SAARLAND UNIVERSITY
COMPUTER SCIENCE

# C/C++-Programmierung

## Schablonen, Ausnahmen, Debugging

Sebastian Hack
Christoph Mallon

(hack|mallon)@cs.uni-sb.de

Fachbereich Informatik
Universität des Saarlandes

Wintersemester 2009/2010

# Schablonen

## templates

```
#define MIN(a, b) ((a) < (b) ? (a) : (b)) // C

template <typename T> T min(T a, T b) // C++
{
  return a < b ? a : b;
}

// Typ (int) automatisch bestimmt
min(23, 42);
// Explizite Angabe, da verschiedene Typen
min<double>(23.42, 1);
```

- Algorithmus soll für verschiedene Typen einsetzbar sein
- C-Lösung: Makros – unübersichtlich, fehleranfällig (Mehrfachauswertung)
- C++: Typen von Klassen und Funktionen sind parametrisierbar
- Typen können automatisch bestimmt oder explizit angeben werden

# Spezialisierung, template mit Werten

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
template<unsigned N> struct Fac {
  static unsigned const i = Fac<N - 1>::i * N;
};
// Spezialisierung fuer 0
template<> struct Fac<0> {
  static unsigned const i = 1;
};

int main() {
  /* Keine Rekursion zur Laufzeit, Wert wird beim
   * Uebersetzen bestimmt */
  std::cout << Fac<10>::i << std::endl;
}
```

- Templateparamter können auch konstante Werte sein
- Kann man für funktionale Programmierung nutzen
  → Syntax allerdings ziemlich hässlich
- Spezialisierung: Erzeuge Variante für festgelegte Werte/Typen
  → Hier Basisfall der Fakultät

# Ausnahmen

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
#include <stdexcept>

try {
  throw runtime_error("something␣did␣not␣work");
}
catch (std::exception const& e) {
  // Faengt std::excpetion und Unterklassen
}
catch (...) { // Faengt alles
}
```

- Prüfen von Rückgabewerten ist mühsam und wird leicht vergessen
- Fehlerbehandlung stört Blick auf normalen Programmablauf
- Ausnahmen sind von normalen Programmablauf getrennt
- In C++ kann beliebiges geworfen werden: `throw 23;`
  $\rightarrow$ Vermeiden; spezielle Klassen für Ausnahmen verwenden

# finally?



```java
// Java
File f = new File("datei");
try {
  /* ... */
} catch (exception e) {
  /* Fehler behandeln */
} finally {
  f.close(); /* Aufraeumen */
}
```

- C++ hat kein `finally`
- Unnötig, denn Resourcenverwaltung mit RAII
  → Beim Verlassen des `try`-Blocks werden lokale Variablen zerstört

```cpp
// C++
try {
  File f("datei");
  // Datei wird automatisch geschlossen
} catch (std::exception const& e) {
  /* Fehler behandeln */
}
```

# Debugging



- printf()-Debugging: Verstreuen von Ausgaben im Programm
  $\longrightarrow$ mühselig, oft wenig zielgerichtet
- Debugger bieten Unterstützung zur systematischen Untersuchung des Programmzustands
- Beispiel hier: gdb
  $\longrightarrow$ Progrmm mit -g für Debugsymbole übersetzen

# Grundlegende Befehle von gdb

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- run, r: Programm starten
- continue, c: Ausführung fortsetzen
- print, p: Wert von Ausdruck ausgeben
- finish, fin: aktuelle Funktion bis zum Ende abarbeiten

# Haltepunkte

## break points



- Befehl: breakpoint, b
- Ausführung hält an, wenn Haltepunkt erreicht wird
- Ortsangabe entweder
  - Zeilennummer: 123
  - Dateiname und Zeilennummer: bla.c:123
  - Funktionsname: main
  - Dateiname und Funktionsname: bla.c:do_stuff
- Einmaliger Haltepunkt: advance, adv
  $\longrightarrow$ Ausführung wird sofort fortgesetzt

# Wertbeobachtung

## watch points



- Befehl: watch, w
- Anhalten, wenn sich der Wert eines Ausdrucks ändert
- Beispiel: w x
  $\longrightarrow$ hält an, wenn sich der Wert von x ändert
- Beispiel: w x == 0
  $\longrightarrow$ hält an, wenn sich Wahrheitswert ändert, also x 0 wird bzw. nicht mehr 0 ist