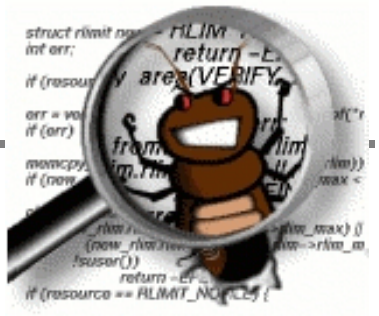
Fehler- und Ausnahmebehandlung

Was sind Software „bugs“



- Syntaktische Fehler
  - Falsche Schreibweise
  - Fehlendes Semikolon
  - Lösung durch den Compiler - zwar lästig aber relativ vollständig

Schwerwiegender sind:

- Semantische Fehler
  - Fehler die während des Programmlaufs auftreten
  - Im einfachen Fall: „Segmentation faults“ (Allg. Schutzverletzung)
  - Oder aber Programmergebnis verfälschen
  - Oft auch noch schlecht reproduzierbar (nur unter Last, nach langer Laufzeit,...)

Typische Fehlerquellen

- Designfehler
  - Fehlende Klammer
  - Vergessenes break
  - Semikolon zu viel
- Programm ist zwar syntaktisch korrekt aber führt in der Regel nicht zum gewünschten Ergebnis
- Ist der Fehler aber erst gefunden...

- Hilfsmittel:
  - Verfolgung des Programmablaufs durch:
    - Log
    - Debugger

```
if (result == 0)
  text = "Hier ist ein Fehler!“;
  cerr << text << endl;
```

```
switch (i) {
case 4:
        text = "gerade“;
Case 3:
        text = "ungerade";
        break;
```

```
for (k=0; k<n; k++) ;
   x += k;
```

Typische Fehlerquellen

- Datenveränderung
  - Überschrieben von Speicherbereichen
  - Überschreiten von Feldgrenzen (Index)
  - Überlauf von Variablen
    - Char text = 312;
  - Falsche Zeiger
- Bedingte Kompilierung
  - Fehlendes #endif
  - Falsche Makros

```
#ifdef DEBUG
  cout "Schleifenstart :!"<< endl;
#endif


//normaler code




#ifdef DEBUG


#endif
```

Typische Fehlerquellen

- Speicher-Fehler
  - Freigeben von bereits freien Speicherbereichen
  - Reservierter Speicher wird nicht freigegeben
    - Temporäre Pointer Variable in Funktion unter Verwendung von new()
  - Unzureichende Reservierung von Speicher für Zeichenketten
  - Überschreitung von Feldgrenzen
  - Nicht initialisierte Variable oder Zeiger

Wie findet man Fehler / Bugs

- Statusausgaben im Code
  - "Log-" oder "Trace-"File
- Im einfachsten Fall:
  - Direkte Ausgabe auf
    - cout aber mit flush, oder
    - Cerr
- Komplexere Ausgabe zum Beispiel
  - Eigene LOG-Funktion
    - Ausgabe in Datei
    - Ausgabesteuerung durch Log-Klassen
    - oder LOG-Level

```
cout << "Schritt 1 fertig" << endl
     << flush;

cerr << "Schritt 1 fertig" << endl;
```

```
LogFile ("IO" , 2 , "Datei geöffnet");

LogFile ("COM" , 1 , "Ausgabefehler");
```

Wie findet man Fehler / Bugs

- PRE-Prozessor Hilfmittel
  - Über Makros

```
#define LOGFILE ( x )   LogFile ( __FILE__ , __LINE__ , x )

LOGFILE ("Fehler")  à führt zu

LogFile ("Datei.ccp" , 125 , "Fehler" );
```

  - Oder Übersetzung nur im DEBUG Fall

```
#ifdef DEBUG
#define LOGFILE( x )   LogFile ( __FILE__ , __LINE__ , x )
inline void LogFile (char * file, int line, char *text)
{ cout << "** " << file << "(" << line << ")" << text << endl
<< flush; }
#else
#define LOGFILE(X)
#endif
```

Wie findet man Fehler / Bugs

```
#ifdef DEBUG
#define LOGFILE( x )   LogFile ( __FILE__ , __LINE__ , x )
inline void LogFile (char * file, int line, char *text)
{ cout << "** " << file << "(" << line << ")" << text << endl <<
flush; }
#else
#define LOGFILE(X)
#endif


int main()
{
        int x = 5;

        LOGFILE("Schleife");
        for (int i=0 ; i < x ; i++)
        {
                cout << "Hallo" << endl;
        }
        LOGFILE("Nach Schleife");

        return 1;
}
```

```
** F:\fh\ss2005\Vorlesung\log\main.cpp(18)Schleife
Hallo
Hallo
Hallo
Hallo
Hallo
** F:\fh\ss2005\Vorlesung\log\main.cpp(23)Nach Schleife
Press any key to continue
```

```
Hallo
Hallo
Hallo
Hallo
Hallo
Press any key to continue
```

Fehler in Funktionen und Modulen

- Bisher war die Fehlersuche beschränkt auf das nachvollziehen des Programmablaufs durch Debug - "Ausgabe"
- Was aber wenn die Funktionsfähigkeit einer Funktion nicht vom eigentlichen Coding sondern auch von der Eingabe von Werten
  - (d.h. den Übergabewerten)
- Abhängt.

- Zur Erinnerung:

  "Funktionen schließen einen Vertrag mit dem Benutzer. Sie sicheren eine Leistung zu, erwarten aber auch das Einhalten bestimmter Bedingungen"

  Siehe "Header-Datei"

Kapitel 9 : Wichtige Grundregeln für den Programmierstil

- Wer Funktionen erstellt muss dem "Verwender" mitteilen was die Funktion kann
  - rein technisch durch den Prototyp
    d.h. in der Header Datei
- ….und welche Voraussetzungen und Bedingungen die Funktion hat.
  - Dokumentation oder/und
  - Kommentare im Quelltext (Header und in der eigenen Funktionsdatei)

```
int fakultaet ( intx) ;
// berechnet die Fakultät von x, d.h. x!
// PRE:  x > 0   &&
//          x klein genug, damit das Ergebnis
//          noch in int passt
// POST: Ergebnis ist die Fakultaet von x
//          &&  Ergebnis >= 1
```

```
/*********************************************
**  void swap ( int &x, int &y)
**  Change the values of the two input parameters
**  Input:            int &y, int &y.
**                    Caution referenz
**  Return:           nothing
**  Precondition:     none
**  Side effects :
**  Version :         1.0
**  Author:           your name
*********************************************/
void swap (int &x, int &y);
```

Wie meldet die Funktion "Fehler"

- Z.B. fehlerhaft Übergabewerte
- Fehlfunktionen innerhalb der Funktion
  - Datei lässt sich nicht öffnen
  - Speicher lässt sich nicht allokieren
  - Berechnungen sind Fehlerhaft (div. by zero)
  - ....
- Welche Entscheidungen können getroffen werden, wenn die Funktion ihren "Vertrag" nicht einhalten kann?

Klassisches Verfahren: Rückgabewert == Fehlercode

- Eine Möglichkeit zur Information des Benutzers ist es den Rückgabewert als Fehlercode zu interpretieren.
- Klassisches Verfahren bei Standard-Funktionen

```
int  Funktion (...)
 return 0 ; // Fehler
```

  - Z.B. malloc für Speicheranforderung

  Die Verantwortung wird an den Benutzer weitergereicht.
- Die Frage ist: "Ist es bei allen Fehlern sinnvoll weiter zu arbeiten?"
  - Oft ist dies nicht sinnvoll!

Sicheres Beenden in Ausnahmesituationen assert()

- Falls die Weiterverarbeitung nicht sinnvoll ist:
  - Verwendung von assert()
- Prüft Bedingung und bricht ab falls "FALSE"

```
include <cassert>

for (int i=0 ; i < x ; i++)
    {
        cout << "Hallo" << endl;
        assert (i<3);
    }
```

Kann durch Compiler Schalter NDEBUG deaktiviert werden (#define NDEBUG)

Weiterreichen oder Abbruch try / catch

- Ein Programmteil versucht eine Funktion aufzurufen. Dieser Versuch wird eingeleitet durch das Schlüsselwort: **try**
- Die Funktion löst bei einem Fehler einen Ausnahme aus. "Wirft eine Ausnahme" Schlüsselwort **throw**
- Der Aufrufer kann nach dem try Block versuchen die Ausnahme zu bearbeiten. D.h. sie aufzufangen. Schlüsselwort **catch**.

```
try
{
   //// irgendeinen Funktion
   funktion();
}
catch (...)
{
   /// Fehler bearbeiten
}
```

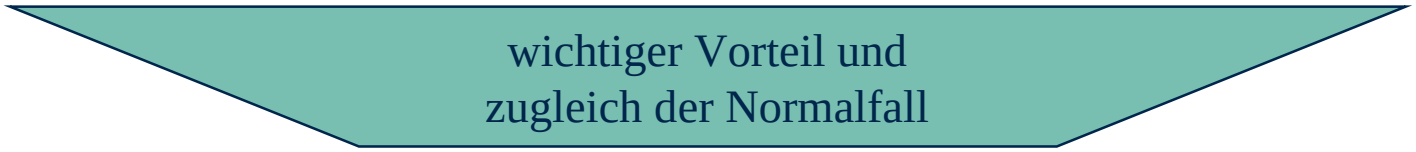
```
Funktion ()
{

  //// Fehler tritt auf
  throw ("Fehler");

 }
```

Ausnahmebehandlung über Funktionsgrenzen

- throw-Anweisung erzeugt ein Ausnahmeobjekt
  - enthält Fehlermeldung und/oder Daten zum Wiederaufsetzen
- das Ausnahmeobjekt wird in der Aufrufhierarchie zur Fehlerbehandlung weitergereicht
  - jede Funktion, die kein passendes catch enthält, wird ordentlich beendet
  - dann wird in der Aufrufumgebung weiter nach einem catch gesucht
- damit Fehlerbehandlung über mehrere Aufrufebenen hinweg
  - Bibliotheksfunktion erkennt Fehler, kann sie aber nicht behandeln
  - Aufrufumgebung kann Fehler behandeln, aber nicht selbst erkennen



Auswerfen einer Ausnahme in einer Funktion

- if-Abfrage auf Fehlerbedingung mit throw-Anweisung
- keine passende catch-Anweisung
- automatisches "stack unwinding": beim Verlassen von Blöcken werden lokale Objekte freigegeben.
- Ausnahmeobjekt muß kopierbar sein (Copy-Konstruktor)
  - dynamische Allokation möglich, aber nicht zu empfehlen

```
void eineFunktion (int
param)
  throw (char*, CMath)
{
  if (FehlerAufgetreten)
    throw "Meldung";

  if (andererFehler)
    throw CMath (Wert);

}
```

Auffangen in der Aufrufumgebung

in Java ein Muß

- erster passender catch-Block wird ausgeführt
  - Auswahl anhand des Datentyps des Ausnahmeobjekts
- Typumwandlung in Basisklasse oder Zeigerumwandlung wird ggfs. vorgenommen
  - deshalb abgeleitete Klasse vor Basisklasse aufführen
- catch (...) fängt alles ab
  - wenn, dann als letzter Block
- Der Fehler kann aus dem catch Block an einen anderen handler weitergereicht werden
  - throw;
- Programmabbruch falls kein passender catch-Block existiert

```
try {
  eineFunktion (27);
  Funktion2 (5);
}
catch (const char* Text)
{
  cout << Text;
}
catch (CMath* Ursache)
{
  // Ursache auswerten
  delete Ursache;
}
catch (...) {
  throw;
}
```

Erweiterung der Funktionsdeklaration

- Funktionsschnittstelle wird um Deklaration der möglichen Ausnahmen erweitert
  - keine Angabe: die Funktion kann beliebige Ausnahmen auswerfen
  - leere Liste: die Funktion wirft garantiert keine Ausnahme aus
- vollständige Beschreibung einer Funktionsschnittstelle:

Ergebnistyp, Name, Parameter, Ausnahmen,
PRE, POST, Kommentar

```
void Funktion1 (int param)
  throw (char*, CMath, CMath*)
{
}
void Funktion2 (int param)
  // beliebige Ausnahmen !!!
{
}
void Funktion3 (int param)
  throw () // garantiert keine
{
}
```