Betriebssysteme

Labor 8

Die Programmiersprache C

Grundlegende Datentypen

Datentyp	Speichergröße	Grenzen des Wertebereichs
`char`	1 Byte	[-128, 127] oder [0, 255]
`int`	2 oder 4 Bytes	[-32.768, 32.767] oder [-2.147.483.648, 2.147.483.647]
`float`	4 Bytes	[1,2E-38, 3,4E+38]
`double`	8 Bytes	[2,3E-308, 1,7E+308]

- auf die Datentypen `char` und `int` kann der Qualifizierer `unsigned` angewendet werden
- auf den Datentyp `int` können die Qualifizierer `short` und `long` angewendet werden

Grundlegende Datentypen

Datentyp	Speichergröße	Grenzen des Wertebereichs
`unsigned char`	1 Byte	[0, 255]
`unsigned int`	2 oder 4 Bytes	[0, 65.535] oder [0, 4.294.967.295]
`short (int)`	2 Bytes	[-32.768, 32.767]
`unsigned short`	2 Bytes	[0, 65.535]
`long (int)`	4 Bytes	[-2.147.483.648, 2.147.483.647]
`unsigned long`	4 Bytes	[0, 4.294.967.295]

Reservierte Wörter (Schlüsselwörter)

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	unsigned
union	void	volatile	while

Konstanten

- können auf zwei Arten definiert werden:
  - durch die Verwendung von `#define`-Vorverarbeitungsdirektive:

```
#define TEN 10
#define NEWLINE '\n'
```

  - durch die Verwendung von `const`-Präfix:

```
const int TEN = 10;
const char NEWLINE = '\n';
```

Variablen

- Syntax:

```
variablen_typ   name_variable;
```

  - `variablen_typ` kann `char, int, short, long etc.` sein:
  - **`name_variable`** kann aus Buchstaben, Ziffern und dem Zeichen „ _” `(underscore)` bestehen
  - das erste Zeichen muss ein Buchstabe sein
  - reservierte Wörter können nicht als Variablennamen verwendet werden
  - `C` unterscheidet zwischen Klein- und Großbuchstaben `(case-sensitive)`
- Beispiele:

```
int n;     int n = 10;
char c;    char c = 'a';
```

Operatoren

- gibt die Operationen an, die mit Variablen und Konstanten durchgeführt werden sollen
- Arten von Operatoren:
  - Arithmetische Operatoren: `+ - * / % ++ --`
  - Vergleichsoperatoren: `== != > < >= <=`
  - Logische Operatoren: `&& || !`
  - Bitweise logische Operatoren: `& | ^ ~ << >>`
  - Zuweisungsoperatoren: `= += -= *= /= %= <<= >>= &= ^= |=`
  - andere Operatoren: `sizeof() & * ?:`

Abgeleitete Datentypen

- **Felder (Arrays)**

```
array_typ name_array[array_größe];
```

  - Beispiele:

```
int list[5];
int list[5] = {10, 20, 30, 40, 50};
double values[] = {100.0, 2.0, 300.0, 40.0, 50.0};
```

- **Zeichenketten (Strings)**

```
char msg[] = "Hello";
char msg[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

Abgeleitete Datentypen

- **Zeiger (Pointer)**
  - **`zeiger`** = eine Variable, die die Adresse einer anderen Variablen enthält
  - Beispiele:

```
int *p;     // Zeiger auf eine int-Variable
char *c;    // Zeiger auf eine char-Variable
float *f;   // Zeiger auf eine float-Variable
double *d;  // Zeiger auf eine double-Variable
```

  - Abrufen der Adresse, an der die Variable `x` gespeichert ist: `&x`
  - Erhalt des Wertes der Variablen `x`: `* p` (wenn `p` ein Zeiger auf die Variable `x` ist)

Abgeleitete Datentypen

- **Strukturen (Datenstrukturen)**
  - Die Deklaration der Strukturen:

```
struct Books
{
    int id;
    char author[50];
    char title[100];
}
```

  - Deklaration und Verwendung einer Struktur:

```
int main(int argc, char** argv)
{
    struct Books book1;
    ...
    book1.id = 1000;
    strcpy(book1.author, "B.W. Kernighan,
    D.M. Ritchie");
    strcpy(book1.title, "The C Programming
    Language");
    ...
    return 0;
}
```

Funktionen

- Syntax:

```
zurückgegebener_typ name_function(typ_param param1, typ_param param2, ...);
```

  - wo:
    - `zurückgegebener_typ` kann ein `grundlegender Datentyp/abgeleiteter Datentyp` oder `void` sein
    - **`name_function`** kann aus Buchstaben, Ziffern und dem Zeichen „ _ ” `(underscore)` bestehen
    - das erste Zeichen muss ein Buchstabe sein
    - reservierte Wörter können nicht als Funktionsnamen verwendet werden
    - `typ_param` **`param1,`** `typ_param` **`param2, ...`** ist die Liste der formalen oder ungültigen Parameter (wenn die Funktion keine Parameter hat)

Funktionen

- Beispiele:

```
void display_array(int** array)
float arithmetisches_mittel(int a, int b)
int** reas_array(FILE* file)
```

- `main()`- Funktion:
  - ist der Haupteinstiegspunkt in das Programm `(the program main entry point)`
  - Der empfohlene Prototyp ist:

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

weil es den Zugriff auf die vom Benutzer auf der Befehlszeile bereitgestellten Argumente ermöglicht.

Eingangs-/Ausgangsfunktionen

```
int getchar(void)
int putchar(void)
char *gets(char *s)
int puts(const char *s)
int scanf(const char *format, ...)
int printf(const char *format, ...)
```

Funktionen zum Arbeiten mit Dateien

- für Textdateien:

```
FILE *fopen(const char *filename, const char *mode)
int fgetc(FILE *fp)
int fgets(char *buf, int n, FILE *fp)
int fputc(int c, FILE *fp)
int fputs(const char *s, FILE *fp)
int fclose(FILE *fp)
```

Funktionen zum Arbeiten mit Dateien

- für Binärdateien:

```
size_t fread(void *buf, size_t bsize, size_t nbyte, FILE *fp)
size_t fwrite(const void *buf, size_t bsize, size_t nbyte, FILE *fp)
```

oder

```
int open(const char *path, int oflag, ... )
ssize_t read(int fd, void *buf, size_t nbyte)
ssize_t write(int fd, const void *buf, size_t nbyte)
int close(int fd)
```

C-Programme unter Unix

Das erste C-Programm in Unix

- UNIX-Texteditoren: `vi, nano, joe`
- Beispiel: `hello.c`

```
#include <stdio.h>

// the program main entry point
int main(int argc, char** argv)
{
     printf("Hello world !\n");
     return 0;
}
```

- Kompilieren: `gcc -Wall -g -o hello hello.c`
- Ausführung: `./hello`

Beispiele

- Abrufen von Anzahl und Liste der Argumente, die auf der Befehlszeile bereitgestellt werden:

lab7_1.c

```
#include <stdio.h>

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

	if (argc == 1)
	{
		printf("Error: Insufficient number of arguments.\n");
		printf("Usage: lab7_1 arg_1 arg_2 ...\n");
		return 1;
	}

	printf("Number of arguments (argc): %d\n", argc);

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

	return 0;
}
```

Beispiele

- Abrufen und Anzeigen von Umgebungsvariablen:

lab7_2.c

```
#include <stdio.h>

int main(int argc, char *argv[], char *env[])
{
	int i = 0;

	printf("Environment variables:\n");
	while (env[i])
	{
		printf("env[%d]: %s\n", i, env[i]);
		i++;
	}

	return 0;
}
```

Beispiele

- Verwendung von eindimensionalen Arrays:

lab7_3.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
	long t[10], *p;

	int i;
	for (i=0; i<10; t[i++]=i);

	p = t;

	for (i=0; i<10; i++)
		printf("%d %d %d %d\n", t[i], p[i], *(p+1), *(t+1));

	return 0;
}
```

Beispiele

- Lesen einer Ganzzahl von der Tastatur:

lab7_4.c

```
#include <stdio.h>

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

      printf("Geben Sie den Wert von n ein: ");
      scanf("%d", n);

      printf("Wert von n ist: %d\n",n);
      return 0;
}
```

Beispiele

- Lesen den Inhalt einer Textdatei:

```
wget http://www.cs.ubbcluj.ro/~dbota/SO/lab2/exemple/lab2_6.c
```

- Lesen den Inhalt eines Arrays aus einer Textdatei:

```
wget http://www.cs.ubbcluj.ro/~dbota/SO/lab2/exemple/lab2_7.c
```

Kompilierungsfehler / Warnungen

- Syntaxfehler
- Überspringen einer Header-Datei
- eine undefinierte Variable verwenden
- Verwenden von zwei gleichnamigen Variablen
- Verwendung einer nicht deklarierten Funktion
- Aufrufen einer Funktion ohne Beachtung ihres Prototyps (falsche Anzahl von Argumenten, Umkehrung der Argumentenreihenfolge usw.)

-> Fehlererkennung in der Speicherverwaltung

Erkennung von Speicherverwaltungsfehlern mit dem Dienstprogramm `valgrind`:

```
valgrind ./myprog
```

C-Programme, die mit Unix-Dateien arbeiten

Theoretische Aspekte

- um auf eine Datei einwirken zu können, benötigt man zunächst eine Methode, um die Datei eindeutig zu identifizieren. Bei den besprochenen Funktionen wird die Datei durch einen sogenannten `Dateideskriptor (`*`file descriptor`*`)` identifiziert. Dies ist eine Ganzzahl, die der Datei beim Öffnen zugeordnet wird.

Öffnen von Dateien

- das Öffnen einer Datei ist der Vorgang, durch den die Datei vorbereitet wird, damit sie weiter verarbeitet werden kann.
- Dieser Vorgang wird über die `open`-Funktion ausgeführt:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int oflag, [, mode_t mode]);
```

- Die Funktion gibt im Fehlerfall `-1` zurück. Andernfalls wird der Dateideskriptor zurückgegeben, der der geöffneten Datei zugeordnet ist.

Öffnen von Dateien

- **Parameter:**
  - **`pathname`** - enthält den Dateinamen
  - **`oflag`** - Optionen zum Öffnen von Dateien. Es ist eigentlich eine Kette von Bits, in der jedes Bit oder jede Gruppe von Bits eine bestimmte Bedeutung hat. Für jede dieser Bedeutungen ist in der `Header-Datei C` **`fcntl.h`** eine Konstante definiert. Konstanten können mit dem '`|`' kombiniert werden (oder durch bitweise logische Operatoren) von `C`, um mehrere Bits im **`oflag`**-Parameter zu setzen (so kann man mehrere Optionen wählen).

Öffnen von Dateien

- **Parameter:**
  - **`oflag`**: Einige dieser Konstanten:
    - **`O_RDONLY:`** nur zum Lesen öffnen
    - **`O_WRONLY`**: nur zum Schreiben öffnen
    - **`O_RDWR`**: zum Lesen und Schreiben öffnen
    - **`O_APPEND:`** Datei öffnen um am Ende zu schreiben
    - **`O_CREAT`**: Erstellen der Datei, falls sie noch nicht existiert. Falls die Datei existiert, ist `O_CREAT` ohne Wirkung. Bei Verwendung mit dieser Option muss die `open`-Funktion auch den `mode`-Parameter erhalten
    - **`O_EXCL:`** "exklusive" Dateierstellung: Wenn `O_CREAT` verwendet wurde und die Datei bereits existiert, gibt die `open`-Funktion einen Fehler zurück
    - **`O_TRUNC:`** wenn die Datei existiert, wird ihr Inhalt gelöscht

Öffnen von Dateien

- **Parameter:**
  - **`mode:`** wird nur verwendet, wenn die Datei erstellt wird, und gibt die der Datei zugeordneten Zugriffsrechte an. Diese werden durch Kombinieren von Konstanten mit dem Operator oder ('|') wie in der vorherigen Option erhalten. Die Konstanten sind:
    - **`S_IRUSR:`** Leserecht für den Eigentümer der Datei (user)
    - **`S_IWUSR:`** Schreibrecht für den Eigentümer der Datei (user)
    - **`S_IXUSR:`** Ausführungsrecht für den Eigentümer der Datei (user)
    - **`S_IRGRP:`** Leserecht für die Gruppe der Dateibesitzer
    - **`S_IWGRP:`** Schreibrecht für die Gruppe der Dateibesitzer
    - **`S_IXGRP:`** Ausführungsrecht für die Gruppe der Dateibesitzer
    - **`S_IROTH:`** Leserecht für andere Benutzer
    - **`S_IWOTH:`** Schreibrecht für andere Benutzer
    - **`S_IXOTH:`** Ausführungsrecht für andere Benutzer

Erstellen von Dateien

- um Dateien zu erstellen, kann auch die `creat`-Funktion verwendet werden:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

creat (const char *pathname, mode_t mode)
```

- entspricht der Angabe von Optionen `O_WRONLY | O_CREAT | O_TRUNC` auf die `open`-Funktion.

Schließen von Dateien

- nach der Verwendung der Datei muss diese mit der `close`-Funktion geschlossen werden:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int close (int filedes)
```

- dabei ist `filedes` der beim Öffnen erhaltene Dateideskriptor.

Lesen von Daten aus Dateien

- das Lesen von Daten aus einer geöffneten Datei erfolgt mit der `read`-Funktion:

```
#include <unistd.h>

ssize_t read(int fd, void *buff, size_t nbytes)
```

- die Funktion liest eine genaue Anzahl von `nbytes` Bytes von der aktuellen Position in der Datei, deren Deskriptor `fd` ist, und platziert sie in dem Speicherbereich, der durch den `buff`-Zeiger angegeben wird.
- Es ist möglich, dass weniger als `nbytes` Bytes gleichzeitig in der Datei gelesen werden können (zB wenn das Ende der Datei erreicht ist), so wird die `read`-Funktion nur so viele Bytes im Puffer (`buff`) setzen, wie sie es lesen kann. In jedem Fall gibt die Funktion die Anzahl der aus der Datei gelesenen Bytes zurück, sodass dies leicht bemerkt werden kann.
- ist das Dateiende genau erreicht, liefert die Funktion `0`, im Fehlerfall `-1`.

Schreiben von Daten in Dateien

- das Schreiben von Daten erfolgt mit der `write`-Funktion:

```
#include <unistd.h>

ssize_t write(int fd, void *buff, size_t nbytes)
```

- die Funktion schreibt die ersten `nBytes` Bytes des durch `buff` angegebenen Puffers in die Datei.
- gibt im Fehlerfall `-1` zurück

Dateideskriptor positionieren

- Schreib- und Leseoperationen in und aus der Datei werden an einer bestimmten Position in der Datei ausgeführt, die als aktuelle Position betrachtet wird. Beispielsweise aktualisiert jede Leseoperation den Anzeiger der aktuellen Position, indem er um die Anzahl der gelesenen Bytes erhöht wird. Die aktuelle Positionsanzeige kann auch explizit mit der Funktion `lseek` gesetzt werden:

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int fd, off_t offset, int pos)
```

Dateideskriptor positionieren

- die Funktion positioniert den Zeiger wie folgt auf die `offset`-Verschiebung in der Datei:
  - wenn Parameter `pos` den Wert **`SEEK_SET`** hat, erfolgt die Positionierung relativ zum Anfang der Datei (Schreib/Lese-Deskriptor vom Dateianfang um `offset` Bytes versetzen)
  - wenn Parameter `pos` den Wert **`SEEK_CUR`** hat, erfolgt die Positionierung relativ zur aktuellen Position (Schreib/Lese-Deskriptor von der aktuellen Position um `offset` Bytes versetzen)
  - wenn Parameter `pos` den Wert **`SEEK_END`** hat, erfolgt die Positionierung relativ zum Ende der Datei (Schreib/Lese-Deskriptor vom Dateiende um `offset` Bytes versetzen)
- der `offset`-Parameter kann auch negative Werte annehmen und stellt die Verschiebung dar, berechnet in Bytes
- im Fehlerfall gibt die Funktion `-1` zurück

Bibliotheksfunktionen

- **Öffnen einer Datei:**

```
#include <stdio.h>

FILE *fopen(const char *filename, const char *mode)
```

  - Die Funktion öffnet die durch `filename` angegebene Datei, erstellt eine `FILE`-Struktur, die Dateiinformationen enthält, und gibt einen Zeiger darauf zurück
  - dieser Zeiger ist das Element, das weiterhin für den Zugriff auf die Datei verwendet werden kann

Bibliotheksfunktionen

- **Öffnen einer Datei:**
  - Der `mode`-Parameter ist eine Zeichenfolge, die angibt, wie die Datei geöffnet wird.
    - **`"r"`** bedeutet Öffnung zum Lesen
    - **`"w"`** bedeutet Öffnung zum Schreiben
    - **`"a"`** bedeutet Öffnen zum Hinzufügen am Ende der Datei
  - der Dateityp kann ebenfalls angegeben werden:
    - **`"t"`** für Textdatei
    - **`"b"`** für Binärdatei
  - Optionen können kombiniert werden, zum Beispiel in Form von **`"r+t"`**.

Bibliotheksfunktionen

- **Schließen einer Datei:**

```
#include <stdio.h>

fclose(FILE *stream)
```

  - wobei `stream` der Zeiger auf die `FILE`-Struktur ist, die beim Öffnen der Datei erhalten wird

Dateioperationen

- `int fprintf(FILE *stream, const char *format, ...);`
  - Schreiben in Datei (mit Formatierung)
  - Der String (Zeichenkette), der das Format angibt, ähnelt dem in der `printf`-Anweisung.

- `int fscanf(FILE *stream, const char *format, ...);`
  - Lesen aus Datei
  - ähnlich der `scanf`-Funktion.

Dateioperationen

- `size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);`
  - liest aus der durch den `stream` angegebenen Datei eine Anzahl von `nmemb`-Elementen, jedes mit der Größe `size`, und legt sie in dem durch `ptr` angegebenen Speicherbereich ab.

- `size_t fwrite( void *ptr, size_t size, size_t nmemb, FILE *stream);`
  - schreibe in die durch den `stream` angegebene Datei eine Anzahl von `nmemb`-Elementen, jedes mit der Größe `size`, die er aus dem durch `ptr` angegebenen Speicherbereich nimmt.

Ressourcen

- Priorität der Operatoren:
  `https://en.cppreference.com/w/c/language/operator_precedence`
- C-Programmierung:
  `https://www.tutorialspoint.com/cprogramming/index.htm`
- Valgrind Handbuch:
  `http://valgrind.org/docs/manual/quick-start.html`
- Standard C I/O:
  `https://en.cppreference.com/w/cpp/io/c`
- Programming in C: A Tutorial by Brian W. Kernighan
  `http://www.lysator.liu.se/c/bwk-tutor.html`