Imperative Programmierung

- Imperativ (lat. *imperare*: anordnen, befehlen)
- Beschreibt *wie* etwas erreicht werden soll
- Elemente: Literale, Variablen, Ausdrücke, Anweisungen,. . .
- Beispiele für Imperative Programmiersprachen:
  Fortran, C, Pascal, Modula-2, . . .
- In dieser Übung am Beispiel von C

Programmiersprache C

- Erfunden von Kernighan und Ritchie, 1978
- Aktuell: ISO 9899:2011, kurz C11 (für Übung empfohlen) [3]
- Imperativ, prozedural (zum Teil funktional)
- Übersetzersprache: von Quelltext zu Maschinencode
- Grundlage für *sehr* viele weitere Programmiersprachen
- Syntax hat Ähnlichkeiten zu Java (vgl. MuP1)
- Einsatz häufig im Systemnahen und Embedded Bereich
- Online Nachschlagewerke:
  - http://www.cppreference.com
  - http://c-faq.com/
  - http://www.cprogramming.com/

[3] ISO 9899:2011 Dokument kostet Geld! Leicht älterer Entwurf: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf

Literatur

- Referenz (All Levels)
  - The C Programming Language, (2nd Ed) – B. Kernighan and D. Ritchie
  - C: A Reference Manual – Samuel P. Harbison and Guy R. Steele
  - C Pocket Reference (O'Reilly) – Peter Prinz, Ulla Kirch-Prinz
- Anfänger
  - Programming in C, (3rd Edition) – Stephen Kochan
  - C Programming: A Modern Approach – K. N. King
  - The C book – Mike Banahan, Declan Brady and Mark Doran
  - Practical C Programming, (3rd Ed) – Steve Oualline
  - C: How to Program, (6th Ed) – Paul Deitel & Harvey M. Deitel
- Fortgeschrittene
  - 21st Century C – Ben Klemens
  - C Interfaces and Implementations – David R. Hanson
  - The Standard C Library – P.J. Plauger
- Experten
  - Expert C Programming: Deep C Secrets – Peter van der Linden

“Hello World“ in C

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

Quelltext übersetzen

- Übersetzer von C-Quelltext in Maschinencode arbeiten wie folgt:
  1. Präprozessor Direktiven (beginnen mit '#') im Quelltext umsetzen:

     ```
     #include <stdio.h>
     #ifdef MY_SYMBOL
     ```

  2. Kompilieren: Syntax analysieren und Präprozessorausgabe in Maschinensprache (Objekt-Dateien) übersetzen
  3. Linken: Objekt-Dateien zu ausführbaren Datei / Bibliothek verbinden
- Die Systeme aus Präprozessor, Compiler und Linker werden oft zusammen einfach nur "Compiler" genannt
- C-Compiler: GCC, CLANG, ICC, PGCC, MSVS, ...
- Alle C Übungsaufgaben müssen wie folgt übersetzungsfähig sein:

  ```
  $ gcc -std=c11 -pedantic -Wall -Wextra
  ```

Kompilieren mit dem GCC

- Erzeugen einer ausführbaren Datei:

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

- ISO C11 Standardkonform übersetzen:

```
$ gcc -std=c11 -o hallo hallo.c
```

- Zusätzliche Warnungen anschalten:

```
$ gcc -Wall -Wextra -std=c11 -pedantic -o hallo hallo.c
$ ./hallo
Hello World!
```

Include- Direktive

Häufig benötigte Bibliotheken:

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

Bezeichner

- C Standard definiert character sets: Teilmenge von ASCII
- Im Vergleich zu Java (UTF-16), nur hoch experimentell UTF-8
- Bezeichner in C:
  An identifier is a sequence of nondigit characters (including the underscore _, the lowercase and uppercase Latin letters, and other characters) and digits.

- `stdbool.h`, `complex.h` definieren:

```
signed char      a = 0;                // sehr kurze Ganzzahl
short int        b = 1;                // kurze Ganzzahl
int              c = 2;                // Ganzzahl
long int         d = 123;              // lange Ganzzahl
long long int    e = 1234;             // sehr lange Ganzzahl
bool             f = false;            // Boolean
float            g = 1.76f;            // kurze Fliesskommazahl
double           h = 3.14;             // Fliesskommazahl
long double      i = 3.1415;           // lange Fliesskommazahl
float complex    j = 2.0 + 3.0*I;      // kurze Komplexzahl
double complex   k = csqrt( -1 );      // Komplexzahl
long double complex l = 1.0 + crealf(k)*I; // lange Komplexzahl
char             m = 'a';              // ein Zeichen
char             n[] = "Hallo Welt";   // Zeichenkette
```

Explizite Bitgröße

- `stdint.h` definiert desweiteren:

```
typedef signed char            int8_t;
typedef short int              int16_t;
typedef int                    int32_t;
typedef long int               int64_t;
typedef long long int          int64_t;
typedef unsigned char          uint8_t;
typedef unsigned short int     uint16_t;
typedef unsigned int           uint32_t;
typedef unsigned long int      uint64_t;
typedef unsigned long long int uint64_t;
```

Typkonvertierung

- Realisiert mit casts: `(ZielTyp) Ausdruck`

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

int main( int argc, char *argv[] )
{
    int a = (int) 3.14; // explicit cast
    int b = (int) 3.54; // explicit cast
    double c = 1;       // implicit cast
    printf( "%d\n", a );
    printf( "%d\n", b );

    return( EXIT_SUCCESS );
}
```

Typmodifizierer

- Alle Ganzzahltypen erlauben auch eine `unsigned` version

```
unsigned char         a = 0;
unsigned short int    b = 1;
```

- Konstanten vereinbart man mit `const` (nicht mit `#define`):

  ..const applies to the thing left of it. If there is nothing on the left then it applies to the thing right of it..[4]

```
const double          PI = 3.1415;
double const          C = 1;
const char            N[] = "Hallo Welt";
```

- Desweiteren gibt es noch die Modifizierer: static, volatile, register

[4] http://stackoverflow.com/questions/5503352/#5503393

C Literale

- Zeichenfolgen, bspw: '`123`' können durch Suffixe explizit typisiert werden
- Zusätzlich gibt es für Numerische-Literale verschiedene Basissysteme

```
int d = 42;       // decimal
int o = 052;      // octal
int x = 0x2a;     // hex
int X = 0X2A;     // hex
int b = 0b101010; // binary

unsigned int      a = 0x1u;
unsigned long int b = 1ul;
float             c = 1.0f
long double       d = 1.0l + 12.0e01l;
```

- Ausnahmen bspw. für `short int`: hat keinen Suffix

Operatoren

<table>
<tr><th colspan="7">Common operators</th></tr>
<tr><th>assignment</th><th>increment
decrement</th><th>arithmetic</th><th>logical</th><th>comparison</th><th>member
access</th><th>other</th></tr>
<tr>
<td>a = b
a += b
a -= b
a *= b
a /= b
a %= b
a &= b
a |= b
a ^= b
a <<= b
a >>= b</td>
<td>++a
--a
a++
a--</td>
<td>+a
-a
a + b
a - b
a * b
a / b
a % b
~a
a & b
a | b
a ^ b
a << b
a >> b</td>
<td>!a
a && b
a || b</td>
<td>a == b
a != b
a < b
a > b
a <= b
a >= b</td>
<td>a[b]
*a
&a
a->b
a.b</td>
<td>a(...)
a, b
(type) a
? :
sizeof
_Alignof (since C11)</td>
</tr>
</table>

Operatoren-Rangfolge

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Structure and union member access	
	->	Structure and union member access through pointer	
	(*type*){*list*}	Compound literal(C99)	
2	++ --	Prefix increment and decrement	Right-to-left
	+ -	Unary plus and minus	
	! ~	Logical NOT and bitwise NOT	
	(*type*)	Type cast	
	*	Indirection (dereference)	
	&	Address-of	
	sizeof	Size-of	
	_Alignof	Alignment requirement(C11)	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ -	Addition and subtraction	
5	<< >>	Bitwise left shift and right shift	

Operatoren-Rangfolge

<table>
<tr><th>Precedence</th><th>Operator</th><th>Description</th><th>Associativity</th></tr>
<tr><td>6</td><td>< <=
> >=</td><td>For relational operators < and ≤ respectively
For relational operators > and ≥ respectively</td><td rowspan="7">Left-to-right</td></tr>
<tr><td>7</td><td>== !=</td><td>For relational = and ≠ respectively</td></tr>
<tr><td>8</td><td>&</td><td>Bitwise AND</td></tr>
<tr><td>9</td><td>^</td><td>Bitwise XOR (exclusive or)</td></tr>
<tr><td>10</td><td>|</td><td>Bitwise OR (inclusive or)</td></tr>
<tr><td>11</td><td>&&</td><td>Logical AND</td></tr>
<tr><td>12</td><td>||</td><td>Logical OR</td></tr>
<tr><td>13</td><td>?:</td><td>Ternary conditional</td><td rowspan="2">Right-to-Left</td></tr>
<tr><td>14</td><td>=
+= -=
*= /= %=
<<= >>=
&= ^= |=</td><td>Simple assignment
Assignment by sum and difference
Assignment by product, quotient, and remainder
Assignment by bitwise left shift and right shift
Assignment by bitwise AND, XOR, and OR</td></tr>
<tr><td>15</td><td>,</td><td>Comma</td><td>Left-to-right</td></tr>
</table>

Anweisungen, Ausdrücke

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

Kontrollstrukturen

- C bietet die folgenden Kontrollstrukturen an:
  - `for`- Schleife
  - `do-while`- Schleife
  - `while`- Schleife
  - `if`- Verzweigung
  - `if-else`- Verzweigung
  - `switch`- Verzweigung
- Zusätzlich mit: `break`, `continue`, `return`, und `goto` Sprünge
- Nutzen Sie immer geschweifte Klammern
- Nutzen Sie für switch-Anweisungen immer einen `'default'`-case
- Endlos-Schleifen: `while( true ){..}` statt `for(;;){..}`

for

```
1 for( int i = 0; i < n; ++i ) {
2   printf( "Mir ist langweilig\n" );
3   if( i == 0 ) {
4     break;
5   }
6 }
```

do-while

```
1 int i = 0;
2 do {
3   printf( "Mir ist langweilig\n" );
4   i++;
5 }
6 while( i < 2 ); // how many printf's?
```

while

```
1 while( true ) { // will this terminate?
2   printf( "Mir ist langweilig\n" );
3   for( int i = n; i > 0; --i ) {
4     ...
5     break; // break stmt terminates smallest enclosing
           switch or iteration
6   }
7 }
```

if-else

```
1 if( only_one_stmt ) {
2   printf( "Still use curly braces!\n" );
3 }
4 else if( two_stmts ) {
5   printf( "You must use curly braces!" );
6 }
7 else {
8   printf( "You must use curly braces!" );
9 }
```

switch

```
1 switch( expr ) {
2   case FOOBAR : printf( "Foobar" );
3                 break;
4   case BARFOO :
5   case FARBOO : printf( "BARFOO or FARBOO" );
6                 break;
7   default : printf( "Whoopsie: Should not happen" );
8             break;
9 }
```

Funktionen und Prozeduren

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

Funktionen und Prozeduren

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void tue_dies( void ) {
5   printf( "Hallo" );
6 }
7
8 void tue_das( void ){
9   printf( " Welt\n" );
10 }
11
```

```
14 int main( int argc, char *
       argv[] ) {
15   tue_dies();
16   tue_das();
17   tue_dies(); /* again */
18
19   return EXIT_SUCCESS;
20 }
```

Variable Argumentliste: vgl. printf

```c
1 #include <stdio.h>
2 #include <stdarg.h>
3
4 int add(int first, ...) {
5   va_list list;
6   va_start( list, first );
7   int result = first;
8   int zahl = 0;
9   do {
10    zahl = va_arg( list, int );
11    result += first;
12  }
13  while( zahl != 0 );
14  va_end( list );
15  return result;
16 }
```

```c
17 int main( int, char** ) {
18   printf( "%d\n",
19     add( 99,0,66 ) ); //99
20   printf( "%d\n",
21     add( 99,66,0 ) ); //165
22   return 0;
23 }
```

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

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

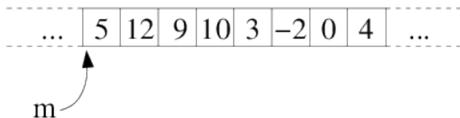
%d, %i	für eine Ganzzahl
%u	für eine Natürliche Zahl
%f, %e	für eine float, resp. double Fließkommazahl
%s	für eine Zeichenkette
%c	für ein Zeichen
%p	für einen Pointer
...	es gibt noch mehr Formatierungsmöglichkeiten

- Bei `printf` Argumente direkt, bei `scanf` als Referenz!

Formatierte Ein-/Ausgabe, Beispiel

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

Eindimensionale Felder



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

Eindimensionale Felder

- Statische Allokation:
  - «*Datentyp*» «*Variablenname*»[«*Größe*»]
  - Größe des Feldes steht zur Kompilierungszeit fest
  - Allokation und Freigabe vom Compiler durchgeführt
  - Statisch Felder können direkt initialisiert werden: `int a[] = {1, 2};`
- Dynamische Allokation:
  - «*Datentyp*» * «*Variablenname*»
  - Feldgröße steht erst zur Laufzeit fest
  - Speicherbereich für diese Felder muss manuell zur Laufzeit angefordert und wieder freigegeben werden

Eindimensionale Felder: Beispiel

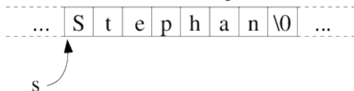
```
1  #include <stdlib.h>
2  int main( int argc, char *argv[] ) {
3    int a[12]; // statisches Feld: 12 Elemente
4    int *b = (int*)malloc(12*sizeof(int)); // dyn. Feld: 12 Elem
5
6    b[3] = a[2]; // weise 4. Elem. von b auf 3. Elm von a zu
7    a[0] = b[1000]; // KEINE Bereichsprüfung
8
9    printf( "a-length: %lu\n", sizeof( a ) ); // 48 Bytes
10   printf( "b-length: %lu\n", sizeof( b ) ); // 8 Bytes
11
12   free(b); // Gibt Speicher für b frei, a wird automatisch
        entsorgt
13   return( EXIT_SUCCESS );
14 }
```

Speichermanagement: `malloc` & `free`

```
void* malloc( int n );
```

- Alloziert zusammenhängenden Speicherbereich von n Bytes
- Die Größe eines Datentyps in Byte liefert: `sizeof`.
- Der Rückgabewert ist eine Speicheradresse vom Typ: `void*`
- `void*` ist untypisiert und sollte daher mit Cast konvertiert werden
- Allgemein, um Feld vom Typ `T` und Größe `n` anzulegen:
  `(T*) malloc( sizeof( T ) * n );`
- Wenn Rückgabewert `0`: war nicht genügend Speicher verfügbar
- Einmal mit `malloc` angeforderte Speicherbereiche müssen immer mit `free` wieder freigegeben werden
- Es gibt keine Garbage Collection!

Zeichenketten



- Zeichenketten sind Felder von Zeichen (`char`)
- Da keine Länge verfügbar, einigt man sich auf `NULL`-Terminierung

```
1 char pw[] = "secret"; // automatisch NULL-terminiert
2 char *buffer = (char*) malloc( sizeof( char ) * 27 );
3 for( char ch = 65; ch < 91; ++ch ) {
4   buffer[ ch - 65 ] = ch;
5 }
6 buffer[26] = '\0'; // manuelle NULL-terminierung
7 buffer[26] = NULL; // ist dasselbe
8 buffer[26] = '0';  // ist falsch
```

- Beispiel: Längenbestimmung einer `'\0'`-terminierten Zeichenkette

Zeichenketten: Beispiel

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

- Realisiert über geschachtelte Felddefinitionen

```
1   int matrix[2][3] = {{ 0,1,-1 },{ -1,4,2 }};
2   int natrix[2*3] = { 0, 1, -1, -1, 4, 2 };
3   int** oatrix = (int**)malloc( sizeof( int* ) * 2 );
4   int c = 0;
5   for( int row = 0; row < 2; ++row ) {
6     oatrix[row] = (int*) malloc( sizeof( int ) * 3 );
7     for( int col = 0; col < 3; ++col, ++c ) {
8       oatrix[row][col] = c;
9     }
10  }
11  printf( "%d\n", matrix[ 1 ][ 1 ] );     // 4
12  printf( "%d\n", natrix[ 1 * 3 + 1 ] ); // 4
13  printf( "%d\n", oatrix[ 1 ][ 1 ] );     // 4
14  for( int row = 0; row < 2; ++row ) {
15    free( oatrix[row] );
16  }
17  free( oatrix );
```

Beispiel mehrere Dateien

- Modularisierter Quelltext in mehreren C-Dateien und Header-Dateien

main.c:

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

modul.h:

```
1 #ifndef MODUL_H //Headerguard
2 #define MODUL_H
3
4 void printHello();
5
6 #endif  // MODUL_H
```

modul.c:

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

Mehrere Dateien Kompilieren mit dem GCC

- `main.c` und `modul.c` in Maschinencode *übersetzen*

```
$ gcc -Wall -Wextra -std=c11 -pedantic -c main.c
$ gcc -Wall -Wextra -std=c11 -pedantic -c modul.c
```

- Maschinencode und Bibliotheken zur ausführbaren Datei *linken*

```
$ gcc -o sayhello main.o
..undefined reference to printHello..
$ gcc -o sayhello main.o modul.o
```

- Der GCC kann Kompilieren und Linken mit einem Befehl:

```
$ gcc -Wall [...] -pedantic -o sayhello main.c modul.c
```

- Bei vielen Dateien: Buildsysteme wie `make`