C Übersicht

21.1.2007 — V1.6 [c-intro-HOWTO.txt]

Inhaltsverzeichnis

1 C-Historie

- 1969: Anfänge von C (Vorläufer waren CPL, BCPL und B).
- 1973: UNIX erstmals auf C portiert (d.h. C hat einen gewissen "Reifegrad" erreicht).
- 1978: Buch "The C programming language" von Kernighan & Ritchie erscheint.
  - ▷ 1. Sprachdefinition von C = K&R-C.
- 1983: Beginn der Normierung von C als ANSI-Standard.
- 1989: Abschluss des ANSI-C-Standards.
  - ▷ Buch "The C programming language" erscheint in der 2. Auflage.
  - ▷ 2. erweiterte Definition von C = ANSI-C.
- 1995: Erweiterungen um Multilanguage-Eigenschaften.
- 1999: Erweiterungen ANSI-C99 (Datentyp bool, complex, . . .).

2 Philosophie von C

- Keep it small and simple (KISS).
- Trust the programmer, don't do any runtime checks.
- Make it fast, even if it is not guaranteed to be portable.
- Provide only one way to do an operation (orthogonal).
- Don't prevent the programmer from doing what needs to be done.

3 Eigenschaften von C

- Allgemein verwendbare, strukturierte, prozedurale Hochsprache (Datentypen, Kontrollstrukturen, Datenstrukturen, Funktionen).
  - ▷ Streng typisiert (integer, float, string, . . .)
  - ▷ Ein C-Programm = "Sammlung von Funktionen" (mind. "main()").
  - ▷ Formatfrei ("Whitespace" trennt "Token").
  - ▷ Unterstützt modulare Programmierung.
- Relativ "kleiner" Sprachumfang.
  - ▷ 32/35 Schlüsselworte + 45 Operatoren.
  - ▷ Auslagerung vieler Funktionalitäten in eine Standard-Bibliothek (z.B. Ein/Ausgabe)

- Erzeugt sehr kompakte + effiziente Programme.
  - ▷ "Maschinen**un**abhängiger Assembler" (wenig Overhead gegenüber Assembler).
- C-Programme sind portabel (Betriebssystem-unabhängig).
  - ▷ Viele Compiler erzeugen C als Zwischencode (z.B. C++, Modula, Eiffel).
  - ▷ Viele Interpreter in C geschrieben (Sed, Awk, Perl, PHP, Lua, Tcl/Tk)
  - ▷ Viele Server in C geschrieben (Apache, qmail, sendmail, MySQL).
- Unterstützt hardwarenahe Programmierung.
  - ▷ Betriebssystem-Implementierung (UNIX, Linux, Windows).
  - ▷ Gerätetreiber-Implementierung (Grafikkarten, . . .).
  - ▷ Embedded Controller (Kfz, Handy, Videorecorder, . . .).

4 Besonderheiten von C

- Viele Operatoren (45 Stück).
  - ▷ Inkrement / Dekrement.
  - ▷ Bitoperatoren (AND, OR, EXCLUSIVE OR, INVERT).
- Zeiger (Speicherzugriff per Adressen).
- Präprozessor (Vorverarbeitung).
- Direkter Speicherzugriff möglich.
- Datentypgrößen (z.B. int) nicht extakt festgelegt.
  - ▷ Nur Minimalanforderungen!
  - ▷ 8 Bit $\Leftarrow$ 16 Bit $\Leftarrow$ short $\Leftarrow$ int $\Leftarrow$ 32 Bit $\Leftarrow$ long.

5 Vorteile von C

- Überall verfügbare, standardisierte Hochsprache.
- Portabel.
- Hohe Geschwindigkeit des Compilers und der erzeugten Programme. (Faktor 1.5-3 gegenüber Assembler).
- Kleiner Sprachumfang + Standardbibliothek.
- Viele Bibliotheken und Tools verfügbar.
- Zeiger ;-)

6 Nachteile von C

- Von Programmierern für Programmierer.
  - ▷ Motto: "Der Programmierer weiß was er tut".
- "Kryptische" Syntax (für den Anfänger).
- Trennung von C-Compiler und lint ("C-Prüfer").
  - ▷ C-Compiler = Reiner Übersetzer mit wenig Prüfungen.
  - ▷ lint = Syntax- und Semantik- und Konsistenz-Checker.
  - ▷ Historisch begründet (Speicherkapazität/Prozessorleistung).
- Keinerlei Laufzeitchecks (Zeiger, Array, Strings, Zahlenüberlauf).
- Operatorvorrang teilweise problematisch (Bit- und Shift-Operatoren).
- Zeiger ;-(

7 Zu C verwandte Programmiersprachen

- Objective C (objektorientierter Nachfolger von C).
- C++ (objektorientierter Nachfolger von C, vormals "C with Classes").
- Java (von SUN, objektorientiert, erzeugt plattformunabhängigen Code).
- C# (Microsoft-eigene Mischung aus C++ / Java mit eigenen Erweiterungen, "C-sharp").
- Awk (Skriptsprache zur Textverarbeitung).
- Perl (sehr mächtige Skriptsprache — "UNIX in a box").
- PHP (sehr mächtige Skriptsprache).

8 Programmiersprachen-Typen

1. Maschinencode
   - ▷ Prozessor-Befehle in Form von Bytes.
   - ▷ Kann sich kein Mensch merken.
2. Assembler
   - ▷ Namen für Befehle / Variablen / Konstanten / Speicherbereiche.
3. Algorithmische Hochsprachen.

   - ▷ FORTRAN, ALGOL, BASIC.
4. Problemorientierte Hochsprachen.
   - ▷ C, Pascal, COBOL, Modula.
5. Objektorientierte Hochsprachen.
   - ▷ C++, Java, C#, Oberon, Smalltalk, Delphi, Ada, Eiffel.

9 Grundlegende Begriffe

- Folgende **Programmier-Konzepte** sollten bekannt sein:
  - ▷ Variablen
  - ▷ Operator-Vorrang und -Assoziativität
  - ▷ Verzweigungen und Schleifen
  - ▷ Funktionen
  - ▷ Arrays
  - ▷ "Mit einem Texteditor eine Quellcode-Datei bearbeiten"
  - ▷ Fehlersuche ;-(
- Folgende Begriffe sollten bekannt sein:
  - ▷ Kommando (Command)
  - ▷ Rechenzeichen (Operator)
  - ▷ Trennzeichen (Separator)
  - ▷ Leerraum (Whitespace)
  - ▷ Kommentar (Comment)
  - ▷ Konstante (Literal)
  - ▷ Variable
  - ▷ Bezeichner (Identifier)
  - ▷ Anweisung (Statement)
  - ▷ (Lexikalischer) Block
  - ▷ Ausdruck (Expression)
  - ▷ Zuweisung (Assignment)
    - ○ Left hand side (LHS)
    - ○ Right hand side (RHS)
  - ▷ Gültigkeitsbereich / Existenzbereich
  - ▷ Kontrollstrukturen
    - ○ Sequenz
    - ○ Verzweigung

        - Schleife
        - Unterprogramm
        - Vorzeitiger Abbruch
    - Deklaration
    - Definition
    - Initialisierung
    - Funktion
        - Aufruf
        - Argumente
        - Übergabe-Parameter
        - Rückgabe-Parameter
        - Rücksprung
        - Call be value (Wert)
        - Call be reference (Adresse)
    - Zeiger (Pointer)
    - Übersetzungseinheit (Module = Datei)
    - Übersetzungszeit (Compile-Time)
    - Laufzeit (Run-Time)
    - Debugger

10 C-Datentypen

- Leerer Datentyp (void)
- Elementare Datentypen
    - Zeichen ('x')
    - Zahl
        - Ganzzahl
        - Gleitkommazahl
    - Aufzählung (Enumeration)
    - Zeiger (Pointer)
- Zusammengesetzte Datentypen
    - Feld (Array, Vektor)
        - Zeichenkette ("String")
    - Struktur (Record)
    - Union (Varianter Record)

11 Wertebereich der numerischen Datentypen

- Ganzzahl
  - ▷ char: -128 ... 127 (1 Byte)
  - ▷ unsigned char: 0 ... 255 (1 Byte)
  - ▷ short: -32768 ... 32767 (2 Byte)
  - ▷ unsigned short: 0 ... 65535 (2 Byte)
  - ▷ int: Je nach Compiler wie short oder long (2/4 Byte)
  - ▷ unsigned int: Je nach Compiler wie unsigned short oder unsigned long (2/4 Byte)
  - ▷ long: -2147483648 ... 2147483647 (4 Byte)
  - ▷ unsigned long: 0 ... 4294967295 (4 Byte)
  - ▷ long long: -9223372036854775808 ... 9223372036854775807 (8 Byte)
  - ▷ unsigned long long: 0 ... 18446744073709551615 (8 Byte)
- Gleitkommazahl
  - ▷ float: 1.17549435e-38 ... 3.40282347e+38 (4 Byte, 6-7 Stellen Genauigkeit)
  - ▷ double: 2.2250738585072014e-308 ... 1.7976931348623157e+308 (8 Byte, 15-16 Stellen Genauigkeit)
  - ▷ long double: ??? (10/12 Byte, 19/23 Stellen)

12 IDE (Integrated development environment)

- Integrierte Entwicklungsumgebung mit:
  - ▷ Projekt- und Dateiverwaltung
  - ▷ Editor (Syntax-Higlighting)
  - ▷ Compiler (Warnstufen)
  - ▷ Debugger
  - ▷ Browser
  - ▷ Make-System
  - ▷ Hilfestellung und Dokumentation
- Beispiel: Kdevelop, Visual Studio C++, Borland Delphi, Eclipse

13 Dateiarten beim C-Programmieren

Programme sind ASCII-Dateien, d.h. sie enthalten nur Zeichen aus aber keine Formatierungen wie Fettschrift, ... Sie sind daher mit einem ASCII-Editor zu erstellen (nicht mit WORD). Dateiendungen:

- Quellcodedatei (Source): *.c (oder *.cpp/*.cxx/*.c++/*.C in C++)
- Headerdatei (Source mit Deklarationen): *.h (oder *.hpp/*.H in C++)
- Objektdatei (übersetzter Maschinencode): *.o (*.obj unter Windows)
- Statische Library (Bibliothek von Objektdateien): *.a (*.lib unter Windows)
- Dynamische Library (Bibliothek von Objektdateien): *.so (*.dll unter Windows)
- Ausführbare Datei (Programm, Executable): * (x-Recht unter UNIX, *.exe/com unter Windows)

14 Übersetzungsschritte

- Editieren: Sourcedatei erstellen
- Präprozessor: Sourcedatei umformen (Textersatz, Header einbinden) ⇒ Sourcecodedatei
- Compilieren/Übersetzen: Sourcedatei ⇒ Objektdatei
- Linken/Binden: Objektdateien + Bibliotheken ⇒ Ausführbares Programm
- Debuggen: Ausführbares Programm + Sourcedatei
- Bibliotheken erstellen: Objektdateien ⇒ Bibliothek

15 Speicherklassen

1. Programmglobal (statisch)
   - ▷ "extern" (oder nichts)
   - ▷ 1x initialisiert (Std: 0)
2. Modullokal (statisch)
   - ▷ "static" am Modulanfang
   - ▷ 1x initialisiert (Std: 0)
3. Funktionslokal (statisch)
   - ▷ "static" am Funktionsanfang
   - ▷ 1x initialisiert (Std: 0)
4. Funktionslokal (dynamisch)
   - ▷ "auto" (oder nichts) am Funktionsanfang,
   - ▷ Nx bzw. nicht initialisiert

    - ▷ Parameter von Funktionen gehören zur gleichen Klasse, sind allerdings initialisiert
5. Blocklokal (statisch)
    - ▷ "static" am Blockanfang
    - ▷ 1x initialisiert (Std: 0)
6. Blocklokal (dynamisch)
    - ▷ "auto" (oder nichts) am Blockanfang
    - ▷ Nx bzw. nicht initialisiert

16 Speicher-Layout

- Feste Größe (durch Compilation ermittelt):
    - ▷ Codesegment ("Text") — Code
    - ▷ Datensegment — (Statische) Variablen, Konstanten, Texte
- Veränderliche Größe (zur Laufzeit angelegt):
    - ▷ Stack — Rücksprungadressen, lokale Variablen (automatisch verwaltet)
    - ▷ Heap — Dynamischer Speicher (manuell zu verwalten)

17 Bestandteile eines C-Programms (Reihenfolge)

- Datei-Kopf (Kommentar: Name, Zweck, Version, Autor, Datum, Todo, Done, . . .)
- Präprozessor-Anweisungen.
    - ▷ Includes von Header-Dateien *.h.
    - ▷ Konstanten-Definitionen.
    - ▷ Makro-Definitionen.
    - ▷ Bedingtes Kompilieren
- Datentyp-Deklarationen.
- Variablen-Definitionen + Initialisierungen.
- Funktions-Deklarationen (Prototypen).
- Sammlung von Funktions-Definitionen.
    - ▷ Genau eine Funktion names main() ist Startpunkt.

18 Bestandteile einer C-Funktion

- Kopf besteht aus:
  - ▷ Rückgabe-Datentyp
  - ▷ Funktions-Name
  - ▷ Übergabeparameter mit Datentyp
- Rumpf (Block) bestehend aus:
  - ▷ Lokale Variablendefinitionen
  - ▷ Konstrollstrukturen (Verzweigungen, Schleifen)
  - ▷ Funktionsaufrufen
  - ▷ Anweisungen
  - ▷ Unterblöcken

19 Phasen der Software-Entwicklung

Im wesentlichen gibt es folgende Phasen, die allerdings kaum jemals linear, sondern meist zyklisch durchlaufen werden. An bestimmten Stellen in diesem Prozess sollte immer ein **schriftliches Dokument** vorliegen, das von allen Parteien (Anwender, Auftraggeber, Entwickler, Tester(QC), Dokumentationsabteilung, Geschäftsleitung, ...) **abgezeichnet** wird. Dies dient der Sicherheit aller Beteiligten, dass das Problem korrekt verstanden + beschrieben ist und adäquat gelöst wird. Mit Hilfe dieser Dokumente kann bei strittigen Punkten der Nachweis geführt werden, wo ein Fehler auftrat und wer ihn zu verantworten hat.

Der Aufwand verteilt sich grob zu jeweils 1/3 auf die 3 Hauptphasen Anforderungsdefinition, Entwicklung und Test.

- **Anforderungsdefinition**
  - ▷ Anforderung vom Anwender $\Rightarrow$ "Lastenheft"
  - ▷ Analyse $\Rightarrow$ "Pflichtenheft" + Aufwandsschätzung
  - ▷ Eventuell "Prototyp" für Oberfläche erstellen und mit Anwender diskutieren
  - ▷ Pflichtenheft **schriftlich** vom Anwender absegnen lassen
  - ▷ Auftrag **schriftlich** vom Anwender erteilen lassen
- **Entwicklung**
  - ▷ Entwurf + Design (Oberfläche, Schnittstellen, Algorithmen, Datenbank-Tabellen)
  - ▷ Implementierung ("eigentliche" Codierung)
  - ▷ Test (durch Entwickler, möglichst automatisiert!)
  - ▷ Programm-Dokumentation (während der Entwicklung, möglichst im Quellcode)

- ▷ Freigabe für die Qualitätskontrolle (QC)

- **Test**
  - ▷ Alpha-Test durch die Qualitätskontrolle (QC) (Fremdpersonen testen, nicht Entwickler)
  - ▷ Integrationstest (in Produktionsumgebung)
  - ▷ Freigabe für Anwender
  - ▷ Beta-Test durch Anwender $\Rightarrow$ **schriftliche** Abnahme
  - ▷ Anwender-Dokumentation erstellen (evtl. nicht notwendig, häufig aus Kostengründen)
  - ▷ Freigabe für Produktion

Die anschließende WARTUNG darf nicht vergessen werden, ohne sie ist jedes SW-Produkt auf die Dauer zum Tode verurteilt!

Wichtig ist last but not least, die Abrechnungsmodalitäten und den Zeitpunkt der Abrechnung (Erreichen von Meilensteinen) festzulegen.

20 Häufige Fehler

- Semikolon ";"
  - ▷ nach Anweisung vergessen.
  - ▷ nach Strukturdefinition vergessen.
  - ▷ nach if/while/for/switch (. . .); zuviel.
  - ▷ nach Präprozessor-Anweisung #define zuviel.
  - ▷ mit "," verwechselt (z.B. in for (. . .))
- Bei Vergleich "=" (Zuweisung) statt "==" verwendet.
- Groß/Kleinschreibung nicht beachtet.
- Tippfehler bei Variablen/Funktionsnamen.
- Einrückung entspricht nicht Ablauflogik.
- Abschließendes " bei Strings oder ’ bei Zeichen vergessen
- Nullbyte ’\0’ am Stringende vergessen (um 1 zu kurz).
- "t" und ’text’ verwechselt (String $\leftrightarrow$ Zeichen).
- Klammerfehler bei Blöcken {. . . } und Ausdrücken (. . .).
- Klammern () nach Funktionsnamen vergessen.

- Rückgabedatentyp bei Funktionsdefinition (Std: int) und/oder return WERT; am Funktionsende vergessen.
- (Lokale) Variablen nicht initialisiert.
- Deklaration/Definition von Variablen, Funktionen und Datentypen vor Verwendung vergessen.
- Deklaration und Definition verwechselt.
- Bei printf()/scanf() Anzahl bzw. Reihenfolge der Formatelemente und Parameter verschieden.
- Bei scanf() Adressoperator & vor Parameter vergessen.
- "*.c" und "*.cpp"-Dateien gemischt (problematisch wg. "Name Mangling bei C++").
- Schlüsselwort falsch geschrieben / nicht vorhanden (z.B. wile, swich, then, elsif, until)
- Schlüsselwort als Variablen- / Funktions- / Typ-Name gewählt.
- Bei Zählschleifen "um 1 vorbei" gezählt (asymmetrische Grenzen wählen!).
- Von malloc() gelieferten Speicher nicht initialisiert.
- Zeilenende bei UNIX (\n), DOS/Windows (\r\n) und Apple Mac (\r) unterschiedlich (Text/Binär-Mode in fopen()).

21 Tips

- Nur 1 Anweisung pro Zeile.
- Per Tabulator einrücken (Breite: 4).
- Code immer gut lesbar für andere schreiben:
  - ▷ Standard-Dateikopf mit den wichtigsten Daten.
  - ▷ Text sauber formatieren.
  - ▷ Leerraum um Operatoren setzen.
  - ▷ Leerzeilen um zusammengehörige Codeteile setzen.
  - ▷ Nicht alles in eine Zeile,
  - ▷ Ausreichend kommentieren.
- Variablen- und Funktionsname gut wählen.
- Konstanten einführen und GROSS schreiben (z.B. DM2EURO = 1.95538).
- "Defensiv" programmieren.
- Immer höchste Warnungsstufe einstellen (Windows: "/W4", UNIX: "-Wall") + alle Warnungen beseitigen (alles Unverständliche ist ein potentieller Fehler).

- Fehlermeldungen **genau** lesen, in 99
- Nur ersten Syntaxfehler beheben, dann neu übersetzen (sonst behebt man Folgefehler). (Anspringen mit "Doppelklick" auf Fehlermeldung).
- Programm auch während dem Eintippen und Rumprobieren ständig korrekt einrücken (Klammerfehler mit {...} werden so vermieden).
- Erst Klammern {...} und (...) schreiben, dann ihren Inhalt (Klammerfehler werden so vermieden).
- Immer default-Zweig in switch-Anweisung vorsehen (evtl. nur mit einer Fehlermeldung)
- In switch-case das break auf extra Zeile setzen.
- In switch-case absichtlich fehlendes break mit Kommentar /* nobreak */ kommentieren.
- goto nur zum Abbruch verschachtelter Schleifen oder Schleifen mit eingeschachteltem switch verwenden.
- Programm zum Abschluß "wasserdicht" machen (d.h. Eingabefehler, fehlende Dateien, ... abfangen).
- Programme nicht von Null auf neu schreiben, sondern ein altes Programm kopieren + verändern (alle Stellen!).
- Standard-Kopf kopieren + anpassen.
- Auf der Kommandozeile mit 3 Fenstern abreiten: A: editieren – B: übersetzen – C: testen.
- Programme schrittweise entwickeln und immer wieder testen ("Rom wurde auch nicht an einem Tag erbaut").
- Komplizierte Programmteile erst in extra Programm ausprobieren, bis Syntax und Funktionalität korrekt sind (dann mit der Maus kopieren).
- Debug-Anweisungen nicht sofort wieder löschen, sondern mit // auskommentiere (später suchen + ersetzen).
- Präprozessor-Tricks verwenden, um den Code lesbarer / wartbarer zu machen (z.B. Debugging mit DEBUG steuern).
- Programm mit verschiedenen C-Compilern übersetzen (jeder findet andere Fehler).
- Programm auf verschiedenen Betriebssystemen und HW-Architekturen laufen lassen (unterschiedlich empfindlich gegenüber Fehlern).
- Testen, testen, testen (Tests wenn möglich automatisieren).
- Bei hartnäckigen Fehlern einen Kollegen zu Hilfe holen und ihm das Programm erklären (er muß es gar nicht verstehen!).

- Bei ständiger Wiederholung des gleichen Fehlers über die Ursache nachdenken ("eingefahrenes Verhalten").
- Aus fremden Programmen lernen (z.B. Linux-Quellcode).
- Duplizierung von Code vermeiden (änderungsunfreundlich, statt dessen in Funktion auslagern).
- Bei Änderungen Usage-Meldung konsistent halten.
- Eingaben besser per Programm-Parameter argc und argv oder direkt im Programm eintragen statt mit scanf() lesen (macht wiederholte Tests deutlich einfacher).
- Suchpfad $PATH bzw. PATH enthält Programm-Verzeichnis nicht.
- Es erfolgt "stillschweigender" Überlauf (z.B. ++127 $\Rightarrow$ -128 bei (signed) char)
- Binäre Gleitkommazahlen können nicht alle Dezimalbrüche exakt darstellen (Rundungsfehler)

— END —