Programmiertechnik 1

Unit 11: Programmiersprache C –
Ein- und Ausgabe (libc)

Ablauf

- Standard Ein- und Ausgabe – stdio
- Formatierte Ausgabe – printf
- Argumentlisten variabler Länge – varargs
- Formatierte Eingabe – scanf
- Dateizugriff
- Fehlerbehandlung – stderr und exit
- Zeilenweise Ein- und Ausgabe
- Weitere Funktionen aus der libc

C-Standardbibliothek

- Ein- und Ausgabe sind nicht Bestandteil der C Sprachdefinition
- Implementiert in Standardbibliothek – stdlib
  - Präzise Definition durch ANSI-Standard
  - Jede ANSI-konforme C-Implementierung unterstützt Standardbibliothek
  - → portabel Programmieren
- Einfaches Modell für Ein- und Ausgabe von Text
  - Behandlung CR/LF ←→ NL
  - Keine durchgängige Unterstützung für Unicode bei stdio
  - → aber mit getw() lassen sich int-weite Worte lesen (wchar_t)
- Standardbibliothek
  - Deklarationen in Header-Dateien
  - <stdio.h, <string.h>, ctype.h>
  - Binden mit Linker-Option –lc (libc.a)

Standard Ein- und Ausgabe: stdio

- Jedes C-Programm hat drei geöffnete Dateiströme zur Hand:
  - stdin – Standardeingabe
    - Häufig mit Terminal verbunden (/dev/stdin)
    - Hier erscheinen „Tastatureingaben“
  - stdout – Standardausgabe
    - Häufig mit Terminal verbunden (/dev/stdout)
    - Hier werden „Bildschirmausgaben“ geschrieben
  - stderr – Standard-Fehlerausgabe
    - Häufig mit Terminal verbunden (dev/stderr/
    - Kann separat von stdout umgelenkt werden
- Diese Ströme werden vom Betriebssystem (der shell) für ein Programm in Ausführung geöffnet
  - Umlenkung, pipes
  - prog < infile; /bin/date | prog; prog >outfile 2>errfile;

Lesen von stdin

- Lesen eines Zeichens:
  - int getchar(void);
    - Liefert nächstes Zeichen oder EOF zurück
    - EOF ist in <stdio.h> definiert (i.d.R. -1)
    - Daher ist Rückgabewert von getchar() vom Typ int, nicht char
  - #include <stdio.h>
    - Alle Programme die stdio verwenden finden hier ihre Deklarationen
    - Stdio.h lebt in /usr/include
- Alternativen:
  - int fgetc(FILE *stream);
  - int getc(FILE *stream);
    - Wie fgetc – allerdings als Makro das inline ersetzt wird
  - int getw(FILE *stream);
    - Lesen eines Wortes (int)
    - Behandlung von Mehr-Byte-Zeichen

Ausgabe von Zeichen

- Zeichenweise Ausgabe – Zeichen sind int (!)
  - int putchar(int c);
  - int fputc(int c, FILE *stream);
    - int putc(int c, FILE *stream); - Makro – inline Ersetzung
  - int putw(int w, FILE *stream);
- Beispiel:

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

main() {
      int c;

      while ((c = getchar()) != EOF)
            putchar( tolower(c) );
      return 0;
}
```

tolower.c

Formatierte Ausgabe – printf

- printf übersetzt interne Werte in Zeichen
  - int printf(char * format, arg_1, arg_2, ...);
  - Konvertiert, formatiert und druckt Argumente auf stdout
  - format-String steuert Umwandlung
  - Gibt Zahl der gedruckten Zeichen zurück

- format-String
  - Enthält „normale Zeichen“ → werden nach stdout kopiert
  - Konvertierungsspezifikationen – beginnen mit ’%’ und enden mit Konvertierungszeichen
  - Dazwischen:
    - ’-’ – für Linksjustierung
    - Zahl n für die minimale Feldbreite (wird mit Leerzeichen aufgefüllt)
    - ’.’ – trennt Feldbreite von Präzision
    - Zahl m für die maximale Zahl von Zeichen (oder Ziffern)
    - ’h’ oder ’l’ um Integer als short oder long auszugeben

printf-Konvertierungen

Zeichen	Argumenttyp, gedruckt als
d, i	int; Dezimalzahl
o	int; vorzeichenlose Oktalzahl (ohne führende Null)
x, X	int; vorzeichenlose Hexadzimalzahl (ohne führendes 0x oder 0X), mit abcdef oder ABCDEF für 10,..,15
u	int; vorzeichenlose Dezimalzahl
c	int; einzelnes Zeichen
s	char *; gibt Zeichenkette aus bis '\0' oder bis zur durch Präzision angegebenen Zahl
f	double; [-]m.dddddd, Zahl der d's ist durch Präzision gegeben (default: 6)
e, E	double; [-]m.ddddddde±xx oder [-]m.ddddddE±xx, wobei die Zahl der d's durch Präzision gegeben ist (default: 6)
g, G	double; wie %e oder %E wenn Exponent < -4 oder Exponent ≥ Präzision, sonst wie %f; Nachfolgende Nullen und nachfolgender Dezimalpunkt werden nicht gedruckt
p	void *; Zeiger (implementierungsabhängige Repräsentation)
%	Argument soll nicht konvertiert werden – gibt '%' aus

printf-Konvertierungen (contd.)

- Feldbreite und Präzision können als '*' angegeben werden
  - Wert wird aus nächsten Argument berechnen (muss int sein)
- Beispiele
  - Geben "hello, world" aus (12 Zeichen) (: als Begrenzer)

```
:%s:          :hello, world:
:%10s:        :hello, world:
:%.10s:       :hello, wor:
:%-10s:       :hello, world:
:%.15s:       :hello, world:
:%-15s:       :hello, world   :
:%15.10s:     :     hello, wor:
:%-15.10s:    :hello, wor     :
```

Falle:

```
/* schlägt fehl wenn s % enthält */
printf( s );

/* richtig *(
printf( "s", s );
```

  - printf wertet format-String aus um Zahl und Typ der Argumente zu bestimmen
    - Fehlerhafte Ausgabe bei falsche Argumentzahl oder –typ
  - Formatierung von Zeichenketten:
    - int sprintf(char * string, char * format, arg_1, arg_2, ...)

Variable Argumentlisten

- Deklaration von printf: int printf(char *fmt, ...);
  - ... ist C-Syntax für Deklaration einer variablen Argumentliste
  - <stdarg.h> enthält Makrodefinitionenen für Argumentverarbeitung
    - In der aufgerufenenen Funktion
  - va_list - Typ
    - Deklaration einer Variablen, die auf Argumente verweist
  - va_start - Makro
    - Initialisierung eines Argumentzeigers ap
  - va_end – Makro
    - Abschluss der Argumentverarbeitung
- Variable Argumentlisten funktionieren weil...
  - ... in C der Aufrufer Argumente auf dem Stack platziert
  - ... der Aufgerufene Zahl und Typ der Argumente zur Laufzeit nicht überprüft
  - ... der Aufrufer für das „Aufräumen“ des Stack verantwortlich ist

minprintf.c

Formatierte Eingabe - scanf

- Analog zu printf() bietet scanf() Konvertierungen für Eingabe
  - int scanf(char * format, ...);
  - Format-Argument gibt zu lesende Datentypen an
  - Jedes weitere Argument muss Zeiger auf Speicherbereich (Variable) sein
- scanf() stoppt wenn:
  - Format-String abgearbeitet ist
  - Eingabe nicht zu Steueranweisungen passt
  - Gibt Zahl der erfolgreich gelesenen Eingabewerte zurück
  - EOF (-1) bei Dateiende
    - Rückgabewert 0 sagt dagegen, dass kein passender Text gelesen wurde
    - Nächster Aufruf von scanf() setzt nach dem letzten verarbeiteten Zeichen fort
- sscanf() liest aus String anstatt von der Standardeingabe
  - int sscanf(char * string, char * format, ...);

scanf: Format-String

- Format-String kann enthalten:
  - Leerzeichen, Tabulatorzeichen: werden ignoriert
  - Reguläre Zeichen (kein %), die auf das nächste Nicht-Leerzeichen im Eingabestrom passen müssen
  - Konvertierungsanweisungen: Leitet Konvertierung in das nächste Eingabefeld
- Konvertierungsanweisung bestehend aus:
  - ’%’,
  - einem optionalen Unterdrückungszeichen ’*’,
  - einer optionalen maximalen Feldweite,
  - einem optionalen ’h’, ’l’, oder ’L’ für die Weite der Zielvariablen
  - einem Konvertierungszeichen
- Eingabetext besteht aus Zeichenkette von Nicht-Leerzeichen
  - ’*’ zeigt an, dass ein Eingabefeld (Variable) ausgelassen werden soll
  - scanf liest über Zeilenenden hinweg
    - Leerzeichen sind blank, tab, newline, carriage return, vertical tab, formfeed

scanf-Konvertierungen

Zeichen	Argumenttyp, interprtiert als
d	int *; Dezimalzahl
i	int *; Integer –Oktal (mit führender Null) oder Hexadezimal (führende 0x oder 0X)
o	int *; vorzeichenlose Oktalzahl (mit oder ohne führende Null)
x	int *; vorzeichenlose Hexadzimalzahl (mit oder ohne führendes 0x oder 0X), mit abcdef oder ABCDEF für 10,..,15
u	int *; vorzeichenlose Dezimalzahl
c	char *; die nächsten Zeichen werden eingelesen (default: 1). Leerzeichen werden nicht überlesen. Mit %1s liest man das nächste Nicht-Leerzeichen.
s	char *; Zeichenkette ohne ’“’-Quotes; Null-Zeichen ’\0’ wird angehängt; Argument muss Adresse eines Feldes genügender Länge sein.
e, f, g	float *; Gleitkommazahl mit optionalem Vorzeichen, optionalem Dezimalpunkt und optionalem Exponent
%	einzelnes ’%’-Zeichen; es erfolgt keine Zuweisung

adder.c

scanf: Verwendung

- Scanf ignoriert Leerzeichen und Tabulatoren im Format-String
  → will man variable Eingabe lesen, besser: fgets() + sscanf()
  – scanf() kann mit anderen E/A-Funktionen gemischt werden
- Beispiel: alternative Datumseingabe
  – 25 Dec 1988 →

```
int day, year;
char monthname[20];
scanf(”%d %s %d”, &day, monthname, & year);
```

  – 12/25/88 →

```
int day, month, year;
scanf(”%d/%d/%d”, &month, &day, &year );
```

- Fehlerquelle:
  – Argumente _MÜSSEN_ Zeiger (Adressen) sein
  – Fehler: scanf(”%d”, n); anstelle scanf(”%d”, &n);
  – Kann vom Compiler nicht entdeckt werden!!!

getdate.c

Dateioperationen

Datei ist ein abstrakter Datentyp

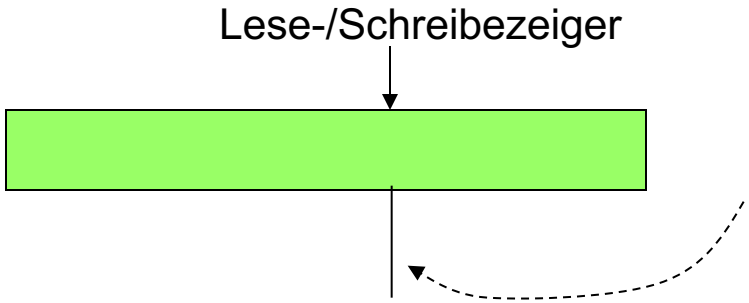
- Werte + Menge von Operationen
- Jedes Betriebssystem unterstützt eine Reihe grundlegender Dateioperationen

- Anlegen einer Datei
  - Platz im Dateisystem allozieren; Eintrag im Verzeichnis erzeugen
- Öffnen einer Datei
  - Name und Art des Zugriffs muss angegeben werden
  - System erzeugt FILE-Pointer oder Dateideskriptor
- Schreiben einer Datei
  - Zu schreibende Information muss angegeben werden
  - System pflegt einen Lese-/Schreibezeiger für die Datei
- Lesen einer Datei
  - Puffer für die zu lesenden Daten muss angegeben werden
  - System pflegt einen Lese-/Schreibezeiger für die Datei
- Schließen einer Datei
  - Alle vom System verwalteten Puffer werden auf die Platte geschrieben

Dateioperationen (contd.)

- Positionieren innerhalb einer Datei (file seek)
  - Datei muss geöffnet sein
  - Lese-/Schreibezeiger wird auf einen Wert gesetzt
- Löschen einer Datei
  - Speicherplatz freigeben, Verzeichniseintrag löschen
  - Unix unterstützt unlink()-Operation; Datei wird freigegeben wenn der letzte Link verschwindet
- Abschneiden (Verkürzen) einer Datei
  - Dateiattribute bleiben unverändert
  - Länge wird auf 0 gesetzt (oder einen angegebenen Wert)

Zusätzlich:
- Append
- Rename
- Set attributes
- ...



UNIX unterstützt „truncation-on-close“
Flagge um Dateien zu verkürzen

Zugriff auf eine Datei erlangen

- open()-Systemaufruf
  - Nimmt Dateinamen als Argument, durchsucht Verzeichnis, prüft Rechte
  - Kopiert Verzeichnis-Eintrag in Tabelle geöffneter Dateien
  - Gibt einen Zeiger in die Tabelle zurück (kleine integer-Zahl)
- close()-Systemaufruf
  - Schreibt gepufferte Daten auf Speichergerät
  - Löscht Eintrag aus der Liste offener Dateien
  - Gibt Systemressourcen frei
- Betrieb in einer multiuser-Umgebung
  - Tabelle offener Dateien existier pro-Prozess und systemweit
  - System pflegt Referenzzähler für geöffnete Dateien

Dateizugriff in C (gepuffert)

- Funktionen deklariert in <stdio.h>
- Vordefinierter Typ: FILE
  - struct-Typ, üblicherweise nur als FILE* verwendet
  - Datenfolge (stream) – Implementierung verborgen
  - 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
- Übertragung des Puffers an Umgebung: int fflush(FILE*);
  - 0 bei Erfolg, EOF bei Fehler

Lesen und Schreiben von Dateien

- Datei öffnen:
  - FILE * fopen(char *name, char *mode);
  - Mode gibt an ob Lesen (”r”) oder Schreiben (”w”, ”a”) erlaubt sind
  - ”w” löscht alten Inhalt; Lese-/Schreibezeiger wird auf 0 gesetzt
  - Binärdateien (”b”) werden anders als Textdateien behandelt
- Zeichen lesen oder schreiben:
  - int getc(FILE * fp);
  - int putc(int c, FILE * fp);
- Standardfilepointer:
  - stdin, stdout, stderr werden vom Laufzeitsystem initialisiert
  - # define getchar() getc(stdin)
  - # define putchar(c) putc((c), stdout)
- Formatierte Ein- und Ausgabe
  - int fscanf(FILE * fp, char * format, ...);
  - int fprintf(FILE * fp, char * format, ...);

mycat.c

Unformatierte Ein- und 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
- Positionierung in Datei:
  - ftell, fseek, fgetpos, fsetpos

Zeilenweise Ein- und Ausgabe

- char * fgets(char *line, int maxline, FILE *fp);
  - Liest Zeile inklusive Zeilenende (newline)
  - Höchstens maxline-1 Zeichen
  - Hängt NULL-Byte an gelesene Zeile an
  - Gibt Zeiger auf die Zeile oder NULL (bei Fehler / EOF) zurück
- int fputs(char *line, FILE *fp);
  - Schreibt Zeichenkette (mit oder ohne newline)
  - Gibt EOF bei Fehler zurück; 0 sonst
- Gefährliche Variante:
  - char * gets(char * buf);
    - Liest bis Zeilenende (verschluckt newline) – von stdin
    - Gefahr eines Puffer-Überlaufs!!!
  - int puts(char * buf);
    - Schreibt Zeichenkette, hängt newline-Zeichen an

compare.c

Fehlerbehandlung bei FILE-I/O

- Funktionen liefern int-Werte, um Fehler anzuzeigen
- Ein/Ausgabe auf FILE*: Test, auf Fehler oder Dateiende mit separaten Funktionen:
  - int feof(FILE*); - Rückgabewert ≠ 0 bei Dateiende
  - int ferror(FILE*); - Rückgabewert ≠ 0 wenn letzte Operation fehlerhaft
- Terminierung mit exit()
  - Deklariert in <stdlib.h>
  - Innerhalb main: exit(*expr*); äquivalent zu return *expr*;
  - Aber: exit() bewirkt Terminierung auch in verschachtelten Funktionen
- 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, ...

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 (kann explizit durch -lc angegeben werden)
  - 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: msvcrt.dll, msvcrt4.dll, msvcr7.dll, msvcr71.dll, msvcr8.dll, msvcrt.lib (import library)
- Auswahl der Bibliothek automatisch und durch Compilerflags.
  - z.B. /MD, /MDd

Funktionen zur Speicherverwaltung

- Funktionen in <stdlib.h>
- Allozierung: 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)
  - Beispiel:

```
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 von malloc() 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
- Zeichenkettenliterale enthalten automatisch Null-Terminierung:
  - char s[] = “Hallo”; /* sizeof(s) == 6 */
- Zeichenkettenliterale sind nicht änderbar (const), können aber in char* konvertiert werden
  - char *t = “Hello”;

Zeichenketten (contd.)

- 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)

stringops.c

Zeichenketten (contd.)

- 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, ...

Verschiedene Funktionen

- Testen auf Typ eines Zeichens
  - # include <ctype.h>
    - isalpha(c), isupper(c), islower(c)
    - isdigit(c), isalnum(c), isspace(c),
    - toupper(c), tolower(c)
- Zeichen „zurückgeben“
  - int ungetc(int c, FILE * fp);
  - Gibt Zeichen c „an FILE zurück“ – maximal 1 Zeichen
  - Kann von scnaf(), getc(), etc. erneut konsumiert werden
- Kommandoausführung
  - int system(char * s);
  - Führt ein Betriebssystemkommando aus – mit Hilfe der shell
  - Gibt exit-Code des Kindprozesses zurück
    - system(”/bin/date”);

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
- ...

Zusammenfassung

- Standard Ein- und Ausgabe – stdio
- Formatierte Ausgabe – printf
- Argumentlisten variabler Länge – varargs
  - In C ist Aufrufer für „Aufräumen“ des Stacks verantwortlich
- Formatierte Eingabe – scanf
- Dateizugriff
  - FILE* implementiert Abstraktionen oberhalb von Betriebssystem-I/O-Rufen
- Fehlerbehandlung – stderr und exit
- Zeilenweise Ein- und Ausgabe
- Weitere Funktionen aus der libc