n);
```

- nächstes Zeichen ansehen, aber nicht aus Datenstrom holen

```
int in.peek();
```

- Zeichen in Datenstrom zurückpacken

```
in.putback(char c);
```


Beispiel: Lesen bis zum Eingabeende

- Schlecht:

```
while (!cin.eof()) {  
    cin >> x;  
    cout << "Read:_" << x << endl;  
}
```

- Richtig:

```
cin >> x;  
while (!cin.eof()) {  
    cout << "Read:_" << x << endl;  
    cin >> x;  
}
```

Beispiel: Lesen mit Datenprüfung

```
template <typename T>
void readdata(istream &in, vector<T> items) {
    items.resize(0);
    T x;
    string junk; // to skip over bad data
    in >> x;
    while (!in.eof()) {
        if (in.fail()) {
            in.clear(); // clear error state
            in >> junk; // skip over junk
            cerr << "Skipping " << junk << endl;
        } else items.push_back(x);
        in >> x;
    }
}
```

Dateibearbeitung

- Dateibearbeitung mit Objekten der Klasse `fstream` / `ifstream` / `ofstream`
- Datei öffnen

```
ifstream fin;  
fin.open(char* filename);  
ifstream fin2(char* filename);
```

- Datei schließen

```
fin.close();
```

Dateibearbeitung (2)

- Beispiel: Datei zeichenweise lesen:

```
#include <fstream>
#include <iostream>
char c, filename[] = "/etc/motd";
ifstream fin(filename);
if (fin.fail()) {
    cerr << "Can't open file" << filename << endl;
    exit(1);
}
fin.get(c);
while (!fin.eof()) {
    cout << c;
    fin.get(c);
}
fin.close();
```

String-Ein-/Ausgabe

- String-E/A mit Objekten der Klassen `stringstream` und `ostringstream`
- Methoden und Operatoren wie bei anderen E/A-Klassen
- zusätzlich Umwandlung Stream-Objekt `ostr` in `String`:

```
string ostr.str();
```

- Beispiel

```
#include <sstream>
string toString(int n) {
    ostringstream ostr;
    ostr << n;
    return ostr.str();
}
```