1 Grundlagen

1.1 Eigenschaften

fest definierten und maschinell überprüfbaren Syntax (Struktur); eindeutige Semantik
kein Interpretationsspielraum

1.2 Compiler

Übersetzen eines Programms in Maschinensprache:

1. Scanner/Parser: feststellen der syntaktischen Korrektheit
2. Codegenerator: Erzeugen des Maschinencodes ggf. Optimierung

1.3 Interpreter

Alternative zum Compiler

- QT wird bei jeder Programmausführung gelesen
- Interpretiert Computerprogramm (Pro Befehl eine Befehlsfolge) schrittweise

1.4 Linker

- Teil der Compilers
- Baut Programm aus verschiedenen Teilen zusammen
- Ergänzt die vom Compiler erzeugen Maschinencodes um vordefinierte Bibliotheken
- ermöglicht getrennte und verteilte Entwicklung

1.5 Debugger

Werkzeug zur Analyse und Behebung von Fehlern

2 C

2.1 Geschichte

- 1971-1973 von Dennis Ritchie in den Bell Laboratories fÃr die Programmierung des damals neuen UNIXBetriebssystems entwickelt.
- Basis: Programmiersprache B von Ken Thompson in den Jahren 1969/70 geschrieben.
- B wiederum geht auf die von Martin Richards Mitte der 1960er-Jahre entwickelte Programmiersprache BCPL zurÃck.
- 1973 erster Unix-Kernel (Betriebbsystem) in C geschrieben 1978 veröffentlichen Brian W. Kernighan und Dennis Ritchie die erste Auflage von "The C Programming Language" deutsch: Programmieren in C).Die darin beschriebene Fassung von C, die nach den Buchautoren "K&R C" genannt wird, wird die erste informelle C Referenz.
- C verbreitete sich rasch und wird laufend weiterentwickelt:
  - Normierung durch ANSI 1989 (ANSI X3.159-1989 Programming Language C)
  - 1990 entsprechende ISO Norm C90 (mit kleinen Änderungen als C90
  - 1995 ISO Norm (C95) und 1999 ISO/IEC 9899 (C99).

3 Programmiersprache C

- Haupteinsatzgebiete:
  - Systemprogrammierung
  - Betriebssysteme (Hardwarenahe Programmierung zB Treibe)
  - Eingebettete Systeme
  - hochperformante Programmteile oder Applikationen
- Basis/Grundlage vieler Andere Programmiersprachen

Programmkomponenten

- Grundstruktur eines C Programms
- Variabeln
- Konstanten
- Datentypen

3.1 Grundstruktur eines C-Programms

- Anweisungen in Einzelschritten (max. eine Anweisung pro Zeile)
- Anweisung durch Semikolon abgeschlossen
- ZusammengehÃ¶rende Anweisungen werden durch {...} zusammengefasst
- Textkonstanten werden durch "..." begrenzt
- Kommentare werden durch /*...*/ begrenzt bzw. //... bis Zeilenende
- mit `getchar();` wird auf Tastendruck gewartet, WICHTIG: Eingabepuffer mit `fflush(stdin);` leeren

3.1.1 Main Funktion

- Bestandteil von jedem C Programm
- Start bzw. Endpunkt vom Hauptprogramm
- mit "{...} " abgegrenzt
- kein komplettes Programm

3.1.2 Präprozessoranweisungen

- werden mit # eingeleitet

3.1.3 Variablen

- Identifikatoren
  - nur Buchstaben, Ziffern und _
  - erstes Zeichen darf keine Ziffer
  - Groß- und Kleinschreibung wird unterschieden
  - C-Keywords dÃ¼rfen nicht verwendet werden
  - SelbsterklÃ¤rend
  - Variablen im CamelCase oder mit underscore
  - Temporaries dem Typen angepasst
  - Englische Namen
- Typisierung
  - Typ gibt den Speicher an
  - Typen sind vordefiniert
  - Typen können in Programm definiert werden

- Deklaration: typ variabelname;
  - fast überall definierbar
  - legt Variabel an und macht sie Bekannt
  - nach Deklaration ist Var in Block ({...}) bekannt
- sind immer zu initialisieren

3.1.4 Konstanten

- Literalkonstanten
  - dezimale Zahl (1,2,3)
  - hexadezimale Zahl, PrÃfix 0x (0x1,0x2,0x3)
  - oktale Zahl, PrÃfix 0 (010)
  - Gleitkomma (10.5,3.14)
- Konstanten
  - Variablen auch als Konstanten
  - mit `const` initialisiert
  - nicht zu Überschreiben, werden im Speicher abgelegt
  - auch als Präprozessor definierbar `#define KONSTANTENAME Ausdruck` wird dann Textuell ersetzt (keine Speicherung im Programm)

3.1.5 Datentypen

Datentyp	Keyword	Größe in Bytes	Wertebereich
Zeichen	char	1	-128 bis 127
ganze Zahl(kurz)	short (short int)	2	-32768 bis 32767
ganze Zahl	int	4(meist)	-2147483648 bis 2147483647
Ganze Zahl lang	long (long int)	4	-2147483648 bis 2147483647
ohne Vorzeichen	unsinged char	1	0 bis 255
ohne Vorzeichen	unsinged short	2	0 bis 65535
ohne Vorzeichen	unsinged int	4(Meist)	0 bis 4294967295
ohne Vorzeichen	unsinged long	4	0 bis 4294967295
einfache Gleitkomma	float	4	Genauigkeit 7 Dezimale
doppelte Gleitkomma	double	8	Genauigkeit 19 Dezimale

Typ-KompatibilitÃ˜t

- Standardtypen sind zueinander Kompatibel, können mit Operationen verknüpft werden (kann Nebeneffekte haben)
- char hat einen "Doppelrolle" sowohl Zeichen als auch Zahl
- Fehlerquellen:
  - Abschneiden des Nachkommateils
  - Abscheiden der hÃ¶herwertigen Bits oder Rundungsfehler
  - Verlust des Vorzeichenes, Anderung des Wertes

3.1.6 Ein/Ausgabe

Ausgabe: printf(" $<$ %[*Flag*][*Breite*][.PrÃzision][PrÃfix]*Typ* $>$ ",$<$ *auszugebendeDaten* $>$);

- Steuerzeichen:
  - \a: BEL - akustisches Warnsignal
  - \b: BS Backspace -Cursor um einen Position nach links
  - \f: FF formfeed - Seitenvorschub
  - \n: NL Newline- der Cursor geht zur nÃchsten Zeile
  - \r: CR Carriage return - der Cursor springt zum Anfang der Zeile
  - \t: HT Horiziontal tab - Zeilenvorschub zur nÃchte horizontalen Tabulatorposition
  - \v: VT vertical tab - Zeilenvorschub zur nÃchte vertikalen Tabulatorposition
  - \": " wird ausgegeben
  - \': ' wird ausgegeben
  - \?: ? wird ausgegeben
  - \\: \wird ausgegeben
  - \0: Endmakierung eines Strings
  - \nnn: Ausgabe eines Oktalwerts
  - \xhh: Augabe eines Hexadezimalwerts
- Typ:
  - %d %i Dezimalzahl mit Vorzeichen
  - %o Oktalzahl
  - %x %X Hexadezimalzahl (klein/groß)
  - %u Dezimalzahl ohne Vorzeichen
  - %c Buchstabe(Charakter)
  - %s Zeichenkette(String)
  - %f Gleitkommazahl
  - %e %E Gleitkommazahl (Exponentialdarstellung)
  - %g %G Double (Exponentialdarstellung)
  - %p Pointer
  - %n Anzahl auszugebender Zeichen
  - %a wie %f (ab C99)
- Flagangabe, optionales "-"-Zeichen legt linksbündige Ausgabe fest "+"-Zeichen gibt Plus bei positiven zahlen aus
- Breite, die Zahl Breite legt die minimales Breites des Ausgabefeldes fest
- Präzision, legt die anzahl an Nachkommastellen fest

Eingabe: scanf(" $<$ %[*Flag*][*Breite*][.PrÃzision][PrÃfix]*Typ* $>$ ",$<$ *Adresse* $>$);

3.1.7 Operatoren

- Verknüpfung/Manipulation von Daten
- Verwendbar bei Variablen Konstante und Ausdrücken
- Allgemeine Operatoren:
  - = Zuweisungsoperator
  - * Multiplikation, / Division, + Addition, - Subtraktion
  - & Adressoperator
  - % Modulo (Restbildung)
  - == Gleichheit
  - != Ungleichheit
  - <, <=, >=, > Vergleich auf kleiner/größer
  - && logisches und, | logisches oder, ! Negation
- Bitweise Operatoren:
  - & und, $\wedge$ oder, $\sim$ Negation
  - ? Bedingter Ausdruck
    $< boolscher Ausdruck >$?TrueStatement:Falsestatement
  - Bitweise Operatoren verknÃpfen die einzelnen Bits
- Schiebe Operatoren(shifts):
  - << Linksshift, >> Rechtsshift
  - Linksshift ziehen immer einen "0" nach, Rechtsshift auf unsinged Größen ebenfalls
  - Rechtsshift auf auf singed Größen ziehen das Vorzeichenbit nach
  - Multiplikation bzw. Division durch 2er Potenzen (bei Laufzeitkritischen Anwendungen)
- einstellige Operatoren:
  - $-$ Negation
  - ++ Inkrement $i++\widehat{=}i = i + 1$
  - $--$ Dekrement $i--\widehat{=}i = i - 1$
  - $+= n$ Addition einer Zahl n $x+ = b\widehat{=}x = x + b$ (analog $*=; /=; \%=$)

3.1.8 Kontrollstrukturen

- if ... else
  - Syntax: if (¡Bedingung¿) {< *TrueStatement* >} else {< *Falsestatement* >}
  - Anwendung: Vergleiche
- switch ... case ...
  - Syntax: switch(Wert) {case Wert^{i}: < *Anweisungsblock* >; break; default:< *Anweisungsblock* > } (case Wert^{i} müssen Konstanten sein)
  - Anwendung: Überschaubare Anzahl diskreter Werte
- while ...
  - Syntax: while(Bedingung) {< *Anweisungsblock* >}
  - Ausführung nur wenn Bedingung wahr
- do ... while ...
  - Syntax: do < *Anweisungsblock* > while (Bedingung)
  - Ausführung min einmal auch wenn Bedingung falsch (While schleife Bevorzugen)
- for ...
  - Syntax:
    for(< *Start* >; < *Abbruchbedingung* >; < Änderung >) < *Anweisungsblock* >
- goto
  - Syntax: goto Sprungmarke; (Sprungmarke Setzen< *Sprungmarke* >:)
  - fÃhrt zu unÃbersichtlichem Code, verboten
- break: Beendet Schleifenbearbeitung, Fortsetzung nach der Schleife
- continue: Beendet Aktuelle SchleifenausfÃhrung, es wird mit der AusfÃhrung der Bedingung fortgesetzt
- return: Ende einer Funktionsberechnung inkl. RÃckgabewert
- exit: beendet ProgrammausfÃhrung

3.1.9 Arrays und Strings

- Deklaration:

  $<typ>$varname[$<ganzzahligerAusdruck>$];

- Globaler Zugriff:

  variabelname, Individueller Zugriff: varaibelname[$<ganzzahligerAusdruck>$]

- erstes Element hat Index -1, letztes Element hat Index

  Anzahl der Vektorelemente -1

- Array = Array nicht mÃ¶glich

- automatische Ermittlung der Elemente: type varname[ ], Beachte:Strings werden immer durch Null-Bit beendet ("\0")

- zur Manipulation: #include$<string.h>$

  - scanf("%s",...) Einlesen Zeichenkette
  - printf("%s",...) Ausgeben von Zeichenketten
  - gets(...) Einlesen von Zeichenketten
  - puts(...) Ausgeben von Zeichenketten
  - ... = getchar() Einlesen von Buchstaben
  - char* strchr(str,Zeichen) suchen nach "Zeichen" in "str"

    Null wenn nicht vorhanden, sonst Adresse des ersten Zeichens
  - strcpy(zielstr, quellstr) strlÃnge muss passen
  - strncpy(char *dest, const char *src, size_t n);
  - strcat(zielstr, quellstr) quell an zielstr anhÃnge strLÃnge von zielstr muss passen
  - strncat(char *dest, const char *src, size_t n)
  - strcmp(str1,str2) vergleicht str1 mit str2 (alphabetisch) 0 wenn gleich; < 0 falls $str1 < str2$; > 0 falls $str1 > str2$
  - strncmp(const char *s1, const char *s2, size_t n)
  - int strcspn(str1,str2) Index des Ersten Zeiches von STR1, welches in Str2 vorkommt
  - int strlen(str) LÃnge von str ohne Stringende
  - char strstr(str1, str2): Festestellen wo str2 in str1

    Ergebnis ist Zeiger auf den Teilstring in str1 falls str2 in str1 vorkommt sonst auf 0

- Mehrdimensionale Array:

  - Deklaration:

    $<typ>$varname[$<Zeile>$][$<Spalte>$]={...}...{...} (Zeile mit "{}" abgetrennt Splaten mit ",")
  - Zugrif auf die Elemente: varname[Zeile][Spalte]

3.1.10 Benutzerdefinierte Datentypen

- typedef Typdefinition Neuer_Typ; (Standarttypen neu Definieren)
- enum . . . ; (Durchnummerieren der Inhalte neue Nummerierung mit "inhalt=nummer")
- structs
  - Zusammenbau eines neuen Typs aus bestehenden Typen
  - typedef struct struct_name typ komponentename typname;
  - Deklaration typname varname;, Komponenten Zugriff varname.komponentename
- union
  - typedef union n_name {typ varname} n; n_i
  - die Komponenten belegen den selber Speicherplatz und haben den Gleichen wert, dieser wird nur anders interpretiert

3.1.11 Präprozessor

- Datenverarbeitung rein Textuell, keine Beachtung des Syntax
- Direktiven
  - #include< *datei* > sucht nach datei im system-verzeichnissen; #include"*datei*" sucht in den Quellverzeichnissen
  - #define TEXT Ersatztext ersetzt Text durch Ersatztext
    #define Makroname(Parameter_i) Ersatztext (Parameter_i wird durchen den I-ten parameter ersetzt)
  - #if AUSDRUCK (Wenn Ausdruck True wird Code übernommen)
  - #ifdef Konstante (Ausdruck wird nur übernommen wenn Konstante Definiert)
  - #ifndef Konstante (Ausdruck wird nur übernommen wenn konstante nicht definiert)
  - #else (Alternative zu vorhergehender #if)
  - #endif (schließt #if)
- symbolische Konstante
  - __LINE__
  - __FILE__
  - __DATE__
  - __TIME__

3.1.12 Pointer

- Ein Pointer zeigt auf eine Stelle im Speicher (Physikalische Speicheradresse) und kann eine Konstante(selten außer Null-Pointer) oder eine Variabel sein und ist typisiert
- Eine Pointervariabel beinhaltet die Speicheradresse und steht selbst im Speicher
- Pointer können auch auf Dateien oder Funktionen zeigen und werden als Referenz bezeichnet
- Verwendung:
  - Hardwarenahe Programierung (Ansprechen von Registern, Ports, Interrupttabellen)
  - Resultat/Parameter Übergabe
  - dynamische Datenstrukturen (Listen/Bäume)
- Deklaration: typ *Pointername; (* drückt aus das es ein Pointer ist)
- typ p;(p ist ein typ) - typ *p;(p ist ein pointer auf einen typ) - typ **p;(p ist ein pointer auf einen pointer auf einen typ)
- Pointer als Ergebnis einer Funktion: typ *f(typ1 p1,...)
- Pointer als Parameter einer Funktion: ergtyo f(typ *p,...)
- Pointer in funktionen als Parameter/Ergebnis erspart das Kopieren großer Datenmengen, ohne Alternative bei Dynamischen Strukturen
- Dereferenzieren:
  - zugriff auf den speicher auf den der Pointer zeigt
  - liefert lesend einen wert von typ auf den der pointer zeigt
  - liefert schreiben eine Speicherstelle vom Typ auf der der Pointer zeigt
  - dieser wert btw die speicherstelle können in operationen oder als parameter oder als ergebnis einer funktion verarbeited werden
  - Dereferenzieren: *pointer wird in einem ausdruck verwendet (muss mit der typ der pointer kompatibel sein und muss immer auf eine bestimmte Speicheradresse zeigen)
- Struckts
  - typedef struct... struct_name; struct_name *spointer; struct_name s;spointer= &s
  - Zugriff:
    (*spointer)....= inhalt
    spointer− >...= inhalt

- Arrays
  - typ typFeld[var]; typ *typPointer=typFeld (typPointer zeigt auf das Feld mit dem Index 0, typPointer+i zeigt auf das feld mit dem index i)
  - Dereferenzieren: *(typPointer+i)=var; oder typPointer[i]=var;(genau wie bei arrays)
  - array kann ohne arrayindex durchlaufen werden
- immer vorsicht bei Pointerarithmetik
- man kann pointer wie normale variablen casten
- Casten muss immer mit Vorsicht durchgeführt werden

3.1.13 Dynamische Speicherverwaltung

- Arbeitsspeicher

  Stack(von dem PC verwaltet beinhaltet, Variablen/Parameter)

  Heap(wird durch den Entwickler verwaltet beinhaltet dynamische Datenstrukturen)
- Bedarfsgerechte Nutzung des vorhanden Speichers, freigeben von nicht mehr benötigtem Speicher, verwendung für dynamische Datenstrukturen(variable Größe)

  $\Leftarrow$ Effiziente Nutzung, Verwaltung liegt beim Entwickler
- Heap starte bei der 1. Adresse, Stack startet bei der Letzten;
- Speicherplatzanforderung:

  malloc()
  - void *p; p=malloc(Anzahl_Bytes);, liefert einen Pointer auf Speicherbereich der benötigten Größe, oder Null falls Speicherbereich der benötigten Größe nicht mehr vorhanden
  - Anzahl der Byte mittels sizeof(typ) ermitteln
- free()
  - free(pointer); gibt den mit malloc reservierten speicherplatz frei(darf nicht mehrfach auf den selber verwendet werden)
  - freigegebene pointer dürfen nicht mehr dereferenziert werden
  - die Speicherplatzfreigabe sollte mit dem gleichem pointer wie die Anforderung erfolgen
  - nach der verwendung immer freigeben
- Memory Leaks:

- im Heap reservierter Speicherbereich jedoch nicht mehr zugänglich (über Pointer erreichbar)
- Probleme:
  führt zu Speichermangel,
  möglicherweise abnormale Programmbeendigung
- Vermeidung
  sorgfältige Programmierung Dynamischer Speicherverwaltung

3.1.14 Funktionen

- Deklaration: Ergenistyp funktionsname (Parameterleiste) funktionsrumpf
- wenn keine ausgabe "void" als ergebnisstyp, sonst "return ergebnis;"(kann auch mehrmals vorkommen)
- lokale Varaibeln, Konstante, Parameter sind nur innerhalb der Funktion bekannt, namensgleiche globale werden überschattet
- funktionen möglichst klein halten (100zeilen) sinnvolle ausgaben (Rückgabe fehlercode) keine globalen variabeln in funktionen
- übergabeprinzip: call by value beim aufruf werden die parameter durch kopieren ersetzt
- parameter werden im stack bereich übergeben, nach schließen der funktion wird dieser frei
- funktionen können auch rekursiv sein
- es wird immer ein wert zurückgegeben
- Rückgabe meherere Werte:
  - Rückgabe eines Typs
  - void functionname (typ *i)... return; funktion schreibt direkt über die adressen in den speicher
  - Per Referenz
- Funktionspointer
  - typ (*fPointer) (typ); zeigt auf einen funktion mit Returntyp typ und einem typ als Parameter
  - wie bei herkömmlichen pointer dereferenziert

3.1.15 Dateien

- Datentyp FILE ist in C vordefiniert
- Deklaration: FILE *datei;

  datei=fopen("c:
dateiname.dateiendung","modus")

  modus: "r" Read "rb" ReadBinary "w" write "wb" write binary "a" Append "ab" append binary

  fscanf(datei," $<$ %[$Flag$][$Breite$][.PrÃzision][PrÃfix]$Typ >$ ", zeile);

- Binärer zugriff:
  - Lesen fread:

    fread(Puffer_Adresse, Puffer_Größe, Anzahl_Puffer, FILE *datei)
    Puffer_Adresse= Pointer auf dateibereich
    Puffer_Größe = Anzahl Bytes des Datenbereichs (sizeof())
    Azahl_Puffer = Anzahl der zu lesenden Records
    datei = dateipuffer
  - schreiben fwrite(Puffer_Adresse, Puffer_Größe, Anzahl_Puffer, FILE *datei)
- fclose(File *datei); (nach zugriff immer schließen)