

Programmiertechnik II

C

C: Eine Übersicht

- Geschichte
- Programmstruktur
- Semantik
- Anweisungen
- Lexik
- Datentypen
- Ausdrücke
- Standardbibliothek

Geschichte

- Entwickelt bei AT&T Bell Labs, ursprünglich mit und für UNIX
 - zwischen 1968 und 1972; Dennis Ritchie
 - Name “C”: Nachfolger von “B” (Ken Thompsons BCPL-Impementierung)
 - 1978: Kerninghan, Ritchie (K&R): The C Programming Language
 - Einführung von struct, long int, unsigned int
 - Ersetzung von =+ durch +=
- 1989: ANSI C (X3.159-1989)
 - Komitee X3J11 seit 1983
 - 1990 normiert von ISO (ISO/IEC 9899:1990)
 - 1995 Amendment 1 (ISO/IEC 9899:1990/AM1:1995)
 - “ANSI C”, “C89”, “ISO C”, “standard C”
- 1999: C99 (ISO/IEC 9899:1999)
 - inline, neue Datentypen, Felder variabler Länge, ...

Ein Beispiel

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

```
int main()
```

```
{
```

```
    printf("Hello, world\n");
```

```
}
```

- Übersetzung mittels
cc hello.c
- Ergebnis: a.out

Programmstruktur

- “schwache” Modulstruktur: *translation units*
- Implementierungsdateien (*source files*, .c)
- Deklarationsdateien (*header files*, .h)
- Objektdateien (*object files*, UNIX: .o, Windows: .obj)
- Bibliotheksdateien
 - statische Bibliotheken (*static libraries*, UNIX: .a, Windows: .lib)
 - dynamische Bibliotheken (*shared libraries, dynamic link libraries*, DLLs, UNIX: .so, .sl, Windows: .dll)
 - Windows: *import libraries* (.lib)
- Eintrittspunkt: Funktion main
 - Rückgabetyt int
 - Erreichen des Endes von main: implizit “return 0;”
 - Rückgabewert i.d.R. im Betriebssystem zugänglich, 0 bedeutet “Erfolg”
 - Signatur: int main(void); oder int main(int argc, char*argv[]);

Übersetzungsprozess

- Präprozessor: Auflösung von Makros, Headerfiles
 - #include, #define, #if, #ifndef, #elif, #else, #endif, #error, #line
 - Kommandozeilenoptionen -D, -I (i)
 - Ergebnis: vorverarbeiteter Quelltext (*preprocessor output*)
- Compiler: Übersetzung des Quelltexts in Maschinencode
 - gcc: Übersetzung des Quelltexts in Assemblercode, Generierung des Maschinencodes durch separaten Assemblerlauf
 - Option -o: Festlegen der Ausgabedatei (Standard: a.out)
 - Option -c: Verzicht auf Linkerlauf (Ergebnis: <datei>.o)
 - Option -S: Verzicht auf Assemblerlauf (Ergebnis: <datei>.s)
 - Option -E, -P: Verzicht auf Compilerlauf (Präprozessorausgabe auf Terminal)
- Linker: Integration verschiedener Objektfiles und Bibliotheken in ein Programm (Windows: .exe, UNIX: keine Endung)
 - Optionen -l (ell), -L

Semantik

- Zwei Implementierungsversionen: *hosted* und *free-standing*
 - *free-standing* für Verwendung in eingebetteten Systemen (kein normierter Eintrittspunkt, keine normierte Ein-/Ausgabe)
- *hosted*: Programmausführung besteht in Aufruf von `main()`
 - Programm erhält Kommandozeilenargumente in `argc` (Anzahl) und `argv` (Argumente); `argv[0]` ist der Programmname
- Bedeutung des Programms durch die Abfolge der Seiteneffekte (*side effects*) definiert
 - Änderungen von Variablen, Zugriff auf als **volatile** erklärte Variablen, Ein-/Ausgabe auf Dateien
- Programmausführung unterteilt in *sequence points*:
 - i.d.R. ist jede primitive Anweisung (jedes Semikolon) ein *sequence point*
 - Seiteneffekte sind zwischen *sequence points* ungeordnet

Semantik (2)

- undefiniertes Verhalten (*undefined behaviour*)
 - Verhalten bei Programmfehlern; die Implementierung darf sich beliebig verhalten
 - Beispiel: Zugriff auf nicht-initialisierte Variablen oder auf freigegebenen Speicher; mehrfache Veränderung eines Wertes zwischen zwei *sequence points*
 - übliches Verhalten: Programmabsturz (sofort oder später); Verwendung scheinbar zufälliger Werte
- implementierungsabhängiges Verhalten (*implementation-defined behaviour*)
 - Norm lässt verschiedene Möglichkeiten offen; Implementierung muss festlegen, welches Verhalten gilt
 - Beispiel: Größe von `short`, `int`, `long`; zusätzliche `main`-Varianten (z.B. POSIX: `main(argc, argv, envp)`); Bedeutung von **register**

Anweisungen (1)

- Anweisungssyntax hat C++, Java, C# beeinflusst
- **if**(*condition*) *statement* **else** *statement*
 - *condition* kann “beliebigen” Typ haben
 - K&R, C89: kein Datentyp bool
- **for**(*init*;*condition*;*step*) *statement*
 - K&R, C89: *init* erlaubt keine Variablendeklarationen
- **while**(*condition*) *statement*
- **do** *statement* **while**(*condition*);
- **switch**(*condition*) { case-block }
- Blöcke: { *declarations statements* }
 - C99: Deklarationen und Anweisungen dürfen gemischt werden

Anweisungen (2)

- Ausdrucksanweisungen: *expression*;
 - Zuweisungen, Funktionsruf
- **return**; oder **return** *value*;
- **break**;
- **continue**;
- Sprunganweisung: **goto** *label*;
 - Definition eines labels vor beliebigen Anweisungen:
label: statement
 - Kerninghan, Ritchie: “infinitely abusable”, Verwendung zur Fehlerbehandlung oder zum Verlassen verschachtelter Schleifen

Lexik

- Zeilenstruktur ist nur für Präprozessor relevant
 - Eine Präprozessoranweisung muss auf einer Zeile stehen
 - Folgezeilen können durch \ in der Vorgängerzeile festgelegt werden
- Kommentare: /* Kommentar */
 - C99: // Zeilenendekommentare
- Lexik für Bezeichner, Zahlen, Zeichenliterale, Zeichenkettenliterale wie Java
 - \uXXXX sowie Nicht-ASCII-Bezeichner nur in C99

Primitive Datentypen

- integrale Typen, Gleitkommatypen, void
- integrale Typen: char, short, int, long int, Aufzählungstypen (enums)
 - C 99: long long int
 - “int” bei long-Typen optional: long, long long
 - Vorzeichenbehaftete Versionen: signed char, signed short, ...
 - short, int, long standardmäßig signed; Vorzeichenbehaftung von char ist *implementation-defined*
 - Vorzeichenlose Versionen: unsigned char, unsigned short, ...
 - “int” bei “unsigned int” optional: unsigned
 - enum Color {red, green, blue};
 - Optional mit Werten
- Gleitkommatypen: float, double
 - C99: long double

Zusammengesetzte Typen

- struct: Datensatz aus mehreren Feldern

```
struct Point{  
    double x;  
    double y;  
};
```

- union: Vereinigung mehrerer Alternativen

```
union reply {  
    struct accepted_reply RP_ar;  
    struct rejected_reply RP_dr;  
};
```

- Lesezugriff nur für die aktuell gültige Alternative definiert; aktuell gültige Alternative muss aus Zusammenhang bekannt sein
- Zugriff auf Elemente von struct oder union über `variable.element`

Zusammengesetzte Typen (2)

- Zeigertypen Pointertypen (*pointers*): T^*
 - `int *x;`
 - Werte von Pointertypen sind Adressen von Objekten
 - Bildung der Adresse: `pointer = &variable`
 - Alternativ: Initialisierung des Pointers mit 0 (*null pointer*)
 - K&R: Statt 0 schreibt man NULL
 - Zugriff auf gespeicherten Werte: `variable2 = *pointer;`
 - Zugriff auf struct-Pointer: `a->b` bedeutet `(*a).b`
- Array-Typen: $T[n]$
 - `int x[10]; /* nicht: int[10] x; */`
 - Wertetyp: Variable enthält selbst den Speicher für die Elemente
 - Übergabe von Arrays an Funktionen: Array “zerfällt” (*decays*) in Pointer
 - Indizierung beginnt mit 0

Arrays und Pointer

- Pointer kann “in ein Array hinein” zeigen:

```
int x[10];  
int *p;  
p = &x[5];  
*p = 4; /* Genauso wie x[5] = 4; */
```
- Array selbst kann automatisch in Pointer (auf erstes Element) konvertiert werden
 - $p = x$; bedeutet $p = \&x[0]$;
- Adressarithmetik: Addition (pointer+int) ergibt Zeiger; (pointer-pointer) ergibt int
 - Zählung in Elementen des zugrundeliegenden Felds:
 - ```
int *p = &x[5]; int *q = &x[1]; /* p-q ist 4 */
```
- Verwendung von Pointern wie Felder:
  - $p[n]$  bedeutet  $*(p+n)$

# Arrays und Pointer: Ein Beispiel

```
void verdoppeln(double *x, int count)
{
 int i;
 for(i=0; i < count; i++)
 x[i] *= 2;
}
```

```
int main()
{
 double zahlen[8] = { 2, 3, 5, 7, 11, 13, 17, 19};
 double *p = zahlen+4;
 verdoppeln(p, 2);
 verdoppeln(zahlen, 8);
}
```



# Funktionen

- Deklaration und Definition
  - Deklaration ist optional; üblicherweise in Headerfile
- Deklaration: `R func(ptyp, ptyp);`
  - `void foo(int, double);`
  - R: Rückgabetyt
  - ptyp: Parametertypen (Parameternamen in Deklaration optional)
- Definition: `R func(ptyp param, ptyp param) { body }`
- `static`: Funktion ist nur innerhalb der *translation unit* sichtbar
- Funktionszeiger: Zeiger auf Funktionen
  - Wie Funktionsdeklaration, nur `(*var)` anstelle von Funktionsname

```
void (*pfunc)(int, double); /* Variable pfunc ist Zeiger auf Funktion */
pfunc = &foo; /* Variable pfunc zeigt auf foo */
(*pfunc)(3, 7.5); /* Aufruf von pfunc */
```

# Typ-Aliases

- Benennung von Typen durch typedef
  - Wie Variablendeklaration, nur Typname anstelle von Variablenname

```
typedef int age_t; /* Typen enden oft per Konvention mit _t */
typedef struct Person {
 char* name;
 age_t age;
} Person; /* Statt "struct Person" nun auch "Person" möglich */
```

# Variablen

- lokale Variablen: Definiert an Blockanfang
  - Speicher bei Eintritt in den Block alloziert (Stack-Variablen, redundantes Schlüsselwort **automatic**)
  - sichtbar bis Blockende
  - Initialwert ist *undefined*, explizite Initialisierung bei Deklaration möglich
- globale Variablen: Definiert außerhalb von Funktionen
  - automatisch null-initialisiert, explizite Initialisierung möglich
  - Deklaration von globalen Variablen durch **extern** möglich
- block-statische Variablen: Definiert als **static** innerhalb eines Blocks
  - sichtbar nur im Block
  - Lebenszeit wie globale Variablen

# Ausdrücke

- Operatorsyntax und Vorrang wie Java
  - kein >>>, instanceof
  - Pointerdereferenzierung: \* (unär), ->
  - Adressbildung: &
  - Bestimmung der Größe (in Bytes, Typ size\_t): sizeof
    - int x;
    - int y = sizeof(x); /\* oder sizeof x; \*/
  - Hintereinanderberechnung von Ausdrücken: ,
    - int a; a = (f(), g(), h());
    - , stellt *sequence point* dar
    - Wert des Gesamtausdrucks ist Wert des letzten Teilausdrucks
- Funktionsruf
  - Reihenfolge der Parameterauswertung *implementation-defined*

# Standardbibliothek

- Deklarationen organisiert in verschiedene Headerfiles
- Implementierung i.d.R. in einer einzigen Bibliothek
  - UNIX: libc.a, libc.so, i.d.R. automatisch in Linkerkommando einbezogen (explizit durch -lc angebbbar)
    - i.d.R. separate Bibliothek für <math.h>; libm.a, libm.so, explizit einzubinden durch -lm
  - Windows: C-Bibliothek compilerabhängig; statisch oder dynamisch
  - Microsoft-Compiler, dynamisch: msvcr.dll, msvcr4.dll, msvcr7.dll, msvcr71.dll, msvcr8.dll, msvcr.lib (import library)
    - Auswahl der Bibliothek automatisch und durch Compilerflags.  
z.B. /MD, /MDd

# Speicherverwaltung

- Speicherarten: global, lokal, dynamisch
  - global: Speicher für globale Variablen wird bei Programmstart bereitgestellt und ist bis Programmende gültig
    - explizit vorinitialisiert oder null-initialisiert
  - lokal: Speicher für lokale Variablen wird auf (i.d.R.) auf Prozessorstack bereitgestellt und ist bis Blockende gültig
    - explizit initialisiert oder uninitialized
  - dynamisch: Explizite Allokation, Initialisierung, Freigabe
    - Halde (*heap*)

# Dynamische Speicherverwaltung

- Funktionen in <stdlib.h>
- Allokierung: malloc, calloc
  - `void* malloc(size_t n); /* uninitialisierter Speicher */`
  - `void* calloc(size_t n); /* null-initialisierter Speicher */`
  - Größe i.d.R. aus sizeof-Operator bestimmt (Parameter n: Zahl der Bytes)
- ```
int *new_int_array(int no_elements) {  
    int *result = (int*)malloc(no_elements * sizeof(int));  
    for(int i=0; i < no_elements; i++) result[i] = i;  
    return result;  
}
```
- Ergebnis ist 0, wenn kein Speicher mehr zur Verfügung steht
- Freigabe: `void free(void* block);`
- Größenveränderung: `void* realloc(void* block, size_t n);`
 - Ergebnispointer u.U. verschieden von Eingabepointer

Zeichenkettenverarbeitung

- Zeichenketten i.d.R. als `char[]` repräsentiert
 - keine explizite Längenspeicherung
 - Länge u.U. separat gespeichert
 - üblich: Länge ergibt sich implizit durch “Stringendezeichen” `'\0'`
 - null-terminierte Zeichenketten
- Zeichenketten als Variablen oder Parameter: `char*`
 - Zeiger ist Zeiger auf erstes Zeichen
- Zeichenkettenlitterale enthalten automatisch Null-Terminierung:
 - `char s[] = “Hallo”; /* sizeof(s) == 6 */`
- Zeichenkettenlitterale sind nicht änderbar (`const`), können aber in `char*` konvertiert werden
 - `char *t = “Hello”;`

Zeichenkettenverarbeitung (2)

- Bibliotheksfunktionen in `<string.h>`
- Stringlänge: `size_t strlen(char*)`;
 - zählt bis zum abschließenden `'\0'`, ausschließlich des `\0`
 - `strlen("Hello") == 5`
- Stringkopie: `char* strcpy(char* ziel, char *quelle)`;
 - kopiert Zeichen von `quelle` nach `ziel`, bis einschließlich terminierendem `'\0'`
 - Ergebnis: `ziel`
 - Speicher an Adresse `ziel` muss groß genug für Zeichenkette an Adresse `quelle` sein (sonst: *buffer overflow*, undefiniertes Verhalten)
- Stringkopie: `char* strdup(char* quelle)`;
 - Alloziert neuen String mittels `malloc` (Größe: `strlen(quelle)+1`)
 - Funktion nicht Teil von Standard-C, sondern nur in POSIX definiert
 - Funktion liefert neuen String (0 falls `malloc` 0 liefert)

Zeichenkettenverarbeitung (3)

- Stringverkettung: `char* strcat(char* s1, char* s2);`
 - Inhalt von s2 wird an Ende von s1 angefügt
 - Speicher an Adresse s1 muss groß genug für Ergebnis sein
- Stringvergleich: `int strcmp(char* s1, char* s2);`
 - Vergleicht s1 und s2 lexikographisch
 - Ergebnis: <0 (string1 < string2), $=0$ (string1 gleich string2), >0 (string1 > string2)
- Weitere Funktionen in `<string.h>`: `memcpy`, `memcmp`, `strcoll`, `strxfrm`, `memchr`, `strchr`, `strcspn`, `strpbrk`, `strstr`, ...

Ein-/Ausgabe

- Funktionen deklariert in `<stdio.h>`
- Vordefinierter Typ: `FILE`
 - struct-Typ, üblicherweise nur als `FILE*` verwendet
 - Datenfolge (*stream*)
 - enthält Position in Datei, Puffer, Dateiendeanzeige, ...
- Vordefinierte *streams*: `stdin`, `stdout`, `stderr` (alle `FILE*`)
- Öffnen von Dateien: `FILE* fopen(char* name, char* mode);`
 - Modi: `r`, `w`, `a`, `rb`, `wb`, `ab`, `r+`, `w+`, `a+`, `r+b (rb+)`, `r+w`, `r+a`
 - liefert `0 (NULL)` bei Fehler
- Schließen: `int fclose(FILE*);`
 - liefert `0` bei Erfolg, `EOF` bei Fehler
- Positionierung in Datei: `ftell`, `fseek`, `fgetpos`, `fsetpos`
- Übertragung des Puffers an Umgebung: `int fflush(FILE*);`
 - `0` bei Erfolg, `EOF` bei Fehler

Unformatierte Ein/Ausgabe

- Verarbeitung von Speicherblöcken
 - Allgemeiner: Arrays von Elementen fester Größe
- `size_t fread(void *ziel, size_t groesse, size_t anzahl, FILE* eingabe)`
 - Lesen von *anzahl* Datenblöcken der Größe *groesse* aus Datei *eingabe* in den Speicher an Adresse *ziel*
 - Ergebnis: Zahl der gelesenen Elemente (evtl. < anzahl bei Fehler oder Dateiende)
- `size_t fwrite(const void *quelle, size_t size, size_t anzahl, FILE* ausgabe)`
 - Schreiben von Blöcken auf Datei *ausgabe*
 - Ergebnis: Zahl der erfolgreich geschriebenen Blöcke

Formatierte Ausgabe

- `int fprintf(FILE *ausgabe, char *format, ...);`
 - Formatzeichen: `%<flags><width><precision><modifier><typ>`
 - `typ`: `d, i` (int), `o, u, x, X` (unsigned int), `f, F, e, E, g, G, a, A` (double), `c` (int/char), `s` (char*), `p` (void*), `n, %`
 - `modifier`: `hh, h, l, ll, j, z, t` (für integrale Typen), `L` (für Gleitkommatypen)
 - z.B. `%ld` für long int
 - `precision`: Zahl der Nachkommastellen, minimale Zahl der Ziffern, maximale Breite einer Zeichenkette
 - z.B. `%.3f` (3 Nachkommastellen), `%.4X` (4 Hexadezimalziffern)
 - `width`: minimale Breite des gesamten Ausgabefelds
 - z.B. `%10s` (Zeichenkette mit mindestens 10 Spalten)
 - `flags`: `-` (Linksausrichtung), `+` (stets Ausgabe eines Vorzeichens), `Leerzeichen` (Ausgabe eines Leerzeichens als positives Vorzeichen), `#` (alternative Form), `0` (Ausgabe führender Nullen)

Fehlerbehandlung bei Ein/Ausgabe

- Funktionen liefern int-Werte, um Fehler anzuzeigen
- Ein/Ausgabe auf FILE*: Test, auf Fehler oder Dateiende mit separaten Funktionen:
 - `int feof(FILE*);`
 - `int ferror(FILE*);`
- viele Funktionen (insbesondere POSIX-Funktionen) setzen Variable `errno`:
 - deklariert in `<errno.h>`
 - Standard-C kennt nur Fehler `EDOM`, `EILSEQ`, `ERANGE`
 - POSIX: auch `EPERM`, `EACCESS`, `EEXIST`, `ENOENT`, ...

Weitere Funktionen der Standardbibliothek

- `<assert.h>`: Assertions
- `<math.h>`: mathematische Funktionen
 - C99: `<complex.h>`
- `<ctype.h>`: Klassifikation von Zeichen
 - `<wctype.h>`
- `<limits.h>`: Bestimmung von Wertebereichen
- `<setjmp.h>`: “Nicht-lokale” Sprünge
- `<signal.h>`
- `<stdarg.h>`: Behandlung variabler Argumentlisten
- `<time.h>`: aktuelle Zeit, Formatierung von Zeit
- ...

Arbeitsweise des Präprozessors

- **#include: Einschließen von Dateien**
 - zwei Formen: `#include <datei>` und `#include "datei"`
 - üblich: erste Form (`<>`) für Systemheaderfiles (z.B. in `/usr/include`), zweite Form für Headerfiles des Programms (z.B. aktuelles Verzeichnis)
- **#define: Definition von Makros**
 - Objektartige Makros: `#define NAME INHALT`
 - `#define MAX_STUDENTS 1000`
 - Verwendung: `struct Student studenten[MAX_STUDENTS];`
 - Funktionsartige Makros: `#define FUNC(params) INHALT`
 - Aktuelle Makroargumente werden in `INHALT` ersetzt
 - `#define matrikel(S) studenten[S].matrikel_nummer`
 - Verwendung: `matrikel(220)`

Arbeitsweise des Präprozessors (2)

- Spezialsymbole in Makroinhalt: ## und #
 - #ARGUMENT: Ersetzung des Makroarguments durch eine Stringkonstante

```
#define MAKE_VARIABLE(N) char* N = #N  
MAKE_VARIABLE(foo);  
MAKE_VARIABLE(bar);
```
 - token1##token2: Verkettung der Terminalsymbole (*token pasting*)

```
#define MAKE_VARIABLE(N) char* string_##N = #N;
```

Arbeitsweise des Präprozessors (3)

- Bedingte Übersetzung: `#if`, `#ifdef`, `#ifndef`, `#elif`, `#endif`
- `#if <Ausdruck>`
 - arithmetische Operationen auf Konstanten, z.B.
`#if MAX_STUDENTS > 2000`
...
`#endif`
 - vordefinierter Operator `defined(name)`
`#if defined(__GNUC__)`
...
`#elif defined(_MSC_VER) || defined(__BORLANDC__)`
...
`#endif`
- `#ifdef N`: gleichbedeutend mit `#if defined(N)`
- `#ifndef N`: `#if !defined(N)`

Arbeitsweise des Präprozessors (4)

- vordefinierte Makros: `__LINE__`, `__FILE__`, `__DATE__`,
`__TIME__`, `__STDC__` (1), `__STDC_VERSION__`
(199901L)
- weitere vordefinierte Makros *implementation-defined*

Bibliotheken

- Kombination mehrerer Objektfiles in eine Datei
- Statische Bibliotheken: Verwendung ausschließlich durch den Linker
 - Beim Linken werden die benötigten Objektdateien in das Programm hineinkopiert
 - UNIX: `ar(1)`
 - Windows (MSC): `lib.exe`
- Dynamische Bibliotheken: Verwendung auch zur Programmlaufzeit
 - Beim Linken wird lediglich ein Verweis auf die Bibliothek in das Programm eingetragen; bei Programmstart wird Bibliothek “dynamisch” zu Programmlauf hinzugefügt
 - UNIX (GCC): `gcc -shared`
 - Windows (MSC): `link.exe /DLL` (erzeugt `.DLL` + *import library*)