Programmstrukturierung

- Funktionen
    - Aufbau und Prototypen
    - Gültigkeitsbereiche und Sichtbarkeit in Funktionen

- Schnittstellen zum Datentransfer
    - Wertübergabe
    - Referenzübergabe
    - Vorgegebene Parameterwerte
    - Überladen von Funktionen
    - Die Funktion `main()`
    - Spezifikation von Funktionen

- Grundsätze der modularen Gestaltung
    - Einbinden vorübersetzter Programmteile
    - Dateiübergreifende Gültigkeit und Sichtbarkeit
    - Übersetzungseinheit, Deklaration, Definition
    - Compilerdirektiven und Makros
    - Einbinden von C-Funktionen
    - `inline`-Funktionen

Programmstrukturierung

Große Programme müssen in übersichtliche Teile zerlegt werden. Dazu dienen die folgenden Mechanismen:

- Delegieren von Teilaufgaben in Form von Funktionen beschreiben
- Strukturieren und Zusammenfassen von zusammengehörigen Teilaufgaben in Modulen

Funktionen

Eine Funktion erledigt eine abgeschlossene Teilaufgabe. Die notwendigen Daten werden der Funktion mitgegeben, und sie gibt das Ergebnis der erledigten Aufgabe an den Aufrufer (Auftraggeber) zurück.

Eine Funktion muss nur einmal definiert werden. Anschließend kann sie beliebig oft durch Nennung ihres Namens aufgerufen werden, um die ihr zugewiesene Teilaufgabe abzuarbeiten.

Teilaufgaben können selbst wieder in Teilaufgaben unterteilbar sein, die durch Funktionen zu bearbeiten sind.

Die Komplexität einer Aufgabe wird durch Zerlegung in Teilaufgaben auf mehreren Ebenen reduziert – nach dem Prinzip „teile und herrsche“.

Funktionen

Aufbau und Prototypen

Eine Funktion `fakultaet()` könnte wie folgt in ein Programm integriert werden:

```
#include <iostream>
using namespace std;

// Funktionsprototyp (Deklaration)
unsigned long fakultaet(int);

int main()
{
   int n;
   do {
      cout << "Fakultaet berechnen. Zahl >= 0: ";
      cin >> n;
   } while(n < 0);

   cout << "Das Ergebnis ist "
        << fakultaet(n) << endl;  // Funktionsaufruf

   return 0;
}

// Funktionsimplementation (Definition)
unsigned long fakultaet(int zahl)
{
   unsigned long fak = 1;
   for(int i = 2; i <= zahl; ++i)
      fak *= i;
   return fak;
}
```

Funktionen

Aufbau und Prototypen

Eine **Deklaration** sagt dem Compiler, dass eine Funktion oder Variable mit diesem Aussehen irgendwo definiert ist.

Eine **Definition** veranlasst den Compiler, entsprechenden Code zu erzeugen und den notwendigen Speicherplatz anzulegen.

Eine Funktionsdeklaration, die nicht gleichzeitig eine Definition ist, wird **Funktionsprototyp** genannt.

Der Aufruf einer Funktion geschieht durch einfache Namensnennung. Von der Funktion auszuwertende Daten werden in runden Klammern übergeben.

Funktionen

Aufbau und Prototypen

Syntax eines Funktions*prototypen*:

Rückgabetyp Funktionsname (*Parameterliste*) ;

Beispiel:

```
unsigned long fakultaet(int zahl);
```

Für den Aufbau einer Parameterliste gibt es folgende Möglichkeiten:

- leere Liste:

```
int func();
```

- gleichwertig mit der leeren Liste ist:

```
int func(void);
```

- Liste mit Parametertypen:

```
int func(int, char);
```

- Liste mit Parametertypen und -namen:

```
int func(int i, char c);
```

Funktionen

Aufbau und Prototypen

Syntax der Funktions*definition*:

Rückgabetyp Funktionsname
(*Formalparameterliste*) *Block*

Beispiel:

```
unsigned long fakultaet(int zahl) { ... }
```

Syntax des Funktions*aufrufs*:

Funktionsname (*Aktualparameterliste*)

Beispiel:

```
unsigned long ergebnis = fakultaet(n);
```

Funktionen

Gültigkeitsbereiche und Sichtbarkeit in Funktionen

Der Funktionskörper ist ein Block, also ein durch geschweifte Klammern {} begrenztes Programmstück.

Danach sind alle Variablen einer Funktion nicht im Hauptprogramm gültig, und auch nicht sichtbar.

Eine Sonderstellung haben die in der Parameterliste aufgeführten Variablen: sie werden innerhalb der Funktion wie lokale Variablen betrachtet, und von außen gesehen stellen sie die *Datenschnittstelle* zur Funktion dar.

Beispiel:

```
#include <iostream>
using namespace std;

int a = 1;       // ueberall bekannt, also global

void f1() {
    int c = 3;   // nur in f1() bekannt, also lokal
    cout << "f1: c = " << c << endl;
    cout << "f1: globales a = " << a << endl;
}

int main() {
    cout << "main: globales a = " << a << endl;

    // cout << "f1: c = " << c;
    // ist nicht compilierfaehig,
    // weil c in main() unbekannt ist.
    f1( );       // Aufruf von f1()
    return 0;
}
```

Funktionen

Gültigkeitsbereiche und Sichtbarkeit in Funktionen

Beim Betreten eines Blocks wird für die innerhalb eines Blocks deklarierten Variablen Speicherplatz beschafft; die Variablen werden gegebenenfalls initialisiert (wenn ein Initialisierungsausdruck angegeben ist). Der Speicherplatz wird bei Verlassen des Blocks wieder freigegeben.

Dies gilt auch für Variablen in Funktionen, wobei der Aufruf einer Funktion dem Betreten des Blocks entspricht. Die Rückkehr zum Aufrufer der Funktion wirkt wie das Verlassen eines Blocks.

Eine Ausnahme bilden Variablen, die innerhalb eines Blocks oder einer Funktion als `static` definiert werden. `static`-Variablen erhalten einen festen Speicherbereich und werden nur ein einziges Mal *beim ersten Aufruf der Funktion* initialisiert. Falls kein Initialisierungswert vorgegeben ist, werden sie automatisch auf 0 gesetzt.

`static`-Variablen wirken wie ein Gedächtnis für eine Funktion, weil sie zwischen Funktionsaufrufen ihren Wert nicht verlieren.

Funktionen

Gültigkeitsbereiche und Sichtbarkeit in Funktionen

Beispiel: eine Funktion, die sich merken kann, wie oft sie aufgerufen wurde

```
#include <iostream>
using namespace std;

// zaehlt die Anzahl der Aufrufe
void func()
{
    static int anz = 0;
    cout << "Anzahl = " << ++anz << endl;
}

int main()
{
    for(int i = 0; i < 3; ++i)
        func();

    return 0;
}
```

Ausgabe des Programms:

```
Anzahl = 1
Anzahl = 2
Anzahl = 3
```

Funktionen

Gültigkeitsbereiche und Sichtbarkeit in Funktionen

Beispiel: Unterschiede `static` vs. `automatic`

```
#include <iostream>
using namespace std;

void func()
{
    auto int a;
    static int s;

    cout << "automatic a = " << ++a << endl;
    cout << "   static s = " << ++s << endl;
}

int main()
{
    for(int i = 0; i < 3; ++i)
        func();

    return 0;
}
```

Ausgabe des Programms:

```
automatic a = -858993459
   static s = 1
automatic a = -858993459
   static s = 2
automatic a = -858993459
   static s = 3
```

Programmstrukturierung

Schnittstellen zum Datentransfer

Der Datentransfer in Funktionen hinein und aus Funktionen heraus wird durch die Beschreibung der Schnittstelle festgelegt.

Die Schnittstelle wird durch den Funktionsprototyp eindeutig beschrieben und enthält

- den Rückgabetyp der Funktion,
- den Funktionsnamen,
- Parameter, die der Funktion bekannt gemacht werden, und
- die Art der Parameterübergabe.

Wir unterscheiden zwei Arten des Datentransports:

- Übergabe *per Wert* (englisch *call by value*)
- Übergabe *per Referenz* (englisch *call by reference*)

Schnittstellen zum Datentransfer

Wertübergabe

Der Wert wird *kopiert* und an die Funktion übergeben. Innerhalb der Funktion wird mit der *Kopie* gearbeitet, das *Original* beim Aufrufer *bleibt unverändert erhalten*.

Beispiel:

```
#include <iostream>
using namespace std;

int addiere_5(int);       // Deklaration

int main() {
    int erg, i = 0;
    cout << i << " = Wert von i\n";
    erg = addiere_5(i);
    cout << erg << " = Ergebnis von addiere_5\n";
    cout << i << " = i unveraendert!\n";
    return 0;
}

int addiere_5(int x) {    // Definition
    x += 5;
    return x;
}
```

Ausgabe des Programms:

```
0 = Wert von i
5 = Ergebnis von addiere_5
0 = i unveraendert!
```

Schnittstellen zum Datentransfer

Wertübergabe

Beispiel: rekursive Berechnung der Quersumme einer ganzen Zahl

```
#include <iostream>
using namespace std;

int qsum(long z)   // Wertübergabe
{
    if(z > 0)      // Abbruchbedingung z == 0
    {
        int letzteZiffer = z % 10;
        // rekursiver Aufruf von qsum()
        return letzteZiffer + qsum(z / 10);
    }
    else
        return 0;
}

int main()
{
    cout << "Zahl: ";
    long zahl;
    cin >> zahl;
    cout << "Quersumme = " << qsum(zahl) << endl;
    return 0;
}
```

Schnittstellen zum Datentransfer

Wertübergabe

Beispiel: iterative Berechnung der Quersumme einer ganzen Zahl

```
#include <iostream>
using namespace std;

int qsum(long z)   // Wertuebergabe
{
    int sum = 0;
    while(z > 0)
    {
        sum += z % 10;
        z /= 10;
    }
    return sum;
}

int main()
{
    cout << "Zahl: ";
    long zahl;
    cin >> zahl;
    cout << "Quersumme = " << qsum(zahl) << endl;
    return 0;
}
```

Schnittstellen zum Datentransfer

Referenzübergabe

Wenn ein übergebenes Objekt modifiziert werden soll, kann die Übergabe durch eine *Referenz* des Objekts geschehen. Die Syntax des Aufrufs ist die gleiche wie bei der Übergabe per Wert, anstatt mit einer Kopie wird jedoch *direkt mit dem Original* gearbeitet, wenn auch unter anderem Namen.

Beispiel:

```
#include <iostream>
using namespace std;

void addiere_7(int&); // int& == Referenz auf int

int main() {
    int i = 0;
    cout << i << " = alter Wert von i\n";
    addiere_7(i);    // Syntax wie bei Wertuebergabe
    cout << i
         << " = neuer Wert von i nach addiere_7\n";
    return 0;
}

void addiere_7(int& x) {
    // Original des Aufrufers wird geaendert!
    x += 7;
}
```

Ausgabe des Programms:

```
0 = alter Wert von i
7 = neuer Wert von i nach addiere_7
```

Schnittstellen zum Datentransfer

Referenzübergabe

Bei der Rückgabe von Referenzen muss darauf geachtet werden, dass das zugehörige Objekt tatsächlich noch existiert.

Das folgende Beispiel zeigt, wie man es *nicht* machen sollte:

```
int& max(int a, int b)
{
   if(a > b)
      return a;
   else
      return b;
}

int main()
{
   int x = 11, y = 22;
   int z = max(x, y);
   // ...
   return 0;
}
```

Schnittstellen zum Datentransfer

Referenzübergabe

Referenzen können zu sehr "häßlichem" Code führen.

Die Funktionsdefinition

```
int &func()
{
  static int value = 10;

  return value;
}
```

ermöglicht z.B. folgende Funktionsaufrufe:

```
func() = 20;
func() += func();
```

Schnittstellen zum Datentransfer

Vorgegebene Parameterwerte

Bei der Deklaration einer Funktion können für Parameter *vorgegebene Werte* (englisch *default values*) festgelegt werden.
Falls der Programmierer diese Parameter beim Funktionsaufruf nicht angibt, werden sie vom Compiler eingesetzt.

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

void showString(string str = "Hello, World!")
{
   cout << str << endl;
}

int main()
{
   showString("Here's an explicit argument.");

   // Bedeutung: showstring("Hello, World!");
   showString();

   return 0;
}
```

Die Parameter mit vorgegebenen Werten erscheinen in der Deklaration **nach** den anderen Parametern.

Schnittstellen zum Datentransfer

Vorgegebene Parameterwerte

Es sind auch mehrere vorgegebene Parameterwerte möglich:

```
void two_ints(int a = 1, int b = 4)
{
   . . .
}

int main()
{
   two_ints();          // Argumente:  1, 4
   two_ints(20);        // Argumente: 20, 4
   two_ints(20, 5);     // Argumente: 20, 5

   return 0;
}
```

Die Anweisung

```
two_ints(, 6);
```

ist nicht erlaubt. Argumente dürfen beim Aufruf nur **rechts** fehlen.

Der Compiler muss die vorgegebenen Parameterwerte kennen, wenn er für Funktionsaufrufe Code generiert, in die evtl. Argumentwerte einzusetzen sind.

Das bedeutet oftmals, dass vorgegebene Parameterwerte in Deklarationsdateien spezifiziert werden müssen.

Schnittstellen zum Datentransfer

Überladen von Funktionen

In C++ ist es möglich, mehrere gleichnamige Funktionen zu definieren, die sich unterschiedlich verhalten. Die Funktionen müssen sich lediglich in ihrer Argumentliste unterscheiden.

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

void show(int val)
{
   cout << "int:    " << val << endl;
}

void show(double val)
{
   cout << "double: " << val << endl;
}

void show(string val)
{
   cout << "string: " << val << endl;
}

int main()
{
   show(12);
   show(3.1415);
   show("Hello, World!");
   return 0;
}
```

Schnittstellen zum Datentransfer

Überladen von Funktionen

Das Überladen von Funktionen funktioniert nur innerhalb **desselben** Gültigkeitsbereichs

```
#include<iostream>
using namespace std;

void f(char c)
{
   cout << "f(char) c = " << c << endl;
}

void f(double x)
{
   cout << "f(double) x = " << x << endl;
}

int main()
{
   // ein neuer Gueltigkeitsbereich beginnt
   void f(double);
   f('a');
   return 0;
}
```

Die Deklaration `void f(double);` führt dazu, dass `f(char)` nicht mehr sichtbar ist. Es wird `f(double)` ausgeführt, wobei das Zeichen `'a'` in eine `double`-Zahl umgewandelt wird.

Schnittstellen zum Datentransfer

Überladen von Funktionen

C++ erlaubt es **nicht**, dass mehrere Funktionen sich nur beim Typ des Rückgabewerts unterscheiden.

Grund: Der Programmierer kann den Rückgabewert einer Funktion ignorieren. So enthält z.B. der Aufruf

```
printf("Hello, World!\n");
```

keine Information über den Rückgabewert der Funktion `printf()`.

Vermeiden Sie Funktionen, die zwar den gleichen Namen haben, aber völlig unterschiedliche Aktionen ausführen.

Schnittstellen zum Datentransfer

Die Funktion `main()`

`main()` ist eine spezielle Funktion. Jedes C++-Programm startet definitionsgemäß mit `main()`, so dass `main()` in jedem C++-Programm genau einmal vorhanden sein muss.

`main()` kann nicht überladen oder von einer anderen Funktion aufgerufen werden.

Der Rückgabetyp der Funktion `main()` soll `int` sein, die Funktion ist ansonsten aber implementierungsabhängig. Die folgenden beiden Formen sind mindestens gefordert:

```
int main()
int main(int argc, char *argv[])
```

Meist steht auch die folgende Form zur Verfügung:

```
int main(int argc, char *argv[], char *env[])
```

Schnittstellen zum Datentransfer

Die Funktion `main()`

Beispiel: Kommandozeilen-Argumente und Umgebungsvariablen ausgeben

```
// C++-Programm args.cpp
// Testen z.B. mit dem folgenden Aufruf:
// args first_arg "arg with blanks" 3  4 "last but
  one" stop!

#include <iostream>
using namespace std;

int main(int argc, char *argv[], char *env[])
{
   cout << "The value of argc is " << argc
        << "\n\n";

   cout << "These are the " << argc
        << " command-line arguments passed to"
        << " main:\n\n";
   for (int i = 0; i < argc; i++)
      cout << "   argv[" << i << "]: "
           << argv[i] << "\n";

   cout << "\nThe environment string(s)"
        << " on this system are:\n\n";
   for (int i = 0; env[i] != NULL; i++)
      cout << "   env[" << i << "]: "
           << env[i] << "\n";

   return 0;
}
```

Schnittstellen zum Datentransfer

Die Funktion `main()`

Beispiel: Kommandozeilen-Argumente und Umgebungsvariablen ausgeben

```
// C-Programm args.c
// Testen z.B. mit dem folgenden Aufruf:
// args first_arg "arg with blanks" 3  4 "last but
  one" stop!

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[], char *env[])
{
   int i;

   printf("The value of argc is %d \n\n", argc);
   printf("These are the %d command-line arguments"
          " passed to main:\n\n", argc);

   for (i = 0; i < argc; i++)
      printf("   argv[%d]: %s\n", i, argv[i]);

   printf("\nThe environment string(s) on this"
          " system are:\n\n");

   for (i = 0; env[i] != NULL; i++)
      printf("   env[%d]: %s\n", i, env[i]);

   return 0;
}
```

Schnittstellen zum Datentransfer

Die Funktion `main()`

Der Aufruf

```
args first_arg "arg with blanks" 3 4 "last but one" stop!
```

könnte z.B. die folgende Ausgabe liefern:

```
The value of argc is 7

These are the 7 command-line arguments passed to main:

   argv[0]: C:\CBUILDER\args.exe
   argv[1]: first_arg
   argv[2]: arg with blanks
   argv[3]: 3
   argv[4]: 4
   argv[5]: last but one
   argv[6]: stop!

The environment string(s) on this system are

   env[0]: COMSPEC=C:\COMMAND.COM
   env[1]: PROMPT=$p $g
   env[2]: PATH=C:\SPRINT;C:\DOS;C:\ CBUILDER
```

Schnittstellen zum Datentransfer

Spezifikation von Funktionen

Eine Funktion erledigt eine Teilaufgabe und ändert dabei den *Zustand* eines Programms. Es ist sinnvoll, die Zustandsänderung durch die Bedingungen, die *vor* und *nach* dem Aufruf gelten, im Funktionskopf als Kommentar anzugeben.

Dazu gehören z.B.:

- Annahmen über die Importschnittstelle (Eingabedaten, z.B. Wertebereich),
- die Fehlerbedingungen,
- die Exportschnittstelle (Ausgabedaten)

Schnittstellen zum Datentransfer

Spezifikation von Funktionen: Beispiel

```
// Berechnung der Nullstelle einer Funktion
// nach dem Newton-Verfahren
int newton(
// Import
   double (*fp) (double),
   double untergrenze,
   double obergrenze,
   double epsilon,
// Export
   double &nullstelle)
/*
Returncodes:
   0: Nullstelle gefunden
   1: Nullstelle nicht gefunden
   2: Verfahren konvergiert nicht
   3: Vorbedingungen nicht erfuellt
Vorbedingungen:
   1. untergrenze < obergrenze
   2. epsilon > Darstellungsgenauigkeit
   3. stetig differenzierbare Funktion f
   4. im Intervall befindet sich eine Nullstelle
Nachbedingungen:
   bei Returncode 0:
      1. untergrenze <= nullstelle <= obergrenze
      2. abs(f(nullstelle)) <= epsilon
   sonst:
      3. siehe Returncode-Beschreibung
*/
{
      // Programmcode
}
```

Programmstrukturierung

Grundsätze der modularen Gestaltung

Die Aufteilung eines großen Programms in einzelne, getrennt übersetzbare Dateien, die jeweils zusammengehörige Programmteile enthalten, ist sinnvoll.

Der folgende Aufbau ist empfehlenswert:

- Standard-Header haben die Form `<headername>`. Es kann außerdem eigene (oder fremde) Header-Dateien geben, die typischerweise die Endung `*.h` im Dateinamen haben. Sie enthalten
  - Konstanten,
  - Schnittstellenbeschreibungen wie z.B. Klassendeklarationen,
  - Deklarationen globaler Daten und
  - Funktionsprototypen.

- Implementierungs-Dateien enthalten die Implementierungen der Klassen und den Programmcode der Funktionen (Endung im Dateinamen: `*.cpp`)

- Die Main-Datei (`*.cpp`) enthält das Hauptprogramm `main()`.

Programmstrukturierung

Grundsätze der modularen Gestaltung

Damit eine Datei für sich übersetzbar ist, müssen Konstanten, Klassen-Schnittstellen und Funktions-prototypen bekannt sein.

Dies wird erreicht durch Einschließen der Header-Dateien mit der Präprozessor-Direktive

```
#include "filename.h"
```

Die Datei `filename.h` wird *im aktuellen Verzeichnis* gesucht und an dieser Stelle eingelesen.

Wird die Datei nicht gefunden, so wird in den voreingestellten `/include`-Verzeichnissen gesucht.

Falls auch diese Suche fehlschlägt, wird versucht, die Direktive in der Standard-Header-Form als `#include <filename>` zu interpretieren.

Grundsätze der modularen Gestaltung

Einbinden vorübersetzter Programmteile

Bei großen Programmen ist es sinnvoll,

- *Schnittstellen* (Funktionsprototypen und Klassen) und
- *Implementierungen* (Programmcode) zu trennen.

Um die automatische Prüfung der Schnittstellen durch den Compiler zu ermöglichen, werden die Header-Dateien mit `#include` in allen Dateien eingeschlossen, welche diese Schnittstellen verwenden.

Mit den Header-Dateien kann jede Datei einzeln übersetzt werden. Wenn es Änderungen gibt, müssen nur noch die davon betroffenen Dateien neu compiliert werden.

Beispiel:

- Die Schnittstellen seien in den Header-Dateien `a.h` und `b.h` abgelegt.
- Die Implementierungen seien in den Dateien `a.cpp` und `b.cpp` enthalten.

Grundsätze der modularen Gestaltung

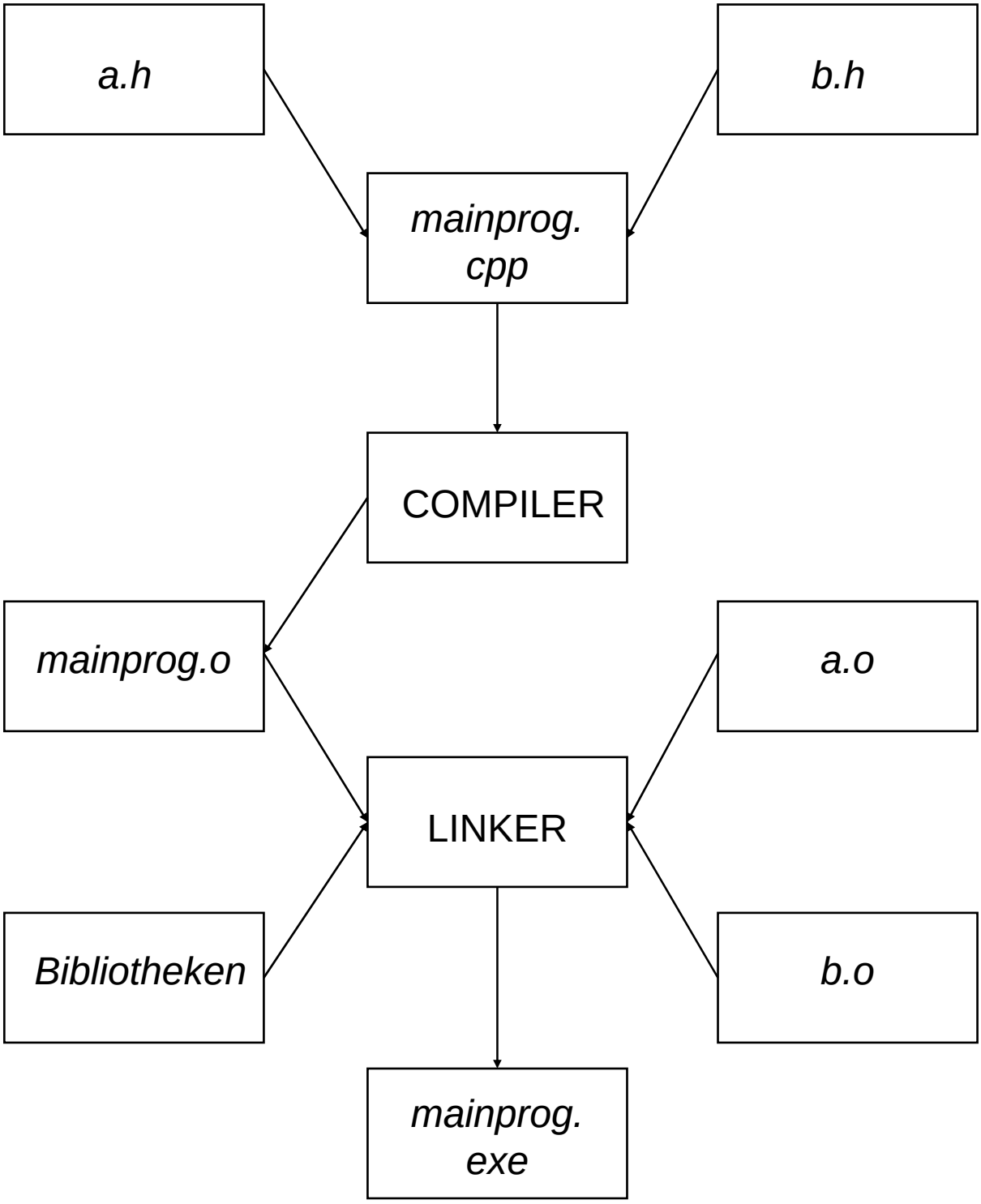
Einbinden vorübersetzter Programmteile

```
// a.h
void func_a1();
void func_a2();

// a.cpp
#include "a.h"
void func_a1() {
   // Programmcode fuer func_a1
}
void func_a2() {
   // Programmcode fuer func_a2
}

// b.h
void func_b();

// b.cpp
#include "b.h"
void func_b() {
   // Programmcode fuer func_b
}

// mainprog.cpp
#include "a.h"
#include "b.h"
int main()
{
   func_a1();
   func_a2();
   func_b();
}
```

Grundsätze der modularen Gestaltung

Einbinden vorübersetzter Programmteile

Übersetzungs- und Binde-Ablauf



Grundsätze der modularen Gestaltung

Einbinden vorübersetzter Programmteile

Die Konzepte

- Trennung von Schnittstellen und Implementierung, und

- Gruppierung zusammengehöriger Funktionen und Klassen zu Bibliotheksmodulen

sind Standard in allen größeren Programmierprojekten.

Grundsätze der modularen Gestaltung

Dateiübergreifende Gültigkeit und Sichtbarkeit

Die Speicherklasse einer Variablen wird durch die Schlüsselwörter

- `auto`
- `static`
- `extern`
- `register`
- `mutable`

bestimmt.

Mit `auto`, `static` und `extern` werden Sichtbarkeit und Lebensdauer von Variablen eingestellt.

Variablen, die außerhalb von `main()` und jeglicher anderer Funktion definiert sind, heißen *global*. Sie sind in *allen* Teilen eines Programms gültig, auch in anderen Dateien.

Eine globale Variable muss in einer anderen Datei lediglich als `extern` deklariert werden, um dort benutzbar zu sein.

Grundsätze der modularen Gestaltung

Dateiübergreifende Gültigkeit und Sichtbarkeit

Beispiel: globale Variablen: *Definition* vs. *Deklaration*

```
// Datei1.cpp
// Deklaration und Definition:
int global;
int main()
{
   global = 4711;
}
```

```
// Datei2.cpp
// Deklaration, aber keine Definition:
extern int global;
int func1()
{
   global = 9876;
}
```

Grundsätze der modularen Gestaltung

Dateiübergreifende Gültigkeit und Sichtbarkeit

Um den Gültigkeitsbereich von Variablen und Funktionen auf eine Datei zu beschränken, ist das Schlüsselwort `static` notwendig.

Beispiel: datei-statische Variablen

```
// Datei1.cpp

// nicht mehr global
// nur in Datei1.cpp gueltig
static int global;
int main()
{
   global = 4711;
}
```

```
// Datei2.cpp

// extern fuehrt jetzt zu einer
// Fehlermeldung des Binders

extern int global;
int func1()
{
   global = 9876;
}
```

Grundsätze der modularen Gestaltung

Dateiübergreifende Gültigkeit und Sichtbarkeit

Auf Dateiebene (d.h., außerhalb einer Funktion) definierte *Variablen* sind *global* und in anderen Dateien benutzbar, wenn sie dort als `extern` deklariert sind.

Bei *Konstanten* (`const`) ist es jedoch umgekehrt. Konstanten sind nur in der Definitionsdatei sichtbar! Sollen Konstanten in anderen Dateien zugänglich gemacht werden, müssen sie als `extern` deklariert und initialisiert werden:

```
// Datei1.cpp

// Deklaration und Definition:

extern const double pi = 3.14159;


// Datei2.cpp

// ohne Initialisierung =>
// Deklaration ohne Definition:

extern const double pi;
```

Grundsätze der modularen Gestaltung

Dateiübergreifende Gültigkeit und Sichtbarkeit

Alle nicht globalen und nicht-`static`-Variablen sind sogenannte automatische (`auto`) Variablen.

Automatische Variablen werden bei Betreten eines Blocks angelegt und bei Verlassen des Blocks wieder zerstört.

Wenn automatische Variablen nicht ausdrücklich initialisiert werden, ist ihr Wert undefiniert:

```
{
   // explizite Initialisierung,
   // i erhält den Wert 10
   int i = 10;
   // Initialisierung fehlt =>
   // der Wert von j ist undefiniert
   int j;
}
```

Mit `register` gekennzeichnete Variablen sind `auto`-Variablen. `register` ist lediglich ein Hinweis an den Compiler, diese Variable in einem Register der CPU abzulegen.

Grundsätze der modularen Gestaltung

Übersetzungseinheit, Deklaration, Definition

Alles, was der Compiler in einem Durchgang liest, ist eine *Übersetzungseinheit*.

Zum Verständnis ist die klare Unterscheidung der Begriffe *Deklaration* und *Definition* wichtig:

- Eine **Deklaration** führt einen Namen in ein Programm ein und gibt dem Namen eine Bedeutung.

- Eine Deklaration ist auch eine **Definition**, *wenn mehr als nur der Name eingeführt wird*, z.B. wenn Speicherplatz für Daten oder Code angelegt oder die innere Struktur eines Datentyps beschrieben wird, aus der sich der benötigte Speicherplatz ergibt.

Grundsätze der modularen Gestaltung

Übersetzungseinheit, Deklaration, Definition

Deklarationen, die gleichzeitig Definitionen sind:

```
// Speicherplatz fuer a wird angelegt
int a;
// Speicherplatz fuer pi wird angelegt
extern const double pi = 3.14159;
// enthaelt Programmcode
int f(int n) { return n * n; }
// definiert meinStrukt
struct meinStrukt {
   // definiert c und d
   int c, d;
};
// Speicherplatz fuer x wird angelegt
meinStrukt x;
// definiert li und re
enum meinEnum { li, re };
// Speicherplatz fuer me wird angelegt
meinEnum me;
```

Deklarationen, aber *keine* Definitionen:

```
extern int a;
extern const double pi;
int f(int);
struct meinStrukt;
enum meinEnum;
```

Grundsätze der modularen Gestaltung

Übersetzungseinheit, Deklaration, Definition

Die folgende unter dem Namen *one definition rule* bekannte Regel ist bei der Strukturierung von Programmen zu beachten:

> Jede Variable, Funktion, Struktur, Konstante usw. in einem Programm hat *genau eine* Definition.

Aus der one definition rule ergibt sich, was in den verschiedenen Dateitypen enthalten sein sollte:

Header-Dateien (* . h, * . hpp)

- Funktionsprototypen (Schnittstellen)

```
void meineFunktion(int einParameter);
```

- reine Deklaration (nicht Definition) globaler Variablen

```
extern int globaleVariable;
```

- reine Deklaration globaler Konstanten (nicht Definition, also ohne Initialisierung)

```
extern const int globaleKonstante;
```

- Definition von Datentypen wie `enum` oder `struct`

```
struct Punkt { int x; int y; };
enum Farbe { rot, gelb, gruen, blau };
```

Grundsätze der modularen Gestaltung

Übersetzungseinheit, Deklaration, Definition

Implementierungs-Dateien (*.cpp, *.cc, *.C)

- Funktionsdefinitionen (Implementierung)

```
void meineFunktion(int einParameter)
{
   // Programmcode
}
```

- Definition globaler Variablen (nur *einmal* im gesamten Programm)

```
int globaleVariable;
```

- Definition und Initialisierung globaler Konstanten (nur *einmal* im gesamten Programm)

```
extern const int globaleKonstante = 1;
```

- Definition von Objekten bestimmter Datentypen

```
Punkt einPunkt;
Farbe eineFarbe;
```

Grundsätze der modularen Gestaltung

Compilerdirektiven und Makros

Compilerdirektiven sind Anweisungen an den Präprozessor, die den Übersetzungsprozess steuern, wie z.B. `#include`. Compilerdirektiven beginnen stets mit # am Zeilenanfang.

`#include`

Die Dateispezifikation kann außer dem Dateinamen auch den vollständigen Pfad enthalten:

```
#include "f:/project/include/ioctrl/io.h"
```

`#define`, `#ifdef`, `#ifndef`

Es kann zu Problemen beim Übersetzen führen, wenn Header-Dateien mehrfach eingebunden sind, so dass sich mehrfache Definitionen ergäben (siehe folgendes Beispiel).

Abhilfe schaffen die Anweisungen

- `#if defined` (Abkürzung: `#ifdef`)
- `#if !defined` (Abkürzung: `#ifndef`)
- `#define`

Grundsätze der modularen Gestaltung

Beispiel: Mehrfach-Inklusion einer Header-Datei

```
// a.h
#include "c.h"
void func_a1();
void func_a2();

// a.cpp
#include "a.h"
void func_a1() {
   // Programmcode fuer func_a1
}
void func_a2() {
   // Programmcode fuer func_a2
}

// b.h
#include "c.h"
void func_b();

// b.cpp
#include "b.h"
void func_b() {
   // Programmcode fuer func_b
}

// mainprog.cpp
#include "a.h"
#include "b.h"
int main()
{
   func_a1();
   func_a2();
   func_b();
}
```

Grundsätze der modularen Gestaltung

Compilerdirektiven und Makros

Beispiel: Mehrfach-Inklusion einer Header-Datei vermeiden mittels `#define` und `#ifndef`

```
// c.h

#ifndef c_h
#define c_h

void func_c1();
void func_c2();

#endif   // c_h
```

Bedeutung:

Falls der Name c_h *nicht* definiert ist,
dann definiere c_h und akzeptiere alles bis `#endif`.

#undef

Mit `#undef` kann eine Definition rückgängig gemacht werden.

Grundsätze der modularen Gestaltung

Compilerdirektiven und Makros

`#define` und `#ifdef` können zur gezielten Ein- und Ausblendung von Testsequenzen in einem Programm verwendet werden.

Beispiel:

```
#define DEBUG
...
#ifdef DEBUG
if(x < 0) cerr << "sqrt(negative Zahl)"
                << endl;
#endif
y = sqrt(x);
...
```

Falls `DEBUG` *nicht* gesetzt ist, wird die Testausgabe vom Präprozessor aus dem Programmcode entfernt.

Nach erfolgreichem Testen des Programms genügt es also die Zeile `#define DEBUG` zu löschen oder auszukommentieren, um alle Testausgaben verschwinden zu lassen.

Grundsätze der modularen Gestaltung

Compilerdirektiven und Makros

Zur gezielten Ein- und Ausblendung von Testsequenzen gibt es eine Lösung, die ohne den Präprozessor auskommt:

```
const bool DEBUG = true;
...
// nur im Test soll bei Fehlern
// eine Meldung ausgegeben werden
if(DEBUG)
   if(x < 0) cerr << "sqrt(negative Zahl)"
                  << endl;
y = sqrt(x);
...
```

Grundsätze der modularen Gestaltung

Compilerdirektiven und Makros

Makros mit `#define`

`#define` ermöglicht das Ersetzen von Makros durch Zeichenketten, wobei Parameter erlaubt sind.

Die Makrodefinitionen

```
#define PI  3.14
#define schreibe  cout
#define QUADRAT(x)  ((x)*(x))
```

erlauben in einem Programm den Text

```
schreibe << PI << endl;
y = QUADRAT(z);
```

der vom Präprozessor ersetzt würde durch

```
cout << 3.14 << endl;
y = ((z)*(z));
```

Grundsätze der modularen Gestaltung

Compilerdirektiven und Makros

Makros mit `#define`

Die Textersetzung mit `#define` sollte im allgemeinen nicht verwendet werden, wenn es Alternativen gibt.

Makros sehen oft harmlos aus, können aber gefährlich sein. Der Aufruf

```
int j = QUADRAT(++n);
```

z.B. wird vom Präprozessor ersetzt durch

```
int j = ((++n)*(++n));
```

Makronamen sind einem symbolischen Debugger nicht zugänglich, wie die Pseudo-Konstante `PI` im obigen Beispiel.

Bei der Makrodefinition müssen der gesamte Makrorumpf und alle Argumente im Makrorumpf in runde Klammern gesetzt werden:

```
#define QUADRAT(x)  ((x)*(x))
```

Begründung: siehe Übung

Grundsätze der modularen Gestaltung

Compilerdirektiven und Makros

Makros mit `#define`

Ein weiterer Nachteil von Makros besteht in der Umgehung der Typkontrolle:

```
// Makrodefinition
#define MULT(a,b)  ((a)*(b))

// Makroaufruf
int a, b = 2, c = 3;
a = MULT(b, c);              // o.k.
a = MULT(b, "Fehler!");
```

Grundsätze der modularen Gestaltung

Compilerdirektiven und Makros

Makros mit `#define`

Speziell für Testausgaben ist das folgende Makro `PRINT( )` nützlich, welches das Argument mit vorangestelltem # in eine Zeichenkette wandelt:

```
#define PRINT(x) cout << (#x) << " = "  \
                      << (x) << endl
```

Damit kann kurz z.B.

```
PRINT(int(xptr) – int(xptr2));
```

geschrieben werden statt

```
cout << "int(xptr) – int(xptr2) = "
     << int(xptr) – int(xptr2) << endl;
```

Die Ausgabe könnte sein

```
int(xptr) – int(xptr2) = 4
```

Grundsätze der modularen Gestaltung

Compilerdirektiven und Makros

Aufbau von Header-Dateien

Header-Dateien sollten nach dem folgenden Schema aufgebaut sein:

```
// Datei fn.h
#ifndef fn_h
#define fn_h fn_h
   // hier folgen die Deklarationen
#endif   // fn_h
```

Warum `#define fn_h fn_h` ?

```
// Datei fn.h
#ifndef fn_h
#define fn_h
   // hier folgen die Deklarationen
   enum fn_h {a, b, c };              // Fehler!
#endif   // fn_h
```

Eine Alternative ist die Hervorhebung durch Großschreibung:

```
// Datei fn.h
#ifndef FN_H
#define FN_H
   // hier folgen die Deklarationen
#endif   // FN_H
```

Grundsätze der modularen Gestaltung

Compilerdirektiven und Makros

Verifizieren logischer Annahmen mit `assert`

Das Makro `assert()` dient zur Überprüfung logischer Annahmen, die an der Stelle des Makros gültig sein sollen. Es lassen sich insbesondere Vor- und Nachbedingungen verifizieren.

Beispiel:

```
#include <cassert>

const int grenze = 100;
int index;

// ... Berechnung von index

// Test auf Einhaltung der Grenzen:
assert(index >= 0 && index < grenze);
```

Falls die Annahme (`index >= 0 && index < grenze`) nicht stimmen sollte, wird das Programm mit einer Fehlermeldung abgebrochen.

Grundsätze der modularen Gestaltung

Compilerdirektiven und Makros

Verifizieren logischer Annahmen mit `assert`

Beispiel:

```
#include <cassert>
using namespace std;

int main()
{
    const int grenze = 100;
    int index;

    // Berechnung von index
    index = -35;

    // Test auf Einhaltung der Grenzen:
    assert(index >= 0 && index < grenze);

    return 0;
}
```

führt zum Programmabbruch mit der Fehlermeldung:

```
Assertion failed: index >= 0 && index < grenze,
file assertion.cpp, line 13

abnormal program termination
```

Grundsätze der modularen Gestaltung

Compilerdirektiven und Makros

Verifizieren logischer Annahmen mit `assert`

`assert()` ist wirkungslos, falls `NDEBUG` vor

```
#include <cassert>
```

definiert wurde, entweder durch die Compilerdirektive

```
#define NDEBUG
```

oder durch Setzen der Compileroption `-D`, mit der Makrodefinitionen voreingestellt werden, z.B.:

```
g++ -DNDEBUG hello.cpp
```

Grundsätze der modularen Gestaltung

Compilerdirektiven und Makros

Verifizieren logischer Annahmen mit `assert`

Vermeiden Sie Seiteneffekte in `assert()` und generell in Makros. Eine in der Zusicherung aufgerufene Funktion wird bei gesetztem `NDEBUG` nicht ausgeführt.

Beispiel:

```
assert(dateiOeffnen(dateiname) == erfolgreich);
```

Falls `NDEBUG` gesetzt ist, wird die Datei nicht geöffnet.

Grundsätze der modularen Gestaltung

Compilerdirektiven und Makros

Einbinden von C-Funktionen

C-Funktionen, die z.B. in einer Programm-Bibliothek enthalten sind, können in C++-Programmen verwendet werden. Sie müssen jedoch als C- Funktionen deklariert werden.

```
extern "C" void *xmalloc(unsigned size);
```

Eine ähnliche Möglichkeit zur Deklaration von C-Funktionen ist:

```
extern "C"
{
    // hier folgen die C-Prototypen
}
```

Anstelle der Prototypen können auch Präprozessor-Direktiven eingefügt werden.

```
extern "C"
{
#include "myheader.h"
}
```

Grundsätze der modularen Gestaltung

Compilerdirektiven und Makros

Einbinden von C-Funktionen

Ein standard-konformer C++-Compiler muss das Symbol `__cplusplus` definieren.

Die Kombination des vordefinierten Symbols `__cplusplus` mit `extern "C"` -Deklarationen ermöglicht Deklarationsdateien, die von C- *und* C++-Compilern verarbeitet werden können.

```
#ifndef _MYHEADER_H_
#define _MYHEADER_H_

#ifdef __cplusplus
extern "C"
{
#endif  /* __cplusplus */

  /* Deklaration der C-Funktionen, z.B.: */
  extern void *xmalloc(unsigned size);

#ifdef __cplusplus
}
#endif  /* __cplusplus */

#endif  /* _MYHEADER_H_ */
```

Grundsätze der modularen Gestaltung

`inline`-Funktionen

Das Schlüsselwort `inline` bewirkt, dass bei der Übersetzung der Funktionsaufruf durch den Funktionskörper ersetzt wird, also gar kein echter Funktionsaufruf erfolgt. Die Parameter werden entsprechend ersetzt, auch die Syntaxprüfung bleibt erhalten.

Bei der Funktionsdefinition

```
inline int quadrat(int i)
{
   return i * i;
}
```

wird der Funktionsaufruf

```
z = quadrat(100);
```

vom Compiler durch

```
z = 100 * 100;
```

ersetzt.

Grundsätze der modularen Gestaltung

`inline`-Funktionen

sollten sich ausschließlich in Header-Dateien befinden.

```
// X.h
int f(int);
int g(int);
// Ende von X.h

// X.cpp
#include "X.h"

// Fehler: inline in Definitionsdatei!
inline int f(int i) {
   return i * i;
}

int g(int i) {
   i += 1;
   return f(i);        // inline ist hier bekannt
}
// Ende von X.cpp

// main.cpp
#include "X.h"

int main() {
   int i = 1, j;
   j = f(i) + g(i);    // inline ist hier unbekannt
}
// Ende von main.cpp
```

Grundsätze der modularen Gestaltung

`inline`-Funktionen

Richtig wäre die folgende Struktur:

```
// X.h
// korrekt: inline in Deklarationsdatei
inline int f(int i) {
   return i * i;
}

int g(int);
// Ende von X.h

// X.cpp
#include "X.h"

int g(int i) {
   i += 1;
   return f(i);        // inline ist hier bekannt
}
// Ende von X.cpp

// main.cpp
#include "X.h"

int main() {
   int i = 1, j;
   j = f(i) + g(i);    // inline auch hier bekannt
}
// Ende von main.cpp
```

Grundsätze der modularen Gestaltung

`inline`-Funktionen

Wann sollten `inline`-Funktionen verwendet werden?

- Im Allgemeinen nicht.
- Wenn eine Funktion lediglich aus einer (sehr einfachen) Anweisung besteht.
- Wenn ein vollständig entwickeltes und getestetes Programm inperformant ist, und Messungen "Flaschenhälse" in bestimmten Funktionen ergeben.
- Allgemein: wenn die Ausführungszeit für den Funktionsrumpf kurz ist im Vergleich zum Verwaltungsaufwand für den Funktionsaufruf.