Outline

1. Einleitung
2. Einführung in C
3. Fortgeschrittenes in C
4. Einführung in Emacs Lisp
5. Einführung in Prolog
6. Formale Semantik

Imperatives Paradigma

- *Ziel:* Ablaufbeschreibung
- *Elemente:* Anweisungen, Anweisungsfolgen, Variablenzuweisungen und Unterprogramme
- Beispiele: Fortran, C, Pascal, Modula-2, ...

Programmiersprache C

- Kernighan und Ritchie, 1978
- Aktuell: ISO 9899:2011 & ISO 9899:1999
- Nachschlagewerk: http://www.cppreference.com
- imperativ, prozedural (zum Teil funktional)
- Grundlage für viele weitere Programmiersprachen

Kleines *Hello World* Ansi-C-Programm

```
1 /* Dies ist ein Kommentar. Er erstreckt sich
2  * ueber mehrere Zeilen bis zum abschliessenden */
3
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 int main( int argc, char* argv[] )
8 {
9     /* gib Hello World auf der Konsole aus */
10    printf( "Hello World\n" );
11
12    return EXIT_SUCCESS; /* oder EXIT_FAILURE */
13 }
```

Kleines *Hello World* Ansi-C-Programm

- Compilieren von C-Programmen mit dem GCC:

```
$ gcc -o hallo hallo.c
```

- Compilieren mit zusätzlicher überprüfung auf Ansi-C Konformheit (für die Übungsaufgaben empfohlen):

```
$ gcc -Wall -Wextra -ansi -pedantic -o hallo hallo.c
```

- Es geht auch kürzer:

```
main(){puts("Hello World!");}
```

ABER das ist nicht Sinn der Sache!

Include- Direktive

Häufig benötigte Bibliotheken:

- `stdio.h` für Ein-/Ausgabe, `printf`, `scanf`
- `stdlib.h` für Speicherverwaltung, `malloc`, `free`
- `math.h` für mathematische Funktionen

Standarddatentypen, Variablendeklaration

```c
#include <stdlib.h>

int main( int argc, char *argv[] )
{
    int a;              /* Ganzzahl */
    const int c = 3; /* Konstante mit Wert 3 */
    long b = 123l;    /* lange Ganzzahl */
    char d = 'a';     /* ein Zeichen */
    short e = 20;     /* kurze Ganzzahl */
    float x = 1.76f;
    double y = 3.14;
    char s[] = "Hallo Welt";  /* Zeichenkette */
    return EXIT_SUCCESS;
}
```

Anweisungen, Ausdrücke

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main( int argc, char *argv[] )
5  {
6      int a = 3;
7      int b = 4;
8      int c = a + b;
9      int d;
10
11     d = a * b / c;
12     printf( "%d\n", d ); /* gib d als Ganzzahl aus */
13     return EXIT_SUCCESS;
14 }
```

Formatierte Ein-/Ausgabe mit `scanf` und `printf`

- Konsoleneingabe:
  `scanf(`«*Formatierungszeichenkette*»,«*Argument*»`...)`
- Konsolenausgabe:
  `printf(`«*Formatierungszeichenkette*»,«*Argument*»`...)`
- Die «*Formatierungszeichenkette*» kann folgende Platzhalter enthalten

„%d“ fügt an dieser Stelle eine Ganzzahl ein

„%f“ fügt an dieser Stelle eine Fließkommazahl ein

„%s“ fügt an dieser Stelle eine Zeichenkette ein

... es gibt noch mehr Formatierungsmöglichkeiten!

Achtung: Bei `printf` müssen die Parameter als Werte übergeben werden, bei `scanf` als Referenz!

Formatierte Ein-/Ausgabe, Beispiel

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main( int argc, char *argv[] )
5  {
6      char name[] = "Peter";
7      int alter = 20;
8      float gewicht;
9
10     printf( "Gewicht? " );
11     scanf( "%f", &gewicht );
12     printf( "Hallo %s, du bist %d Jahre alt und wiegst %f
           Kilo.", name, alter, gewicht );
13     return EXIT_SUCCESS;
14 }
```

Kontrollstrukturen

C bietet die folgenden Kontrollstrukturen an:

- for- Schleife
- do-while- Schleife
- while- Schleife
- if- Verzweigung
- if-else- Verzweigung
- switch- Verzweigung

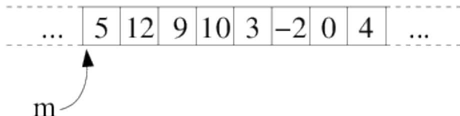
Wichtiger Unterschied zu Java, C++, etc.: In C89 dürfen Variablen nur am Anfang von Blöcken deklariert werden!, Ab C99 genauso wie in C++.

Funktionen und Prozeduren

- Prozedurdefinition:
  `void` «*Prozedurname*» («*Parameterliste*») { . . . }
- Funktionsdefinition:
  «*Rückgabetyp*» «*Funktionsname*»(«*Parameterliste*») { . . . }
- «*Parameterliste*» kann Folgendes beinhalten
  - `void`
  - «*Datentyp*» «*Parametername*»
  - «*Datentyp*» «*Parametername*», «*Parameterliste*»

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 void tue_dies( void )
 5 {
 6   printf( "Hallo" );
 7 }
 8
 9 void tue_das( void )
10 {
11   printf( " Welt\n" );
12 }
13
14 int main( int argc, char *
      argv[] )
15 {
16   tue_dies();
17   tue_das();
18   tue_dies(); /* again */
19
20   return EXIT_SUCCESS;
21 }
```

Eindimensionale Felder



- Reihung (Array) von Elementen gleichen Typs
- Feldgrenze wird nicht gespeichert
- Zugriffe sind ungeprüft

Eindimensionale Felder

- *Deklaration*:
  - «*Datentyp*» «*Variablenname*»[«*Größe*»]
  - «*Datentyp*» * «*Variablenname*»
- *statische Allokation*:
  - Größe des Feldes steht zur Kompilierungszeit fest
  - Allokation und Freigabe vom Compiler durchgeführt
  - statisch alloziierte Felder können direkt initialisiert werden (keine Größeninformation nötig)
- *dynamische Allokation*:
  - Feldgröße steht erst zur Laufzeit fest
  - Speicherbereich für diese Felder muss zur Laufzeit angefordert und wieder freigegeben werden

Eindimensionale Felder: Beispiel

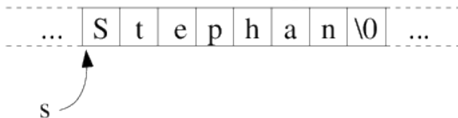
```c
#include <stdlib.h>
int main( int argc, char *argv[] )
{
    /* statisch alloziiertes Feld, 12 Eintraege */
    int a[12];
    /* dynamisch alloziiertes Feld, 12 Eintraege */
    int *b = (int *)malloc( 12 * sizeof(int) );
    /* weise 4. Elem von b das 3. Elem von a zu. */
    b[3] = a[2];
    /* keine Bereichspruefung in C! */
    a[0] = b[1000];
    free( b ); /* gibt Feld b frei */
    return EXIT_SUCCESS;
}
```

malloc & free

`void * malloc(int «Größe»)`

- …alloziiert einen zusammenhängenden Speicherbereich von «*Größe*» Bytes
- Die Größe eines Datentyps in Byte ermittelt `sizeof`.
- Der Rückgabewert ist ein spezieller Zeiger, dieser sollte (muss aber nicht) mit einem Cast in den richtigen Feldzeiger umgewandelt werden.
- Allgemein, um Feld vom Typ T und Größe n anzulegen:
  `(T *)malloc(sizeof(T)*n)`
- Mit `malloc` angeforderte Speicherbereiche können mit `free` freigegeben werden. Es gibt keine Garbage Collection!
- Wenn der Rückgabewert 0 ist, so war nicht genügend Speicher vorhanden um das Feld zu alloziieren.

Zeichenketten



- Eine Zeichenkette ist ein Feld von Zeichen (`char`) die durch das spezielle Zeichen `'\0'` beendet werden.
- Alle Operationen auf Zeichenketten lassen sich wie Feldoperationen behandeln.

$\Rightarrow$ . . . beispielsweise wie das Bestimmen der Zeichenkettenlänge.

Zeichenketten: Beispiel

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int string_laenge( char s[] )
4 {
5     int i = 0;
6     while( s[i] != '\0' )
7     {
8         i++;
9     }
10    return i;
11 }
12 char name[] = "Hallo Welt!";
13 int main( int argc, char *argv[] )
14 {
15    printf( "%d\n", string_laenge( name ) );
16    return EXIT_SUCCESS;
17 }
```

Mehrdimensionale Felder

- “geschachtelte” Felddefinition mit [ ]
- . . . werden intern als eindimensionales Feld umgesetzt. Hierfür wird ein entsprechender Index aus den mehrdimensionalen Indizes berechnet.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main( int argc, char *argv[] )
5 {
6     int matrix[2][3] = {{ 0,1,-1 },{ -1,0,2 }};
7     printf( "%d\n", matrix[0][2] );
8 }
```

Vom Code zum Programm - Beispiel

- modularisierter Code in mehreren C Dateien mit Header:

main.c:

```
1 #include "modul.h"
2
3 int main( int argc, char
      *argv[] )
4 {
5   printHello();
6 }
```

modul.h:

```
1 void printHello();
```

modul.c:

```
1 #include <stdio.h>
2
3 /* Wichtig */
4 #include "modul.h"
5
6 void printHello()
7 {
8   printf( "Hallo Leute\n" );
9 }
```

Vom Code zum Programm

- Was versteckt sich hinter modernen Entwicklungsumgebungen?
- Was passiert beim Bau des ausführbaren Programms?

Vom Code zum Programm

- Schritt 1: Kompilieren
  - wandelt jede einzelne C Datei in Machinencode
  - erzeugt Object Dateien (`modul.o` und `main.o`)
  - ⇒ Sind noch kein ausführbares Programm!
  - ⇒ In `main.o` fehlt die Information wo eigentlich die Funktion `printHello` liegt.

Vom Code zum Programm

- Schritt 2: Linking
  - kombiniert alle Object Dateien zu einem ausführbaren Programm
  - Linker findet `main.o` und merkt, dass das Symbol `printHello` fehlt
  - Linker findet `modul.o` mit dem Symbol `printHello`
  - `printHello` enthält `printf` was bisher unbekannt ist
  - Linker findet `printf` in sogenannter *Standard Bibliothek*
  - Alle unbekannte Funktionen und Symbole sind nun aufgelöst
  - Erzeugen des Programms da nun die genauen Adressen von `printHello` und `printf` bekannt sind

Vom Code zum Programm - GCC

- `gcc -c main.c`

  ⇒ Kompilieren von `main.c` zu `main.o`
- `gcc -c modul.c`

  ⇒ Kompilieren von `modul.c` zu `modul.o`
- FALSCH: `gcc -o sayhello main.o`

  ⇒ Linker beschwert sich: `undefined reference to printHello`
- RICHTIG: `gcc -o sayhello main.o modul.o`

  ⇒ Linken von `main.o` und `modul.o` zu ausführbaren Programm

Vom Code zum Programm - Vereinfachung

- Der GCC kann Kompilieren und Linken mit einem Befehl:

  ⇒ `gcc -o sayhello main.c modul.c`
- Problem: Wenn man nur in `modul.c` den Text ändert, wird auch `main.c` neu kompiliert. Das ist bei großen Programmen mit vielen C Dateien problematisch.
- Daher bei großen Programmen einzeln kompilieren und danach linken.
- Dabei helfen *Build Management Systeme* wie `make`.

Guter Code, Schlechter Code - Stil

- Achten Sie auf einen einheitlichen Stil[4] in Ihren Projekten
- Namensgebung: CamelCase **oder** Unterstrich_Notation
- Code Stil bringt bessere Lesbarkeit und Wartbarkeit von Code
  - Negativbeispiel: Schlechte Einrückung.
  - Syntaktisch belanglos in C aber kann Programmstruktur verdeutlichen und so sofort verständlich machen.

```
1 int funktion( int wert ){ int t=0; if( wert < 0 )
2 {
3     return 0;}
4 else
5 { for( t = 0; t<wert; t++)
6   {
7 printf("%d\n",t);
8 }}}
```

[4] http://google-styleguide.googlecode.com

Guter Code, Schlechter Code - Struktur

- Strukturieren Sie den Code in orthogonale Einheiten.
- Vermeiden Sie Codeduplikation mittels Funktionen in separaten Modulen (C Datei mit Header).

```
1 /* schlecht: */
2 void EierlegendeWollMilchSau( int a, char b, float d,
      double k, bool s );
3 /* besser: */
4 void eier( int wieviele );
5 void milch( float menge );
6 void wolle( bool gewaschen );
7 void fleischVomSchwein( double gewicht );
```

Guter Code, Schlechter Code - Dokumentation

- Dokumentieren Sie Ihren Code und Ihre Funktionen.
- Doxygen: http://www.doxygen.org
- Kommentieren Sie ihre Algorithmen und geben Sie Hinweise warum Sie ungewöhnliche Befehle nutzen.
- Eine gründliche Dokumentation hilft Ihnen und Ihren Kollegen!
- Vermeiden Sie “Write-Only-Code”!