Programmieren / Algorithmen & Datenstrukturen

Grundlagen (i), Teil 4

Prof. Dr. Skroch

Universitatea
BABEŞ-BOLYAI

Grundlagen (i)

Inhalt.

- Hallo C++
- Objekte, Typen, Werte, und Steuerungsprimitive
- Berechnungen und Anweisungen
- Fehler
- Fallstudie: Taschenrechner
- Funktionen und Programmstruktur
- Klassen

Fehler

"My guess is that avoiding, finding, and correcting errors is *95% or more* of the effort for serious software development" (B. Stroustrup)

'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe;
All mimsy were the borogoves,
And the mome raths outgrabe.

Beware the Jabberwock, my son!
The jaws that bite, the claws that catch!
Beware the Jubjub bird, and shun
The frumious Bandersnatch!

o.A. (1992), *Alice's Adventures in Wonderland and Through the Looking-Glass by Lewis Carrol, With illustrations by John Tenniel, in one volume*. Yearling Classics, Dell Publishing, New York, New York, USA: ii/22–ii/24.

Fehler

Beim Programmieren ist es wie sonst auch: man macht Fehler.

- Entscheidend ist (wie sonst auch), wie man mit den Fehlern umgeht.
  - Das Auftreten von Fehlern durch systematische und überlegte Vorgehensweisen von Anfang an minimieren.
  - So viele der trotzdem aufgetretenen Fehler wie möglich finden und eliminieren.
  - Sicherstellen, dass die verbliebenen Fehler nicht zu schwerwiegend sind.
- Sehr häufige Fehlerquellen in Programmen:
  - Falsche Funktionsaufrufe und –rückgaben.
    ```
    double d { sqrt( -1 ) };
    ```
  - Speicherbereichs-Verletzungen.
    ```
    myVector[-3] = x;
    ```
  - Unzulässige Eingaben.
    ```
    char c {'y'}; while( c != 'n' ) cin >> c;
    ```
  - "Narrow casts".
    ```
    int i {20000}; char c {i};
    ```

Ziel: das möglichst fehlerfreie Programm

Richtlinien für den minimal erwarteten Umgang mit Fehlern in Ihren Programme im Rahmen dieser Lehrveranstaltung.

- Der Umgang mit Fehlern ist eine der schwierigsten Aufgaben beim Programmieren.
- Gerade deshalb fangen wir so früh wie möglich an, uns damit zu beschäftigen.
- Ihre Programme sollen im Rahmen dieser Lehrveranstaltungen
  1. *für alle zulässigen Eingaben die gewünschten Ergebnisse erzeugen,*
  2. *für unzulässigen Eingaben vernünftige Fehlermeldungen ausgeben,*
  3. *sich kontrolliert beenden, wenn ein Fehler auftritt.*
  4. Unerwartetes Verhalten der Hardware und der Systemsoftware (Betriebssystem usw.) braucht nicht berücksichtigt werden.

Fehlerarten

Es gibt viele Möglichkeiten, Fehler zu klassifizieren, z.B. danach, wann sie gefunden werden.

- Build-time (Übersetzungsfehler)
  - Vom Compiler gefunden (compile-time).
    - Syntaxfehler, Typfehler.
  - Vom Linker gefunden (link-time), wenn er versucht, die Objektdateien zu einem ausführbaren Programm zu verbinden (etwa inkonsistente Deklarationen).
- Run-time (Laufzeitfehler)
  - Vom Computer gefunden (Hardware oder Betriebssystem).
  - Von einer Bibliothek gefunden (wie Exceptions).
  - Vom Quellcode selbst gefunden.
- Application-time
  - I.Allg. vom Benutzer gefunden: das Programm läuft zwar, macht aber nicht das, was es eigentlich machen soll.

Fehler beim Kompilieren

Entspricht das Programm vollständig der Sprachspezifikation?

- Quellcode einer einfachen Funktion:

```
int area( int length, int width ) {
    return length*width;
}
```

- Beispiel für Syntaxfehler:

```
do area( :8 :13 ) end;     // nicht C++ Syntax
```

- Beispiele für Typfehler:

```
int x1 { area( 7 ) };          // falsche Anzahl von Parametern
int x2 { area( "5", 8 ) };     // falscher Typ des ersten Parameters
int x3 { area( 21, 34 ) };     // ok

int x4 { area( 8, 12.9 ) };    // Achtung: schneidet 12.9 zu 12 ab
                               // viele Compiler warnen

int x5 { area( 55, -89 ) };    // Typen korrekt, aber
                               // es macht keinen Sinn
```

Fehler beim Linken

Passen mehrere separat kompilierte Übersetzungseinheiten zusammen?

- Jede Funktion eines Programms muss in jeder Übersetzungseinheit (Datei), in der sie benutzt wird, identisch *deklariert* sein.
  - Um dies sicher zu stellen werden i.Allg. Headerdateien verwendet.
- Jede Funktion muss genau einmal in einem Programm *definiert* sein.
- Die beiden Regeln gelten nicht nur für Funktionen sondern auch für alle anderen Entitäten eines Programms.
  - Wie Variablen, Typen, ...
  - Sog. *one definition rule*.
- Der Linker überprüft (neben seinen anderen Aufgaben) die Einhaltung dieser Regeln.

Laufzeitfehler

Probieren Sie aus, was passiert, wenn Sie die u.g. Quellcodefragmente in einem Programm verwenden und dieses ausführen.

- Quellcode einer einfachen Funktion:

```
int div( int z, int n ) {
    return z/n;
}
```

- Aufruf z.B.:

```
for( int i {-9}; i < 10; ++i )
    cout << i << '\t' << div( 100, i ) << endl;
```

- Ein typisches Beispiel für einen Laufzeitfehler...

Laufzeitfehler: Fehlerquelle Funktionsaufruf

Wie kann man Laufzeitfehler vorsorglich vermeiden?

- Möglichkeiten zur vorsorglichen Vermeidung von Laufzeitfehlern:
  - Nur Quellcode schreiben, der nie zu Laufzeitfehlern führt.
    - Schafft man in realen Programmen nicht.
  - Beim Aufruf einer Funktion den *aufrufenden* Programmteil sicher stellen lassen, dass die Funktion keine ungültigen Aufrufparameter erhält.
    - Bei Funktionen, deren Quellcode wir nicht kennen (oder nicht kennen wollen), die einzige Alternative.
  - Bei der Definition einer Funktion sicher stellen, dass diese, wenn sie *aufgerufen* wird, sich selbst vor ungültigen Aufrufparametern schützt und diese nicht verarbeitet.
    - Sprechen keine besonderen Gründe dagegen, sollte jede Funktion so programmiert werden, dass sie ihre Aufrufparameter immer selbst prüft.

Fehlerquelle Funktionsaufruf

Funktionsaufrufe sind eine entscheidende, mögliche Fehlerquelle in Programmen.

- Bedeutung von Prüfungen bei Funktionsaufrufen:
  - Funktionen sind (neben den Klassen) das Hauptmittel in C++, um Programme in logisch separate Einheiten zu strukturieren.
    - Ein Funktionsaufruf kann als der Übergang zu einem logisch separaten Programmteil angesehen werden.
  - Üblicherweise hat der Programmierer einer Funktion keine Kontrolle über die Art, wie eine von ihm definierte Funktion später von anderen Programmierern verwendet wird.
  - Beschreibungen oder Kommentare helfen nur bedingt: werden oft nicht gelesen!
  - Daher sollte sich jede Funktion auch selbst auf mögliche Fehler prüfen.
    - Etwa bevor die eigentlichen Berechnungen erfolgen oder bevor ein Ergebnis zurückgegeben wird.
- Funktionen in Ihren Programmen:
  - Definieren Sie Ihre Funktionen immer sorgfältig mit solchen Prüfungen.
  - Wenn Sie das nicht für jede Funktion schaffen, nehmen Sie sich zumindest alle wichtigen vor.

Fehlerquelle Funktionsaufruf

Vor- und Nachbedingungen von Funktionen sind elementare Hilfsmittel zur Sicherstellung korrekten Quellcodes.

- Beim Aufruf einer Funktion: *Vorbedingungen* (pre-conditions)
  - Müssen unmittelbar vor dem Aufruf einer Funktion erfüllt sein, damit die Funktion ihre Berechnungen korrekt ausführt.
  - Betreffen v.a. die übergebenen Aufrufparameter, an die die Funktion bestimmte Anforderungen hat.
- Am Ende einer Funktion: *Nachbedingungen* (post-conditions)
  - Müssen unmittelbar nach dem Ablauf einer Funktionen erfüllt sein.
  - Betreffen v.a. den Rückgabewert, den man von der Funktion bei korrektem Aufruf erwartet.

- Die Prüfung von Vor- und Nachbedingungen kann kompliziert werden.

Fehlerquelle Funktionsaufruf

Vor- und Nachbedingungen von Funktionen sind elementare Hilfsmittel zur Sicherstellung korrekten Quellcodes.

- Sie sollten eigene Vor- und Nachbedingungen von Funktionen in Kommentaren dokumentieren.
- Beispiel:

```
int complicatedFunction( int a, int b, int c ) {
   // PRECONDITION: a,b,c alle positiv und a < b < c
   if( !(0<a && a<b && b<c) ) //…
       // Fehler, cF() kann nicht laufen
       // Vorbedingung verletzt

   // Quellcode berechnet Ergebnis r

   // POSTCONDITION: Ergebnis r der Berechnung kleiner als -c
   if( !(r < -c) ) //…
       // Fehler, cF() Ergebnis kann nicht stimmen
       // Nachbedingung verletzt

   return r;
}
```

Laufzeitfehler in Funktionen

Wie kann man eine Funktion so definieren, dass sie mit möglichen Laufzeitfehlern vernünftig umgeht?

- Besondere Fehlerwerte zurückgeben, um die sich der aufrufende Programmteil kümmern muss.

```
int area( int length, int width ) {
    if( length <= 0 || width <= 0 ) return -1;
    return length*width;
}

int z { area( x,y ) };
if( z == -1 ) cerr << "Fehler: Area-Berechnung gescheitert!";
// …
```

- Die Rückgabe von Fehlerwerten ist aber wenig empfehlenswert.
  - Manchmal ist kein Fehlerwert möglich, z.B. `max( )`.
  - Was passiert, wenn vergessen wird, das Funktionsergebnis auf den Fehlerwert zu prüfen?

Laufzeitfehler in Funktionen

Wie kann man eine Funktion so definieren, dass sie mit möglichen Laufzeitfehlern vernünftig umgeht?

- Einen Wert in eine global gültige Fehlerstatus-Variable setzen, um die sich der aufrufende Programmteil kümmern muss ("C-Stil").

```
int errno {0}; // globale Variable mit Fehlernummer

int area( int length, int width ) {
   if( length <= 0 || width <= 0 ) errno = -7;
   return length*width;
}

int z { area( x,y ) };
if( errno == -7 ) cerr << "Fehler: Area-Berechnung gescheitert!";
// …
```

- Ein globaler Fehlerstatus ist aber wenig empfehlenswert.
  - Verletzung des Grundprinzips "keine globalen Variablen", denn:
    - Welche Werte kann/darf `errno` annehmen? Ist der Fehlerzustand auch in sehr großen Programmen noch mit vertretbarem Aufwand nachvollziehbar?
  - Was passiert, wenn vergessen wird, `errno` zu prüfen?

Grundgedanke der Ausnahmen (Exceptions)

Ausnahmen sind ein Mechanismus zur Behandlung von Fehlern in Programmen, dessen Grundidee es schon in den 1970er Jahren (vor C++) gab.

- Eine Funktion, die einen Fehler feststellt, den sie nicht selbst behandelt, gibt *nicht* auf normalem Weg mit `return` ein Ergebnis zurück.
- Statt dessen löst diese Funktion – in C++ mit dem Schlüsselwort `throw` – eine sog. *Ausnahme* aus, die anzeigt, dass etwas fehlgeschlagen ist.
  - Üblicherweise durch Erzeugung eines anonymen Objekts von bestimmtem Typ (Ausnahmetyp).
- Eine Funktion, die Ausnahmen auslösen ("werfen") kann, wird in C++ in einem sog. `try` Block aufgerufen.
  - Die Ausnahmen können dann in C++ in den sog. `catch` Teilen ("Ausnahme-Handler") des `try` Blocks behandelt werden.
    - Für praktisch jede Art von Fehler kann man einen spezifischen Handler schreiben, der ihn "fängt" (und ggf. weiter verarbeitet).
- Das gesamte C++ Programm terminiert, wenn eine im `try` Block mit `throw` angezeigte Ausnahme nicht mit `catch` behandelt wird.
  - Die aufrufende Stelle kann daher eine Ausnahme nicht mehr ignorieren oder vergessen.

Fehler mit C++ Ausnahmen

Falsche Funktionsaufrufe und -rückgaben: nur gültige Werte in Aufrufparametern und als Rückgabewerte zulassen.

- Jede Funktion soll ihre eigentliche Ausführung nur zulassen, wenn ihre Aufrufparameter-Werte gültig sind (analog für Rückgabewerte).

```
int div( int z, int n ) {
    if( n == 0 )                          // die Ausnahme melden (hier durch ein
        throw string{"zero divisor"}; // anonymes string-Objekt)
    return z/n;
}
```

- Den Fehler "fangen" und etwas damit tun, z.B. in `main( )`.

```
int main( )
    try {
       for( int i {-9}; i < 10; ++i )
          cout << i << '\t' << div( 100, i ) << endl;
       return 0;
    }
    catch( string& s ) {
       cerr << s; // Meldung und Programmende
       return -1;
    }
```

Fehler mit C++ Ausnahmen

Um Ausnahmen anzuzeigen verwenden wir einen der in der C++ StdLib zur Verfügung stehenden Ausnahmetypen.

- Der `try` / `catch` Mechanismus unterscheidet *beliebige* Typen der anonymen Objekte, die von einem `throw` erzeugt wurden.
- Um eine Ausnahme zu melden setzt man besondere, speziell für diesen Zweck definierte Typen ein.
  - In der StdLib gibt es vordefinierte Ausnahmetypen (Header `stdexcept`).
  - Beispielsweise den Typ `std::runtime_error`.

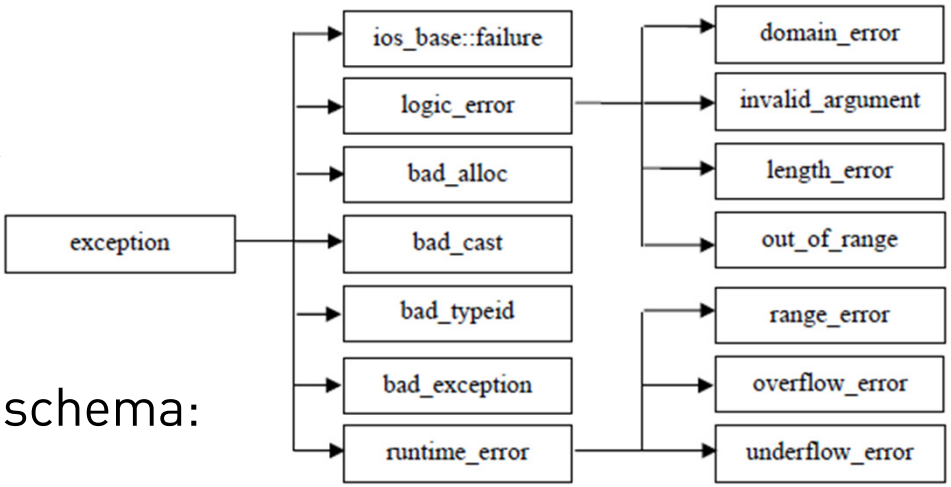
```
int div( int z, int n ) {
    if( n == 0 ) throw std::runtime_error{ "div(), zero divisor" };
    return z/n;
}

int main( )
    try {
        div( 100, 0 );
        return 0;
    }
    catch( std::runtime_error& re ) {
        cerr << re.what(); // Alle StdLib Ausnahmetypen kennen eine what()
                           // Operation zur Fehlermeldungs-Ausgabe.
        return -1;
    }
```

C++ Standard-Ausnahmetypen

Die Hierarchie der C++ Ausnahmetypen aus der StdLib, und ein allgemeines Quellcodeschema.

- Die Ausnahmetypen der StdLib bilden eine hierarchische Struktur mit `exception` als Grundlage.



- Es ergibt sich folgendes Quellcodeschema:

```
int main( ) try {
    // …
    div( 100, 0 ); // z.B.
    // …
    return 0;
}
catch( std::exception& e ) { // alle Ausnahmen der StdLib
    cerr << "Ausnahme: " << e.what();
    return -2;
}
catch( ... ) { // andere Ausnahmen beliebigen Typs ("catch all")
    cerr << "Unbekannte Ausnahme";
    return -1;
}
```

C++ Standard-Ausnahmetypen

Die in der StdLib vordefinierten Ausnahmetypen.

- Die C++ StdLib stellt vordefinierte Ausnahmetypen bereit.
- Diese bilden eine *Hierarchie* mit `exception` als gemeinsamen Basistyp, von dem alle weiteren Ausnahmetypen abgeleitet sind.
  - `exception` beinhaltet z.B. auch `runtime_error`.
- Alle C++ Standard-Ausnahmen enthalten eine Memberfunktion `what()`.
  - So kann eine Fehlermeldung ausgegeben werden.
- Mit `catch( ... )` werden alle Ausnahmen abgefangen, die vorher noch nicht behandelt wurden ("catch all").
- Ausblick

  `catch( exception e )` Wertübergabe vs.
  `catch( exception& e )` Referenzübergabe

  - Bei Referenzübergabe einer ausgelösten Ausnahme vom abgeleiteten Typ wird beim Fangen des Basistyps der *abgeleitete* Typ zur Verfügung stehen (d.h mehr Informationen) .
  - Wird z.B. ein `runtime_error` (der von `exception` abgeleitet ist) geworfen und als Referenz `exception&` gefangen, stehen alle Informationen des `runtime_error` Typs zur Verfügung, wird der `runtime_error` aber als Wert `exception` gefangen, stehen nur die Informationen des `exception` Typs zur Verfügung.

Ausblick: eigene Ausnahmetypen

Man kann auch eigene Typen konstruieren, die man zur Meldung einer Ausnahme verwenden will.

- Man könnte sich seinen eigenen Typ für eine Ausnahme wünschen...

```
namespace my_div {
   class X_div { };       // Eine Klasse (class) beschreibt einen benutzerdefinierter Typ.
                          // Der Typ X_div wird hier genau deshalb definiert,
                          // damit die Erzeugung eines Objekts vom Typ X_div
                          // einen Fehler der Funktion div() anzeigen kann.

   int div( int z, int n ) {
      if( n == 0 ) throw X_div{ }; // Ausnahme melden (anonymes X_div Objekt).
      return z/n;
   }
}

int main( ) try {
    my_div::div( 100, 0 );
    return 0;
}
catch( my_div::X_div& ) { // X_div Ausnahme aus dem namespace my_div.
    cerr << "X_div exception"; return -3;
}
catch( std::exception& e ) { // StdLib Ausnahmen.
    cerr << e.what(); return -2;
}
catch( ... ) { // "catch all", alle Ausnahmen.
    cerr << "unknown exception"; return -1;
}
```

Fehler mit C++ Ausnahmen

Bereichsverletzungen: Zugriffe nur auf gültige Speicherbereiche erlauben, Beispiel, `std::vector` Index auf gültigen Bereich prüfen.

- Der `std::vector` Typ kennt die eigene Größe und kann damit Zugriffe auf Bereichsverletzung prüfen.
- Bei Bereichsverletzung wirft `std::vector::at()` eine Ausnahme vom Typ `std::out_of_range`.
  - Die Indexoperation `std::vector::operator[]` wirft dagegen keine Ausnahme.
- Quellcode-Beispiel:

```
int main( ) try {
     vector<int> v{}; int x{};
     while( cin >> x ) v.push_back( x );
     for( unsigned int i{}; i <= v.size(); ++i )
          cout << "v[" << i <<"] ist " << v.at(i) << endl;
     return 0;
}
catch( exception& e ) {
     cerr << e.what;
     return -2;
}
catch( ... ) {
  //...
}
```

Fehler mit C++ Ausnahmen

Unzulässige Eingaben: **Überprüfung von `std::cin` und Meldung, ggf. durch eine einfache `error()` Funktion.**

- Grundgedanke der `error()` Funktion: `throw` "maskieren".
- Wie die `error()` Funktion aussehen kann:

```
void error( string s ) { throw runtime_error{ s }; }
```

  - Header `stdexcept` für den Typ `runtime_error` erforderlich.

- Quellcode-Beispiel:

```
try {
    int i{};
    cin >> i;
    if( !cin ) error( "Fehler beim Einlesen" );
    return 0;
}
catch( exception& e ) {
    cerr << e.what();
    return -2;
}
catch( ... ) {
   //...
}
```

Fehler mit C++ Ausnahmen

"Narrow cast": explizite Überprüfung im Quellcode.

- Wird einer Variablen ein Wert zugewiesen, der für diese Variable "zu groß" ist, wird er implizit abgeschnitten, z.B.:

```
int i {2.95};    // i wird 2
char c {20000};  // Was wird c auf Ihrem System?
```

- Prüfen Sie den Wert, bevor Sie ihn zuweisen.
  - Quellcode-Beispiel:

```
try {
  int i {20000};
  char c {i}; // narrow cast ("Einengung")
  int i2 {c};
  if( i2 != i ) error( "int passt nicht ins Zielobjekt!\n" );
  cout << "Ein char-Wert von 20000!" << endl;
  return 0;
}
//...
```

Fehler erkennen vs. Fehler behandeln

Ausnahmen sind ein geeigneter Mechanismus, um Fehler abzufangen, der Umgang mit den dann abgefangenen Fehlern ist oft schwierig.

- Immer noch kompliziert: wie wird mit den ausgelösten Ausnahmen im aufrufenden Programmteil weiter umgegangen?
  - Fehlerbehandlung ist fast immer nicht ganz einfach.
- Im Rahmen dieser Lehrveranstaltungen:
  - Definieren Sie Ihre Funktionen so, dass sie wie besprochen auf Fehler prüfen (*Erkennung* von Fehlern) und diese als C++ Ausnahmen melden (`try`-Block und `throw`).
  - Definieren Sie *keine weitergehende Behandlung* von Fehlern, d.h. verwenden Sie die Ausnahmen-Handler (`catch`-Blöcke) nur, um ein Programm mit sinnvoller Fehlermeldung kontrolliert zu beenden, z.B.:

```
int main( ) try {
     // …
     return 0;
}
catch( std::exception& e ) {
     cerr << e.what(); return -2;
}
catch( ... ) {
     cerr << "Unbekannte Ausnahme"; return -1;
}
```

Übung

Eine `error()` Funktion in einem Header, und einen Wert sicher einlesen.

- Bauen Sie sich Ihre erste eigene Headerdatei `myerror.h` mit der folgenden überladenen `error()` Funktion:

#include-Wächter

```
#ifndef MY_ERRHEADER
#define MY_ERRHEADER
    #include <stdexcept>
    #include <string>
    using std::string;
    void error( string s ) { throw std::runtime_error{ s }; }
    void error( string s1, string s2 ) { error( s1 + s2 ); }
#endif
```

- Schreiben Sie nun ein Programm, das `error()` verwendet, um eine ganze Zahl aus dem Bereich $[1, \ldots, 10]$ mit `std::cin >>` sicher einzulesen.
  - Binden Sie zunächst Ihren Header ein.
  - Geben Sie eine Eingabeaufforderung aus.
  - Prüfen Sie, ob der Eingabestrom `cin` Fehler enthält (andernfalls: Programmabbruch mit Fehlermeldung, verwenden Sie `error` aus Ihrem Header).
  - Prüfen Sie dann, ob die eingelesene Zahl im gültigen Bereich ist (andernfalls: erneute Eingabeaufforderung).
  - Geben Sie die eingelesene Zahl aus.

LEERE SEITE

Debuggen

Einige praktische Hinweise zum defensiven Programmieren und "Debuggen", dem Suchen und Entfernen von Fehlern im Quellcode.

- Verwenden Sie wenn möglich Bibliothekselemente der StdLib statt eigenen Quellcode zu schreiben.
- Gliedern Sie Ihre Programme in relativ kleine, logisch zusammen gehörende Funktionen.
  - *Die meisten Funktionen sind viel kürzer als 60 Zeilen.*
- Vermeiden Sie, wenn irgendwie möglich, unnötig komplizierten Quellcode.
  - Verschachtelte `if` Anweisungen, komplizierte Bedingungen, ...
- Machen Sie sich selbstständig mit dem Debugger Ihrer IDE und den dort zur Verfügung stehenden Möglichkeiten (break points usw.) vertraut.
- Instrumentieren Sie ggf. Ihren Quellcode zum Debuggen.
  - Vergessen Sie nicht, die Instrumentierungen später auch wieder zu entfernen.
- Kommentieren Sie Ihren Quellcode an komplizierten Stellen.
- Verwenden Sie aussagekräftige, nicht zu lange Namen.
- Verwenden Sie ein gut strukturiertes, einheitliches Layout für Ihren Quellcode.

Testen

Testen beinhaltet Testfallermittlung, Testdatenerzeugung, Testdurchführung und Testergebnis-Auswertung.

- Testfallermittlung
  - Testfälle beantworten die Frage, was genau getestet werden soll.
  - Woher kommen die Testfälle?
    - Aus dem Kontrollfluss des Programms (Überdeckungsmaße).
    - Aus den Ein- und Ausgaben des Programms (Äquivalenzklassenanalysen).
    - Aus Vor- und Nachbedingungen und Invarianten des Programms und seiner Teile.
    - Aus aufgetretenen Fehlern.
    - Aus Überlegungen und Analysen zum Programm.
    - Aus Überlegungen und Analysen zu den Anforderungen.
    - ...

Testen

Testen beinhaltet Testfallermittlung, Testdatenerzeugung, Testdurchführung und Testergebnis-Auswertung.

- Testdatenerzeugung
  - Die Daten, die den Systemzustand zum Testbeginn festlegen.
  - Die Eingabedaten, die für den Testablauf selbst benötigt werden.
  - Die erwarteten Ergebnisse ("Testorakel").
- Testdurchführung
  - Kontrolliert (d.h. wiederholbar, gemessen).
- Testergebnis-Auswertung
  - Schlussfolgerungen aus den Tests.
  - Problembereich Testorakel.

Qualitätssicherung

Man kann konstruktive und analytische Qualitätssicherung unterscheiden.

- Konstruktive Qualitätssicherung
  - Einsatz von Methoden und Techniken zur *Vermeidung* von Fehlern *vorbeugende* Qualitätssicherung.
  - Einhaltung von Standards und Richtlinien.
  - Ingenieurmäßige, systematische Softwareentwicklung (Vorgehensmodelle usw.).
  - Einheitliche Quellcode-Erstellung.
  - ...
- Analytische Qualitätssicherung
  - Einsatz von Methoden und Techniken zur *Erkennung* von Fehler *prüfende* Qualitätssicherung.
  - Prüf- und Kontrollmaßnahmen.
  - Analyse- und Testverfahren.
  - ...

Quellcode-Inspektion / Code-Review

Beispiel: die sog. Fagan-Inspektion als Qualität sichernde analytische Maßnahme in der Programmierung.

- Beteiligte Personen
  - Moderator, bereitet die Inspektion vor und nach und moderiert den Termin.
  - Ca. zwei weitere Personen (gewöhnlich andere Programmierer) als Inspektoren.
  - Der oder die Programmierer.
- Ablauf
  - Vorbereitung
    - Moderator und Inspektoren erhalten rechtzeitig vorab den zu prüfenden Quellcode und ggf. weiteres erforderliches Material, und bereiten sich gründlich vor (auf Grundlage ihrer eigenen Erfahrung und mittels Inspektions-Richtlinien).
  - Inspektionstermin
    - Der Quellcode wird gemeinsam durchgelesen, die Teilnehmer weisen auf (mögliche) Fehler hin, der Moderator klassifiziert und dokumentiert sie nach Art und Schwere.
  - Nachbereitung
    - Anfertigung des Inspektionsberichts durch den Moderator, der für den oder die Programmierer die Basis für Korrekturen darstellt.

Einige Beispielfragen

Fehler.

- Bauen Sie sich gegenseitig Fehler in das `nickname` Programm und/oder in das `ggT` Programm ein. Verwenden Sie den Debugger, um die Fehler zu finden und das Programm auszubessern.
- Nennen Sie drei wichtige Fehlerarten und beschreiben Sie diese kurz.
- Welche Arten von Fehlern können wir in dieser Lehrveranstaltung ignorieren?
- Wie können Sie testen, ob eine Eingabeoperation erfolgreich war?
- Nennen Sie drei Möglichkeiten zur vorsorglichen Vermeidung von Laufzeitfehlern. Bewerten Sie jede der Möglichkeiten.
- Was ist ein Syntax- / Typ- / Runtime-Fehler? Nennen Sie jeweils Beispiele.
- Beschreiben Sie, wie C++ Ausnahmen "geworfen" und "gefangen" werden.
- Erklären Sie, was in der folgenden Quellcode-Zeile für einen `vector` namens `v` passiert: `v[v.size()] = x;`
- Was verstehen Sie unter instrumentiertem Quellcode?
- Wann würden Sie Vor- und Nachbedingungen und Invarianten *nicht* prüfen?
- Wie unterscheiden sich Qualitätssicherung, Testen und Debugging?

Nächste Einheit:
Fallstudie Taschenrechner