DI Christian Bucher aka /cbx

Skriptum zur Vorlesung „Softwareentwicklung“
an der fh Kufstein im Wintersemester 2004/05

„Die wundersame Welt der Softwareentwicklung“

oder

„Warum uns uns Softwareentwicklung in C zu einem besseren Sexualleben verhilft“

oder

auch nicht.

Die wundersame Welt der Softwareentwicklung

1.Inhalt

Inhaltsverzeichnis

2. Prä-Prolog

Vor das Vorwort sei noch folgender Satz gestellt:

„DON'T PANIC“ [Douglas Adams] oder übersetzt:

„Versuchen Sie, dies hier nicht ernster zu nehmen als unbedingt nötig“ [cbx]

3. Prolog

Vorab seien folgende Worte zur meditativen Kontemplation gestellt:

- *„Programmieren ist Abenteuer im Kopf“* [Hannes Rieser]
- *„Der Computer löst Probleme, die man ohne ihn nicht hätte.“* [Volksweisheit]
- *„Vor dem PC bitte erst Hirn booten“* [Anweisung im LRZ der TU München]
- *„Wenn Du glaubst, mit dem Computer ein Problem lösen zu können, hast Du noch gar nichts verstanden“* [cbx]
- *„Hüte Dich vor Weisheiten, die in einem Satz ausgedrückt werden können“* [cbx-Paradoxon]

Mit dieser Basisausstattung an Lebensweisheit sollte es gelingen, sich der Kunst der Programmierung von Mikrocomputern (also PCs) zu nähern. Dass hierbei die Betrachtung von PCs gegenüber den Großrechnern dominiert hat mehrere Gründe:

- Sämtliche Übungen werden auf PC-Systemen stattfinden.
- In den letzten Jahren haben sich die Programmierparadigmen beider Welten stark aneinander angenähert.
- Auch anspruchsvolle Client/Server-Systeme werden heute auf gewöhnlicher PC-Architektur implementiert.
- Die Programmierung von Mainframes (z.B. auch noch in COBOL) stellt eine (finanziell zugegebenermaßen sehr lukrative) Nische dar.
- Im Bildungsbereich stehen ohnehin kaum Mainframe-Systeme zur Verfügung

Mit der *aktiven* Beherrschung einer Programmiersprache der dritten Generation (3GL) erwirbt man sich einen Erfahrungsschatz, der die Einarbeitung in weitere Sprachen der dritten und vierten Generation wesentlich vereinfacht.

Ein kleiner historischer Abriß soll den Einstieg plastisch gestalten:

4.Historischer Abriss

Die Ära der persönlichen Computer, die auch gewöhnlichen Privatpersonen das „Programmieren“ ermöglichte, begann vielleicht 1975 mit dem Altair 8800, einem Gerät von bemerkenswert nichtexistentem praktischen Nutzwert.

Dennoch markiert der Altair 8800 einen Meilenstein, da er nicht nur als erster in Serie hergestellter Consumer-Computer einen industriell standardisierten Bus (S100-Bus) anbot sondern auch, weil etwas später ein unbekanntes Startup-Unternehmen mit dem Namen *Microsoft* einen BASIC-Interpreter dafür programmierte. Der Altair war um den damals neuen Intel 8080 Prozessor aufgebaut und brachte mit 2MHz Taktfrequenz eine damals kaum vorstellbare Rechenleistung in die Bastelzimmer (wozu auch immer).Der Altair wurde über Kippschalter an der Front direkt in 8080-Maschinensprache programmiert.

Bis nach 1980 dominierten auch im Bereich des Office computing 8bit-Architekturen um die Prozessoren Intel 8080, 8085 und Zilog Z80, die mit Taktfrequenzen bis 8Mhz und Speichergrößen bis 64kBytes RAM über serielle Terminals durchaus anspruchsvolle Mehrbenutzersysteme unter dem Betriebssystem CP/M bedienten. Im Standard Lieferumfang dieser professionellen DV-Anlagen befand sich meiste ein BASIC-Interpreter zur Erstellung der erforderlichen Unternehmenssoftware. Kommerzielle Standardapplikationen waren zu dieser Zeit noch nicht üblich.

Erst 1981 brachte IBM mit dem 8088-Prozessor des *PC* halbherzig die modernere 16bit-Technologie in den Mainstream ein.Auch der IBM-PC verfügte über ein (wenigstens ansatzweise) „standardisiertes“ Bussystem und ermöglichte es somit Drittherstellern, eigene Hardwareerweiterungen anzubieten.

Mit dem Erfolg des PC tauchten (wenigstens ansatzweise) auch andere Programmiersprachen im Blickfeld der Programmierer auf. Insbesondere die Firma Borland machte sich damals mit dem legendären *Turbo-Pascal*, einer sehr effizient und stabil arbeitenden Implementierung des Pascal-Standards von Nikolaus Wirth, einen guten Namen.

Weiterhin konnte auch die damals bereits allgegenwärtige Firma Microsoft mit ihrer

Entwicklungsumgebung *Programmers Workbench* und dem ursprünglich von Lattice gekauften C-Compiler mittelfristig der im UNIX-Bereich populärsten Programmiersprache C am PC eine Vormachtstellung verschaffen. Schließlich wurde C, versehen mit diversen Standardisierungen und Erweiterungen, zusammen mit dem Nachfolger C++ zur derzeit meist benutzten Programmiersprache.

Genau deshalb, und wegen einiger sehr unangenehmer Eigenschaften dieser Sprache, werden auch wir uns zum Einstieg in die Programmierung mit ANSI-C befassen und nach einigen schmerzhaften Basiserfahrungen den Aufstieg nach C++ oder C# wagen.

Unser erstes Ziel für das erste Semester soll es sein, dass alle Teilnehmer der Vorlesung über das folgende Kapitel lachen (bzw. wenigstens schmunzeln) können und auch wissen, warum. Deshalb hier der erste Test:

5.Shooting Yourself in the Foot in different languages

Im Lauf der stürmischen Entwicklung der Rechnertechnik seit den 60er Jahren des letzten Jahrhunderts haben sich auch die Methoden und Denkmodelle zur Programmierung ebenso entwickelt. In den letzten knapp 50 Jahren sind so zahlreiche mehr oder minder exotische Sprachen und Dialekte entstanden, die sich meist, in unterschiedliche Verbreitung, bis heute gehalten haben. Der folgende Klassiker gibt einen groben Überblick über einige heute noch relevanten Sprachen:

It was long ago that someone once said that using the C programming language, you can shoot yourself in the foot. Every now and then I came across what would happen if you tried to shoot yourself in the foot using other programming languages. So, I've collected all the ones I could find. Here they are.

APL

1. *You shoot yourself in the foot, then spend all day figuring out how to do it in fewer characters.*
2. *You hear a gunshot and there's a hole in your foot, but you don't remember enough linear algebra to understand what has happened.*
3. *@#&^$%&%^ foot*

Assembler

1. *You try to shoot yourself in the foot, only to discover you must first invent the gun, the bullet, the trigger, and your foot.*
2. *You crash the OS and overwrite the root disk. The system administrator arrives and shoots you in the foot. After a moment of contemplation, the administrator shoots himself in the foot and then hops around the room rabidly shooting at everyone in sight.*
3. *By the time you've written the gun, you are dead, and don't have to worry about shooting your feet. Alternatively, you shoot and miss, but don't notice.*
4. *Using only 7 bytes of code, you blow off your entire leg in only 2 CPU clock ticks.*

BASIC (interpreted)

1. *Shoot yourself in foot with water pistol. On big systems, continue until entire lower body is waterlogged.*
2. *Lacking a gun, you hold the bullet in your hand and throw it at your foot....and miss.*

BASIC (compiled)

You shoot yourseld in the foot with a BB using a SCUD missle launcher.

C

1. *You shoot yourself in the foot.*
2. *You shoot yourself in the foot and then no one else can figure out what you did.*

C++

You accidentially create a dozen instances of yourself and shoot them all in the foot. Providing emergency medical assistance is impossible since you can't tell which are bitwise copies and which are just pointing at others and saying, "That's me, over there."

COBOL

1. *USEing a COLT 45 HANDGUN, AIM gun at LEG.FOOT, THEN place ARM.HAND.FINGER on HANDGUN.TRIGGER and SQUEEZE. THEN return HANDGUN to HOLSTER. CHECK whether shoelace needs to be retied.*
2. *Allocate $500000 for the project. Define gun,bullet,foot. Run press_trigger. Go for coffee break. Return in time to put foot under bullet.*
3. *USE HANDGUN.COLT(45), AIM AT LEG.FOOT THEN WITH ARM.HAND.FINGER ON HANDGUN.COLT(TRIGGER) PERFORM SQUEEZE, RETURN HANDGUN.COLT TO HIP.HOLSTER, SCREAM.*
4. *You try to shoot yourself in the foot, but the gun won't fire unless it's aligned in column 8.*

forth

1. *Foot in yourself shoot.*
2. *First you decide to leave the number of toes lost on the stack and then implement the word foot-toes@ which takes three numbers from the stack: foot number, range, and projectile mass (in slugs) and changes the current vocabulary to 'blue'. While testing this word you are arrested by the police for mooning (remember, this is a bottom-up language) who demonstrate the far better top-down approach to damaging yourself.*
3. *BULLET DUP3 * GUN LOAD FOOT AIM TRIGGER PULL BANG! EMIT DEAD IF DROP ROT THEN (This takes about five bytes of memory, executes in two to ten clock cycles on any processor and can be used to replace any existing function of the language as well as in any future words). (Welcome to bottom up programming - where you, too, can perform compiler pre-processing instead of writing code)*

FORTRAN

1. *You shoot yourself in each toe, iteratively, until you run out of toes, then you read in the next foot and repeat. If you run out of bullets, you continue anyway because you have no exception-handling ability.*
2. *You shoot yourself in each toe, iteratively, until you run out of toes; then you shoot the sixth bullet anyway since no exception processing was anticipated.*

HTML

1. *You shoot yourself in the foot, only to find out that no matter how gory the result looks, your foot keeps working. Your foot finally stops working when you stub your toe kicking the box the gun came in.*
2. *Shoot here*

Java

1. *You write a program to shoot yourself in the foot and put it on the Internet. People all over the world shoot themselves in the foot.*
2. *The gun fires just fine, but your foot can't figure out what the bullets are and ignores them.*

Modula-2

After realizing that you can't actually accomplish anything in this language, you shoot yourself in the head.

Pascal

The compiler won't let you shoot yourself in the foot.

PHP

If you're lucky and the HTTP connection doesn't time out, or you mis-spelt a variable name, you shoot yourself in the foot.

Visual Basic

You'll shoot yourself in the foot, but you'll have so much fun doing it that you won't care.

Credits

The above came from the following sources:

Developers' Insight, December 1991

http://www.cs.bgu.ac.il/~omri/Humor/shoot.html

http://www.netfunny.com/rhf/jokes/90q4/shf4.html

http://fortunecity.com/bennyhills/holygrail/42/shootfoot.html

http://www.musc.edu/~adeimaas/Shooting_yourself.html

http://www-i3.informatik.rwth-aachen.de/funny/shootfoot.html

http://www.funnies.com/computer/other/language.htm

http://www.progress.demon.co.uk/Fun/shoot-foot.html

http://paul.merton.ox.ac.uk/computing/foot-lang.html

http://www.leo.org/information/freizeit/fun/funengl.html

http://village.vossnet.co.uk/p/prar/toast.html#shoot

http://m5p.com/~pravn/foot.html

http://carls.lav10.nu/programming.html

6.Das große Warum

Warum wird programmiert? Wollen wir hier nur die Gegenwart betrachten, so ergeben sich folgendes Varianten:

1. Customising: Durch Programmierung werden bestehende große Systeme an Kundenanforderungen angepast (z.B. SAP).
2. Application development: Ein vollständiges Anwendungsprogramm mit (mehr oder weniger) neuen Funktionen wird auf ein Betriebssystem aufgesetzt (z.B. WORD.EXE).
3. OS development: Ein Betriebssystemkern wird neu oder weiter entwickelt (z.B. HURD, LINUX).
4. Embedded Systems: Auf einer sehr kleinen speziellen Plattform wird eine sehr spezifische Funktion direkt implementiert (z.B. ABS)

Satz: *„C ist eine der am universellsten einsetzbaren Sprachen, da sie sehr effizient compiliert wird und sehr weitgehenden Zugriff auf Hardware-Ressourcen erlaubt. Außerdem ist C eine der gefährlichsten Sprachen, da sie sehr effizient compiliert wird und sehr weitgehenden Zugriff auf Hardware-Ressourcen erlaubt.“*

7.Das noch größere Wie

Wie wird programmiert? Im Bereich der Wirtschaftsinformatik kommen lediglich Aufgaben des Typs 1. und 2. vor. Für Customising werden vorwiegend Sprachen der vierten Generation verwendet, für die Applikationsentwicklung vorwiegend Sprachen der dritten Generation wie

- Java (sehr plattformunabhängig mit hohem Ressourcenverbrauch)
- C und C++ (effizient, verbreitet und relativ plattformunabhängig, sehr unsicher)
- Delphi (ein Pascal-Derivat, effizient und sicher, dementsprechend wenig verbreitet)
- C# (modern und stylish, sicherer als C, nicht plattformunabhängig)

Die Psychologie lehrt uns, dass schmerzhafte Erfahrungen den größten Lerneffekt bewirken, weshalb sich C als idealer Lehrinhalt erweist. In dieser Sprache sind noch fast alle prinzipiellen Probleme moderner Programmiersprachen ungelöst, weshalb die Vorteile der moderneren Methoden anschließend um so klarer hervortreten.

Programmieren bedeutet, einem *von jeder Art von Verstand und Intelligenz freien* Gegenüber zu erklären, was er zu tun hat, um eine geforderte Aufgabe zu lösen. Die Eigenschaft „*von jeder Art von Verstand und Intelligenz freie*“ stellt hierbei die entscheidende Herausforderung dar. Es gilt, zur Bewältigung dieser Aufgabe eine Kommunikationsform zu wählen, die beim Empfänger weder das eine noch das andere voraussetzt.

Die Standardlösung für diese Aufgabe besteht bis heute meist in der Definition einer speziellen (meist *problemorientierten*) Sprache, die ein sehr kleines Vokabular und eine Extrem strenge Grammatik (Syntax) aufweist.

Satz: *„Die Kunst des Programmierens besteht lediglich darin, diese Sprache so gut zu beherrschen, dass damit eine Aufgabenstellung korrekt und vollständig ausgedrückt werden kann.“*

8.Wie man C programmiert, ohne den Verstand zu verlieren

Im Gegensatz zu Sprachen wie dem historischen BASIC ist C eine Compilersprache. Dies bedeutet, dass der Programm*text* in einem mehrstufigen Prozess von einem Compiler direkt in die Maschinensprache der CPU des Zielsystems übersetzt wird. Dabei werden alle syntaktischen Fehler bereits vor der Ausführung des Programms erkannt. Compilierte Programme laufen sehr effizient und verbrauchen relativ wenige zusätzliche Ressourcen. Dafür ermöglicht die direkte Umsetzung im Maschinencode im Fehlerfall sehr weitgehende Schadwirkungen.

Im Gegenzug werden beispielsweise bash-Skripte und BASIC-Programme interpretiert, was bedeutet, dass der Programm*text* direkt von einer Applikation (dem Interpreter) analysiert und in Aktionen umgesetzt wird. Syntaktische Fehler werden so meist erst zur Laufzeit erkannt. Interpretierte Programme laufen langsamer und unter höherem Ressourcenverbrauch, im Gegenzug sorgt aber der Interpreter für Schadensminimierung im Fehlerfall.

Um aus der Idee zu einem C-Programm eine lauffähige Applikation zu erzeugen, braucht man im Minimalfall lediglich einen Editor zur Eingabe des *Quelltextes* und einen Compiler zur Erzeugung des Maschinencodes. Hardcore UNIX-Freaks erreichen das (inklusive Ausfuhrung) mit einer einzigen Befehlszeile:

```
hackbox: ~$ vi main.c && cc main.c -o prog && ./prog
```

Da diese Vorgehensweise bei größeren Projekten etwas ineffizient ist, werden heute praktisch ausschließlich IDE (Integrated Development Enviroment) eingesezt. Diese verbinden Editoren Projektverwaltung, Compiler und Debugger unter einer Oberfläche. Die allgemein bekannteste IDE ist das Visual Studio von Microsoft, wir werden die freie, auf dem MinGW basierende Bloodshed DevC++ IDE verwenden. Beide IDEs unterscheiden sich nur kosmetisch voneinander.

Der erste Schritt, im Kontakt mit C einen augenblicklichen Ausfall aller höheren Hirnfunktionen zu verhindern, ist der Einsatz einer Idee und die sichere Beherrschung derselben. Dies kann einige Tage dauern.

Der Zweite Schritt besteht in der Aneignung der geeigneten Fachterminologie, um die eigenen Benühungen adäquat kommentieren zu können. Dies kann einige Wochen dauern

Der dritte Schritt besteht im Erlernen der eigentlichen Programmiersprache in Vokabular (leicht) und Syntax (schwer). Dies kann einige Monate dauern.

Der letzte Schritt besteht im festigen der Kenntnisse und Erwerben von Erfahrung durch praktischen Einsatz. Dies wird einige Jahre dauern.

Beginnen wir also lieber mit den ersten drei Schritten, so lange noch Zeit dazu ist. Erwartungsgemäß beginnen wir mit Schritt drei.

9.Zum Kern vordringen heißt lallen lernen

Ein gültiges C-Programm besteht aus einer Aneinanderreihung von Worten, die aus Buchstaben des 7-bit ASCII-Zeichensatzes bestehen dürfen. Dies bedeutet, dass in einem Wort **Sonderzeichen** wie Umlaute **nicht vorkommen dürfen**. Das allein genügt aber noch nicht.

Die Worte werden durch **Whitespace**, also Leerzeichen getrennt. Dabei sind beliebige Mengen von Leerzeichen (Space), Ttabulatoren und Zeilentrenner weitgehend äquivalent. Dies ermöglicht, wie wir noch kennen lernen werden, albtraumhafte Möglichkeiten der **Quelltextformatierung**.

Erschwerend kommt hinzu, dass strikt zwischen **grossen und kleinen Buchstaben unterschieden** wird.

Aus den Worten werden **Bezeichner** und **Schlüsselwörter** formuliert und zu **Anweisungen** zusammengesetzt. Die Anweisungen werden immer mit einem Semikolon „**;**“ abgeschlossen.

Satz: *„Man beherrscht C sicher, wenn man ohne Nachdenken die Strichpunkte richtig setzt“*

Das Vokabular von C besteht aus so wenigen Worten, dass es trügerisch einfach wirkt:

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

Diese Schlüsselwörter (**keywords**) dürfen als einzige nicht als **Bezeichner** verwendet werden. Unter einem Bezeichner versteht man den Name einer **Variablen**, einer **Funktion** oder eines **Typen**. Ein Bezeichner darf aus allen gültigen (also 7-bit ASCII) Buchstaben bestehen, muss aber mit einem Buchstaben oder eine Unterstrich (**underscore**) beginnen.

Gültige Bezeichner sind z.B.:

Hans, saubled, unter13leichen, HierNichtParken_sie_Idiot, SoASemf, Register, llll

Ungültige Bezeichner sind z:B.:

.super, 15dosenbier, was.auch.immer, Dödel, was?, register, 11111

Wozu diese Quälereien dienen, wird vielleicht aus dem nächsten Kapitel klar.

10.Wie variabel sind Sie?

Programmieren heißt Daten zu verarbeiten. Diese Daten müssen **direkt adressierbar** gespeichert werden und dazu dienen Variablen:

Satz: *„In der Mathematik stellt die Variable eine Unbekannte dar, in der Informatik hingegen stellt die Variable eine Bekannte dar. Eine Variable ist ein benannter Speicherort für Daten“*

Deshalb hat eine Variable drei grundlegende Eigenschaften

- einen Namen
- einen Typ und
- einen Wert

Der **Name** soll in einer erkennbaren Relation zur Verwendung der Variablen stehen, da hiermit eine gute Selbstdokumentation erreicht wird.

Der **Typ** gibt einerseits an, wie viel Speicher für die Variable alloziert werden muss, andererseits, wie der Inhalt dieses Speichers interpretiert werden muss.

Der **Wert** schließlich ist naheliegenderweise das, was in der Variable gespeichert wird, üblicherweise Text oder Zahlenwerte.

C kennt nur sehr wenige generische Typen:

- char: ein numerischer 8 bit Typ
- short: ein numerischer 16 bit Typ
- long: ein numerischer 32 bit Typ
- int: ein numerischer 16-32 bit Typ
- float: ein numerischer 32 bit Fließkommatyp
- double: ein numerischer 64 bit Fließkommatyp
- bool: ein C++ Metatyp, mit einem bit Informationsgehalt, der meist auf einen größeren Typ abgebildet wird.

Achtung: C kennt **keinen Typ** zum Speichern von **Text!**

Variablen müssen in C *deklariert* werden, sodass der Compiler entscheiden kann,

- wie viel Speicher für die Variable alloziert werden muss und
- wie der Inhalt dieses Speichers zu interpretieren ist.

Die Variablendeklaration erfolgt in C durch das Nennen des Typs, gefolgt von einem oder mehreren, durch Kommata getrennten, Variablennamen und optionalen Initialisierungen.Beispiel

```
int nTest, nJohn=7, nAllan=55;
short sDick;
```

Wichtig hierbei ist, zu beachten, dass C als *high performance system* keinerlei implizite Initialisierungen vornimmt. Es folgt somit folgender wichtige

Satz: *„Die **Deklaration** einer Variable in C führt **keine Initialisierung** des Wertes durch. Der Wert einer deklarierten Variablen ist **unbestimmt**“*

11.Ich sehe was, was Du nicht siehst

Variablen sind stets innerhalb des Codeabschnitts sichtbar (und damit verwendbar), in dem sie deklariert wurden. Man unterscheidet grob zwischen *globalen* und *lokalen* Variablen.

Globale Variablen werden *ausserhalb* von Funktionen im Kopf eines Quelltextmoduls deklariert und sind für *alle Funktionen* innerhalb dieses Moduls sichtbar. Sie werden üblicherweise im BSS (Block Storage Segment) des Systems angelegt. Die Lebensdauer globaler Variablen erstreckt sich üblicherweise auf die gesamte Programmlaufzeit.

Lokale Variablen werden *innerhalb* von Funktionen oder (in C++) Anweisungsblöcken deklariert und sind lediglich innerhalb dieser Funktion (bzw des Blocks) sichtbar. Sie werden üblicherweise am Stack des Systems angelegt. Die Lebensdauer lokaler Variablen erstreckt sich üblicherweise nur auf die Laufzeit der Funktion bzw. des Blocks.

Für den Anfänger empfiehlt sich die Berücksichtigung folgenden Satzes:

Satz: *„Die Verwendung globaler Variablen sollte auf Fälle beschränkt werden, wo deren Notwendigkeit überzeugend begründet werden kann“*

12.Ausgeben!

Damit das langsam entstehende Programm überhaupt mit dem Benutzer interagieren und ihn über den Inhalt von Variablen informieren kann, benötigt es eine Bildschirmausgabe. Diese kann in Art unserer Vorfahren als einfache Textausgabe am Bildschirm erfolgen. Dazu bemüht C die Bibliotheksfunktion *printf() „print with format“*.

Die printf()-Funktion nimmt als erstes Argument einen C-String, der die Formatierung beinhaltet sowie eine Variable für jeden im *Formatstring* enthaltenen Platzhalter. Beispiel:

```
int a = 7;
printf ("a hat den Wert %d\n",a);
```

Folgende Liste der zeigt die möglichen Platzhalter und Escape-Sequenzen:

```
%d %i Decimal signed integer.
%o    Octal integer.
%x %X Hex integer.
%u    Unsigned integer.
%c    Character.
```

```
%s    C-String.
%f    double
%e %E double.
%p    pointer.

%%      %

\n    newline
\r    carriage return
\t    tabulator
\b    backspace
\a    bell
\f    form feed
\\    \
```

Genaueres dazu bietet stets die Online-Hilfe oder die Seite der Uni Bern.

13.Und jetzt wird abgerechnet (und zugewiesen)

Angeblich stellt die herausragendste Fähigkeit der Computer ja das Rechnen dar. Deshalb soll nun erläutert werden ,wie Berechnungen in C notiert werden. Zum Rechnen werden *Ausdrücke* mit *Operatoren* verknüpft. Dies ist eine Liste der zulässigen Operatoren:

Operator	**Erklärung**
+, -,*, /	Arithmetische Operatoren der Grundrechenarten
%	Modulo-Operator (ermittelt den ganzzahligen Rest einer ganzzahligen Division).
=	**Zuweisungsoperator**
+=, -=, *=, /=, %=	Verkürzende Schreibweise für Ausdrücke. (z. B.: a += b entspricht a = a+b)
<, <=, >, >=, = =, !=	**Vergleichsoperatoren** Aussage wahr (true) hat einen Wert ungleich Null Aussage falsch (false) hat den Wert 0
&&	logischer UND-Operator
||	logischer ODER-Operator
!	logischer NICHT-Operator (Negation)
&	bitweiser UND-Operator
|	bitweiser ODER-Operator
~	bitweise Negation
&variable	Referenzierungsoperator: evaluiert die Adresse der Variable
*pointer	Dereferenzierungsoperator: deklariert einen Variable vom Zeigertyp (Pointervariable) liefert den Wert eines Zeigers, der per Referenz (Adresse) übergeben wurde
(<Typ>) ausdruck	Cast-Operator: erzwingt die Umwandlung der Variablen zahl in den durch Typ angegebenen Datentyp (Gefährlich)
sizeof(<typ>)	Operator sizeof: liefert die Größe des Typs in Bytes

Die offensichtlichen Operatoren sind hierbei die vier binären Operatoren der Grundrechenarten. Es ist zu bemerken, dass C außer diesen und dem Potenz- und Modulooperator keine weiteren Rechenarten direkt unterstützt. Kompliziertere Funktionen werden in Bibliotheken ausgelagert.

Die Verknüpfung zweier Ausdrücke mit einem binären Operator ergibt einen Ausdruck, der einem *lvalue* zugewiesen werden kann. Der hinlänglich bekannte Pythagoras würde sagen:

```
a^2 + b^2 = c^2;
```

Daran kann man erkennen, dass Pythagoras nicht C programmieren konnte. Die Ausdrücke „a^2“ und „b^2“ evaluieren wohl zu einem Ausdruck, der dem jeweiligen Wert der Variablen „a“ und „b“ zum Quadrat entspricht, auch die Summe dieser beiden Teiausdrücke ergibt wieder einen gültigen Ausdruck. Dieser kann allerdings nicht einem weiteren Ausdruck „c^2“ zugewiesen werden. Als Ziel einer Zuweisung ist lediglich eine Variable zulässig.

Außerdem erfolgen Zuweisungen in C in der mathematisch üblichen Richtung von *rechts nach links* (so wie der Koran gelesen wird). Der Satz mußte deshalb lauten:

```
c_quadrat = a^2 + b^2;
```

Womit uns Pythagoras allerdings auf dem entscheidenden Problem sitzen lässt, wie den nun aus der Variablen „c_quadrat“noch eine Quadratwurzel gezogen werden kann. Wie so oft tut sich auch hier mit der Lösung eines kleinen Problems ein größeres auf.

14.Rudelzuweisung mit Ausdruck

Im eben gezeigten Beispiel tritt eine *Zuweisung* auf. Diese Zuweisung ist einerseits eine gewöhnliche *Anweisung*, andererseits aber auch ein *Ausdruck,* der zum zugewiesenen Wert evaluiert. Beispiel:

```
nVariable = 777 – 111 ;
```

ist eine Zuweisung, die der Variablen nVariable den Wert 666 zuweist. Außerdem ist dies aber auch ein Ausdruck, der zu diesem Wert evaluiert. In klassisch mathematischer Lesart kann deshalb auch eine Konstruktion wie

```
nHans = nFrans = nZepp = 0;
```

verwendet werden. Zuerst wird nZepp auf 0 gesetzt. Der Ausdruckswert dieser Zuweisung (also 0) wird nFrans zugewiesen und der Ausdruckswert dieser Zuweisung (also wieder 0) schließlich nHans zugewiesen.

15.Ausdrucksvolle Zuweisung mit Vergleich

Die Ausdruckseigenschaft der Zuweisung eröffnet eine völlig neue Möglichkeit zur Erzeugung unverständlicher und fehleranfälliger Programme. Genau deshalb wird diese Eigenschaft weidlich genutzt, wie beispielsweise bei den Erfindern von C, Kernigham & Ritchie exerziert:

```
while ((c = getch()) != 27)
```

Hier wird erst der Rückgabewert der *getchar()*-Funktion der Variablen *c* zugewiesen, anschließend der Ausdruckswert mit einer *Literalkonstanten* (27) verglichen und als *Laufbedingung* für eine *while()* Schleife verwendet.

Zum Nachdenken: Was könnte im Gegensatz dazu

```
while (c = getch() != 27)
```

bedeuten? Dabei ist die Priorität der Operatoren zu beachten (nach dem Motto „Punkt vor Strich“). In C gilt folgende Liste (1 ist die höchste Priorität)

Priorität	Operatoren
1	() [] -> . :: ! ~ ++ --
2	- (unär) * (Dereferenzierung) & (Addressoperator) sizeof
3	->xxx .xxx
4	* (Multiplikation) / %
5	+ -
6	<< >>
7	< <= > >=
8	== !=
9	&
10	^
11	|
12	&&
13	||
14	? :
15	= += -= etc.
16	,

Gefährlich wird das ganze auch, weil aus einem Vergleich auf Gleichheit (a = = 123) durch ein kleines Missgeschick leicht eine Zuweisung (a = 123) werden kann. Aus diesem Grund bevorzugen erfahrene C-Programmierer, die äquivalente Vergleichsnotation (123 = = a), da in diesem Fall die Zuweisung (123 = a) nicht gültig ist.

Satz: *„In C wird aus einem verunglückten Vergleich (= =) sehr leicht eine Zuweisung (=) mit sehr unangenehmen Effekten. Hier ist große Sorgfalt anzuwenden.“*

Und noch ein

Satz: *„Die Verwendung von Zuweisung und Vergleich in **einem** Ausdruck ist gefährlich und sollte nur von erfahrenen Programmierern (und mit vollständiger Klammerung) erfolgen.“*

16.Wir spielen Boole

Eine besonders wichtige Stellung nehmen die booleschen Operatoren und die booleschen Ausdrücke ein, weil sie unabdingbar für *Bedingungen* und *Iterationen* sind. Die *Vergleichsoperatoren* liefern einen Ausdruck des Typs *bool* und können über entsprechende Operatoren weiter verknüpft werden. Beispiel:

```
(nAlter > 18 && nAlter < 85) || nKontoStand > 100000
```

könnte dazu dienen, den Zugang zu *adult content* derart zu beschränken, dass keine rechtlichen Konsequenzen für den Anbieter zu befürchten sind.

Achtung: *Die booleschen Operatoren (&&, ||, !) müssen unbedingt von den bitweisen logischen Operatoren (&, |, ~) unterschieden werden.*

17.Eingeben!

Die unsägliche *printf()*-Funktion hat auch eine hässliche Schwester, die (orthogonal dazu) der Eingabe von Werten dient und den nicht minder einprägsamen Namen *scanf()* (scan with format) trägt.

Auch scanf() arbeitet mit einem Formatstring mit *Platzhaltern*. Immerhin lassen sich die bei printf() gewonnenen Erkenntnisse bezüglich des Formatstrings auf diese Funktion übertragen. Die Eingabe einer Dezimalzahl erfolgt beispielsweise via scanf(“%d“,&a);. Beispielcode:

```
int a = 12345;
printf ("Eingeben Sie ein a:");
scanf  ("%d",&a);
printf ("Sie haben %d eingegeben\n",a);
```

Wichtig ist hierbei der unverzichtbare *Adressoperator* vor dem Variablennamen. Da die Funktion scanf() die *Inhalte* der verwendeten Variablen verändern muss, ist es erforderlich, die *Adresse* der Variablen anstatt des Wertes zu übergeben. Dies wird als Referenzübergabe bezeichnet.

Satz: *„Das Motto der Referenzübergabe lautet: »Sag mir nicht, was jetzt drin steht, sag mir einfach, wo ich es hinschreiben soll.«*

Achtung: *Wenn die Eingabe nicht zum Formatstring passt, wird einerseits die Variablenzuweisung **gar nicht** durchgeführt, andererseits bleibt auch der Eingabepuffer erhalten.*

18.Entscheiden, jetzt!

Der Ablauf eines Programmes findet stets „von oben nach unten“, also in der programmierten Reihenfolge der einzelnen Abweisungen statt. Das ist auf Dauer sehr langweilig und ineffizient.

Eine spannende Möglichkeit, in den Programmablauf einzugreifen, stellen Kontrollstrukturen dar. Die einfachste Kontrollstruktur ist die *if-Bedingung*.

Wichtig: *„Die Verwendung des Begriffs „if-Schleife“ wird mit schwerer Verspottung nicht unter drei Monaten bestraft.“*

Die Syntax der if-Bedingung sieht so aus:

```
if(<boolescher Ausdruck>)
      Anweisung
[else
      Anweisung]
```

Die Anweisung, die nach *if()* folgt, wird nur ausgeführt, wenn der Ausdruck in der Klammer zum Wert *true* evaluiert. Die Anweisung nach dem Optionalen *else* wird alternativ dazu nur ausgeführt, wenn der Ausdruck in der Klammer zum Wert *false* evaluiert.

Mehrere Anweisungen können jederzeit durch eine beliebige Menge öffnender { und schließender } geschwungener Klammern zu einer Anweisung gruppiert werden, womit sich der Entscheidungsbereich der *if*-Bedingung auf beliebig viele Anweisungen erweitert.

19.Eine coole Entscheidung

Die Sprache C kennt auch eine sehr elitäre Formulierung der Bedingung auf Ausdrucksbasis. Dies ist der *conditional operator* „?“ Die Syntax des conditional operators sieht so aus:

```
<boolescher Ausdruck> ?  <ausdruck bei true> : <ausdruck bei false>
```

Diese Konstruktion wird vorwiegend eingesetzt, um boolesche Entscheidungen in anderswertige Ausdrücke zu transformieren: Der Gesamtausdruck evaluiert - abhängig vom Wert des Ausdrucks vor dem Fragezeichen zu entweder dem Wert *vor* (true) oder *hinter* (false) dem Doppelpunkt. Beispiel:

```
return NULL != pTest ? OK : ERR_NOMEM;
```

20.Entweder – oder - oder ganz anders

Gelegentlich ist eine binäre Entscheidung nicht *sophisticated* genug. Dann muss eine mehrfache Fallunterscheidung her. In C kann so etwas mit der *switch()*-Konstruktion gelöst werden.

Die Syntax der switch()-Konstruktion sieht so aus:

```
switch(<ganzzahliger Ausdruck>)
      {
      case <Konstanter Ausdruck>:
            [Anweisung]
      [case <Konstanter Ausdruck>:
            [Anweisung]]
      [default:
            [Anweisung]]
      }
```

Mit dem Eintritt in den switch()-Kopf wird der Ausdruck in den Klammern evaluiert. Anschliessend wird in der Liste der *case*-Konstanten nach diesem Wert gesucht. Wird er gefunden, so wird die Programmausführung dort fortgesetzt, wird er nicht gefunden, so setzt sich das Programm entweder an der Stelle *default* fort, wenn diese optionale Marke nicht existiert, nach der schließenden geschwungenen Klammer.

Achtung: *Im Fall, dass ein case gefunden wird, setzt sich die Programmausführung mit der nächsten Anweisung nach diesem case fort. Sie* ***endet aber nicht*** *mit dem nächsten case. Um diesen Effekt zu erreichen, ist ein zusätzliches abschließendes* ***break*** *nötig.*

21.Auf ein Neues!

Wiederholung prägt bekanntlich ein. Außerdem ist mit einem streng linearen Programmablauf auf Dauer kein Blumentopf zu gewinnen. Die *Iteration* bzw. *Schleife* ist eine der wichtigsten Kontrollstrukturen jeder Programmiersprache.

Die einfachste Form der Schleife ist die Forever-Schleife, eine Endlosschleife: Gemäß der Notation:

```
for(;;)
  Anweisung  //Schleifenrumpf
```

Wird die folgende Anweisung unendlich (naja...) oft wiederholt. Auch hier können natürlich mehrere Anweisungen über geschwungene Klammern gruppiert werden. Die Schleife wird verlassen, wenn das Schlüsselwort *break* im Ablaufpfad auftaucht. Beispiel:

```
int nSec = 5;

for(;;)
  {
  printf("Noch %d Sekunden bis zum Start\n",nSec);
  nSec -= 1;
  if (0 == nSec) break;
  }
```

Diese Schleife endet mit der Sekunde Null durch einen break.

Satz: *„Hinter der Klammer einer for()-Schleife sollte nie ein Semikolon stehen“*

Weil Endlosschleifen an sich sehr wenig Sex-Appeal haben, bietet die for()-Schleife wesentlich leistungsfähigere Möglichkeiten. Die Beiden Semikolons innerhalb der runden Klammern teilen deren Inhalt in drei Bereiche.

```
for(<Initialisierung>;<Laufbedingung>;<Iteration>)
```

Der Inhalt des Feldes *Initialisierung* wird *einmalig* nur beim ersten Eintritt in die Schleife ausgewertet. Hier werden meist Zählervariablen initialisiert. Mehrere Ausdrücke dürfen durch *Kommata* getrennt werden.

Der Inhalt des Feldes *Laufbedingung* wird vor jeder Eintritt in den Schleifenrumpf zu einem booleschen Ausdruck evaluiert. Nur wenn dieser Ausdruck *true* ist, wird der Rumpf ausgeführt. Im gegenteiligen Fall setzt die Programmausführung hinter dem Schleifenrumpf fort.

Der Inhalt des Feldes *Iteration* wird immer am Ende des Schleifenrumpfes evaluiert. Hier steht üblicherweise ein Seiteneffekt, der die Zählervariablen aktualisiert.

Ein Beispiel:

```
int e,p;

for (e = 0, p = 1 ; e < 16 ; e++, p*=2)
  {
  printf("2 hoch %d = %d\n", e, p);
  }
```

Jedes der drei Felder kann auch leer sein (und wird dementsprechend nicht ausgewertet), im Falle einer leeren Laufbedingung entsteht eine Forever-Schleife.

Die for()-Schleife arbeitet *abweisend*, was bedeutet, dass der Schleifenrumpf, wenn die Laufbedingung bereits beim Eintritt in die Schleife false liefert, überhaupt nicht ausgeführt wird.

22.Langewhile

Eine weitere Form der Schleife stellt die *while()*-Schleife dar, die in zwei Geschmacksrichtungen auf den Markt kommt. Die abweisende while()-Schleife hat - analog zur for()-Schleife - nur ein Laufkriterium. Sämtliche Initialisierungen und Iterationen müssen hier ggf. gesondert erledigt werden. Diese Schleife kommt meist zum Einsatz, wenn Iteration und Initialisierung keiner strengen Systematik folgen (z.B. beim Verarbeiten von Benutzereingaben.). Die while()-Schleife stellt sich syntaktisch so dar:

```
while(<Laufbedingung>)
  Anweisung  //Schleifenrumpf
```

Die zweite Geschmacksrichtung der while()-Schleife ist die nicht abweisende *do-while();*-Schleife. Sie sieht so aus:

```
do
  Anweisung  //Schleifenrumpf
while(<Laufbedingung>);
```

Zwei Dinge sind hier sehr **wichtig**.

- Der Rumpf der do-while()-Schleife wird *mindestens ein mal* ausgeführt, bevor das Laufkriterium evaluiert wird.
- Die do-while()-Schleife ist die einzige Schleifenkonstruktion, in der hinter dem while() ein Semikolon stehen muss.

23.Style hat man, oder man codet ihn

Da der C-Compiler keinerlei Ansprüche an die ästhetische Qualität des Sourcecodes stellt, ist es eminent wichtig, durch *Selbstdisziplin* für eine durchgängige Versteh- und Lesbarkeit der Sourcen zu sorgen. Unter dem Begriff *Coding style* fasst man üblicherweise drei Themenbereiche zusammen:

- Nomenklatur: Die Benennungsrichtlinie für alle frei wählbaren Bezeichner (also vorwiegend Variablen- Konstanten- und Funktionsnamen, aber auch Filenamen). Außerdem sollll auch festgelegt werden, in welcher (natürlichen) Sprache Bezeichner, Module und Kommentare benannt werden (english, deutsch, esperanto, denglish...).
- Quelltextformatierung: Die Richtlinie zur Einrückung und Abteilung von Strukturblöcken (insbesondere der Klammersetzung). Hier steht der effektive Einsatz von *Whitespace* im Mittelpunkt.
- Modulorganisation: In größeren Projekten werden die Programmfunktionen auf mehrere Quelltexte aufgeteilt. Hier muß definiert werden, nach welchen Gesichtspunkten diese Aufteilung erfolgt und wie die Benennung der Files organisiert wird.

Ein guter Coding style zeichnet einen guten Programmierer aus, in den meisten Fällen wird der style aber vom Auftraggeber (Arbeitgeber) vorgeschrieben.

24.Mit Style oder ohne - Programmaufbau

Ein C-Quelltext kann so unleserlich aussehen, wie er will, er muss dennoch einigen formalen Kriterien genügen. Für die Applikationsentwicklung muß ein Programm folgende Kriterien erfüllen:

- Eine main() Funkrion muss vorhanden sein. Sie wird nach dem Laden des ausführbaren Programms aufgerufen. Die logische Ausnahme ist Windows: Dort heisst die Funktion WinMain().
- Die weiteren Kriterien sind in dem folgenden Beispiel kommentiert:

```
#include <stdio.h>
#include <stdlib.h>
#include "mystuff.h"
// Am Anfang werden alle nötigen Headerdateien inkludiert. (darf prinzipiell
überall stehen).

#define MAGIC_SIZE 255
#define MOGRIFY(_a) (~_a+1)
// Anschliessend werden Makros deklariert (darf jederzeit erfolgen)

char someFunc(char chWhatEver);
// Funktionsprototypen stehen vor dem ersten Funktionsrumpf. Verpflichtend vor
der ersten Referenzierung.

int nGlobal;
// die Deklaration globaler Variablen erflolgt ausserhalb aller
// Funktionen und vor dem ersten Funktionsrumpf

int main (int argc, char **argv)
// die main()-Funkrion muss in einer Applikation immer vorhanden sein
// und braucht keinen prototyp.
  {
  // die geschwungenen Klammern sind in Funktionen verpflichtend.

  char chTest = 0;
  // die Deklaration lokaler Variablen muss innerhalb dergeschwungenen
  // Klammern (Funktionsrumpf) und in ANSI-C vor der ersten Anweisung
  // der Funktion stehen.

  //jetzt beginnen die Anweisungen
  while ((chTest = getch()) != 27)
      {
      putch (someFunc(chTest));
      }
  return 0;
  }

// Hier folgen weitere Funktionen
char someFunc(char chWhatEver)
  {
  return chWhatEver & ~0x20;
  }
```

Vieles in der gezeigten Anordnung ist nicht Vorschrift, spricht abr für guten Coding Style. Zwingend sind:

- Die Deklaration von *globalen* Variablen muss ausserhalb aller Funktionsrümpfe erfolgen.
- Die Deklaration von *lokalen* Variablen muss innerhalb eines Funktionsrumpfes erfolgen.
- Die Deklaration von *lokalen* Variablen muss vor allen Anweisungen stehen.
- Funktionsprototypen müssen vor ihrer ersten Verwendung deklariert werden.

25.In Reih' und Glied ins Verderben marschier'n

Gewöhnliche skalare Variablen sind der ideale Datenspeicher für einzelne Informationen, versagen aber bei der Verarbeitung tabellarischer Daten, wie sie in Wissenschaft und Wirtschaft oft auftreten. Für derartige Daten wurde der *Arraytyp* geschaffen.

Die Arraydeklaration kann auf jeden Datentyp angewendet werden und ermöglicht den *indizierten* Zugriff auf ein benanntes Feld gleich typisierter Daten. Die Arrayeigenschaft wird in der Deklaration durch das Anhängen eckiger Klammern an den Variablennamen ausgedrückt. Innerhalb der eckigen Klammern steht die konstante Größe des Arrays. Folgende Beispieldeklaration:

```
short asNixSpezielles[365];
int   anMonatsUmsatz[12] = {111,122,100,122,123,200,175,95,80,100,121,152};

anMonatsUmsatz[0] = 100;
anMonatsUmsatz[1] = 125;
```

zeigt die Deklaration eines Arrays vom Typ *int*. Es enthält **12 Einträge**, die über *Indexausdrücke* von **0 bis 11** adressiert werden können. Das Array vom Typ short enthält 365 Einträge mit Indizes von 0 bis 364. Ihren großen Vorteil entfalten die Arrays mit variablen Indexausdrücken in einer Schleife. Um beim Beispiel zu bleiben:

```
int nSumme;

for (i=0, nSumme=0 ; i<12 ; i++)
      nSumme += anMonatsUmsatz[i];
```

Summiert in zwei Zeilen sämtliche Monatsumsätze. So weit zu den Vorteilen. Nun zu den Macken:

Satz: *„In C können Arrays nicht kumulativ zugewiesen, sondern nur elementweise kopiert werden*“

Diese Eigenschaft bedeutet für gewöhnliche Arrays allerdings nur einen kleinen Komfortverlust.

Satz: *„Der gültige Indexbereich eines Arrays der Größe n geht von **0 bis n-1**.*“

Aufgrund der *Performance-Orientierung* von C führen die Arraytypen in Summe zu weit mehr Problemen als sie zu lösen imstande sind. Ist die (bei Programmiersprachen allgemein übliche) Beschränkung der Indexwerte noch akzeptabel, so tut sich mit dem nächsten Satz der Abgrund der Hölle auf:

Satz: *„Die Gültigkeit des Index wird bei Arrayzugriffen **nicht geprüft***“

So harmlos dieser Satz klingt, so weitreichend sind die Konsequenzen. Jeder Arrayzugriff wird vom Compiler in eine einfache Adressberechnung aufgelöst:

```
Zugriffsadresse = Startadresse + index * sizeof(Arraytyp)
```

Da hierbei als Index jede ganze (also auch negative!) Zahl stehen kann, ist mit dieser Methode prinzipiell *jede Speicheradresse* erreichbar. Den Arrayzugriff mit einem unzulässigen Index wird allgemein als *Buffer Overflow* bezeichnet.

26.C-Strings oder „beyond apocalypse“

Wie eingangs bereits erwähnt, kennt C keinen generischen Typ zur Speicherung von Text. Dieser offensichtliche Mangel wurde in der standardmäßig zum System gehörenden Bibliothek *libc* „behoben“.

Text wird in C ASCII-codiert in einem Array von Typ char – dem so genannten C-String - gespeichert, womit dieser bereits alle vom Array her bekannten Probleme erbt. Im Code sieht das so aus:

```
char szText[128] = "Hallo C-String";
printf ("Im String steht: %s\n",stText);
strcpy (szText,"Und Tschuess...");
printf ("Im String steht jetzt: %s\n",stText);
```

Damit aber Texte *unterschiedlicher* Länge in einem Array *konstanter* Größe sinnvoll verarbeitet werden können, muss die Länge des Nutztextes zusätzlich dokumentiert sein. Dies geschieht im C-String durch die so genannte *Nullterminierung*. Am Ende des Textes steht eine binäre Null (der kein ASCII-Code zugeordnet ist).

Satz: *„Der Inhalt eines C-Strings endet mit dem ersten Auftreten einer binären Null“*

Während diese Konvention eigentlich ermöglichen sollte, kurze Texte in großen Arrays unterzubringen, führt sie zu erheblichen Effekten, wenn diese Null abhanden kommt. In diesem Fall endet der C-String erst mit der nächsten Null und die kann, weitgehend zufällig, auch erst weit jenseits des eigentlichen Arraybereichs im Speicher stehen. Damit erfolgt eine erhebliche Verlängerung des Inhaltes, der dann meist in der folgenden Verarbeitung einen Buffer Overflow auslöst.

Weil Arrays ja nicht direkt zugewiesen werden können, gibt es auch zur Zuweisung und Manipulation von Stringinhalten eigene Funktionen in der libc. Beispiel:

```
// Stringzuweisung
char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
// „Stringaddition“
char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src, size_t n);
```

Die Versionen, die mit mit *„strn“* beginnen, beschränken die Länge der Kopieraktion, um mögliche Bufffer Overflows zu verhindern. Sollte dabei allerdings die Begrenzung wirklich greifen, so wird auch die terminierende Null *nicht kopiert* und der String damit (wie erwähnt) auf eine unbekannte Länge vergrößert. Der Teufel wird dadurch mit dem Beelzebub ausgetrieben.

Der C-String ist derzeit das häufigste Einfallstor zur feindlichen Übernahme öffentlich erreichbarer Rechnersysteme durch so genannte *Buffer Overflow Exploits*.

27.Wohin zeigst Du?

Im Zusammenhang mit *scanf()* wurde bereits die Notwendigkeit des Adressoperators angedeutet. Der Operator „&“ evaluiert zur logischen Speicheradresse des folgenden Bezeichners. Dieser Bezeichner ist in den meisten Fällen eine variable, manchmal auch eine Funktion. Beispiel:

```
int nVariable

printf ("nVariable steht an der Speicheradresse %x\n",&nPointer);
```

Diese Adressen werden beispielsweise zur *Referenzübergabe* benötigt. Der Adressausdruck kann aber auch einer Variablen zugewiesen werden. Eine Variable, die eine Adresse beinhaltet bezeichnet man als *Pointer* (oder uncool als *Zeiger*).

Auch Pointer werden speziell deklariert, ähnlich wie bei Arrays haben auch sie einen Basistyp und einen Zusatz, der die Pointereigenschaft anzeigt. Die Deklarationen

```
int   *pnIntPointer;
float *pftFloatPointer;
```

deklarieren zwei Variablen von Typ *Pointer auf int* und *Pointer auf float*. Die Pointereigenschaft wird bei der Deklaration durch einen dem Namen vorangestellten Stern „*“ ausgedrückt.

Dem Stern begegnet man wenig später als *Dereferenzierungsoperator* wieder. Beispiel:

```
int  nVariable = 12345;
int *pnIntPointer;

pnPointer = &nVariable;
printf ("nVariable steht an der Speicheradresse %x\n",pnPointer);
printf ("In nVariable steht der Wert %d\n",*pnPointer);
```

Die letzte printf()-Anweisung ist interessant. Durch den *Dereferenzierungsoperator* kann *indirekt* lesend *und schreibend* auf den Wert einer Variablen zugegriffen werden, deren Adresse bekannt ist. Damit ist auch etwas wie eine *variable Variable* möglich. Beispiel:

```
int  nHinz = 12345;
int  nKunz = 54321;

pnPointer = &nHinz;
printf ("nHinz hat den Wert %d\n",*pnPointer);
pnPointer = &nKunz;
printf ("nKunz hat den Wert %d\n",*pnPointer);
```

Die identischen printf()-Anweisungen geben, abhängig vom Wert von pnPointer den wert zweier verschiedener Variablen aus.

Pointer und Arrays sind eng miteinander verwandte satanische Geschwister. Der Name einer Arrayvariablen ohne eckige Klammern evaluiert zu dessen Anfangsadresse und stellt somit einen Pointer auf das erste Element dar.

Satz: *Nicht initialisierte Pointervariablen enthalten – wie alle anderen nicht initialisierten Variablen – undefinierte Werte und zeigen dementsprechend auf eine beliebige Speicheradresse. Mit einem solchen* ***dangling pointer*** *lässt sich noch weit mehr Kollateralschaden anrichten als mit jedem buffer overflow.*

28.Kleine Häppchen erleichtern die Verdauung

Softwareprojekte mit mehreren Millionen Codezeilen lassen sich nur recht schwer in ein einziges main() quetschen. Deshalb unterstützt C, wie praktisch jede andere Programmiersprache auch, die Modularisierung von Programmen. Diese umfasst zwei wesentliche Aspekte:

- Modularisierung durch Gliederung in *Funktionen* und, darauf aufbauend,
- Modularisierung durch Verteilung des Codes auf mehrere Quelltextfiles.

Eine Funktion haben wir bereits kennen gelernt, nämlich die Hauptfunktion jeder C-Applikation mit dem Namen *main()*. Darüber hinaus kann eine beliebige Menge weiterer Funktionen deklariert und implementiert werden. Und das geht so:

Auch Funktionen sollen (nicht müssen) vor ihrer Verwendung *deklariert* werden. Die Deklaration einer Funktion nennt man auch *Prototyp.* Der Prototyp definiert

- den Namen
- den Rückgabewert (return value) und
- die Parameterliste

Eine Funktion muss einen Namen und *genau einen* Rückgabewert, der aber *void, (also nix)* sein kann, haben. Weiterhin ist eine beliebige Menge an Parametern (auch null) zulässig. Beispiel:

```
float VektorBetrag (float ftX, float ftY);
```

Dieser Prototyp deklariert eine Funktion, die zwei Parameter vom Typ float nimmt und einen float Wert zurückliefert. Zusätzlich zur Deklaration ist bei Funktionen natürlich auch noch eine *Implementierung* erforderlich. Ein suboptimales Beispiel:

```
float VektorBetrag (float ftX, float ftY)

  {
  float ftTemp; // Eine funktionslokale Variable

  ftTemp = sqrt(pow(ftX,2) + pow(ftY,2));
  return ftTemp;
  }
```

Die Funktion evaluiert bei ihrem Aufruf zum Ausdruck ihres Rückgabewertes. Beispiel:

```
float ftBetrag;

ftBetrag = VektorBetrag (4, 5);
```

Die Variable ftBetrag hat anschliessend ungefähr den Wert 6.4. Diese Funktion ist nun modulübergreifend nutzbar, insbesondere, wenn der Prototyp in ein Headerfile ausgelagert wird.

Viele solcher Funktionen können in ein Sourcefile gruppiert werden, womit ein Modul ensteht, das der Compiler in ein eigenes Objektfile compiliert. Erst der Linkerlauf integriert sämtliche Module in eine Applikation.

Ein solches Objektmodul kann auch zusammen mit einem passenden Headerfile, das die Prototypen deklariert, als *statische Bibliothek* (*static library)* weitergegeben werden, ggf. auch ohne den Quelltext selbst mitliefern zu müssen.

29.Und wer räumt bei den Daten auf?

Die Welt ist komplex und das erfordert entsprechende Gegenstücke auf der Softwareseite. Oftmals gehören mehrere verschiedene Daten zu einem Objekt der realen Welt. Ein Studierender besitzt beispielsweise einen Vor- und Nachnamen, ein Geburtsdatum und – als Wichtigstes – eine Matrikelnummer. Dies könnte man klassisch in Variablen so ausdrücken:

```
char  szVorname[NAME_LEN];
char  szNachname[NAME_LEN];
char  cGeburtstag;
char  cGeburtsmonat;
short sGeburtsjahr;
long  lMatrikelNummer;
```

Diese lose Datensammlung drückt allerdings nicht aus, dass dies alles in einem Zusammenhang steht. Eine solche logische Gruppierung erreicht man durch *Strukturierung* der Daten:

```
struct STUDIERENDER
  {
  char  szVorname[NAME_LEN];
  char  szNachname[NAME_LEN];
  char  cGeburtstag;
  char  cGeburtsmonat;
  short sGeburtsjahr;
  long  lMatrikelNummer;
  };
```

oder besser:

```
struct DATUM
  {
  char  cTag;
  char  cMonat;
  short sJahr;
  };

struct STUDIERENDER
  {
  char szVorname[NAME_LEN];
  char szNachname[NAME_LEN];
  struct DATUM Geburtsdatum;
  long  lMatrikelNummer;
  };
```

Aus dem letzten Block ist bereits zu erkennen, dass solche Strukturen wie eigene Typen funktionieren. Dementsprechend kann eine Variable dieses Strukturtyps deklariert werden:

```
struct STUDIERENDER Bucher, Jarz; //sehr altes Programm...
```

Auf die einzelnen Teile dieser Struktur kann über den Punktoperator zugegriffen werden.

```
printf("Student Bucher hat am %d.%d. Geburztag\n",
        Bucher.Geburtsdatum.cTag, Bucher.Geburtsdatum.cMonat);
```

Auch dieses Spiel kann beliebig weit getrieben werden. Da bei größeren Strukturen die Wertübergabe in eine Kopierorgie ausartet, wird häufig eine Referenzübergabe über Pointer auf Strukturen verwendet. In diesem Fall kann der Pointer im *ersten Schritt* direkt über den *Pfeil-Dereferenzierungsoperator* dereferenziert werden. Beispiel:

```
struct STUDIERENDER *pAktuellerDepp;
pAktuellerDepp = &Bucher;
Printf("Student Bucher hat die Nummer %d\n", pAktuellerDepp->lMatrikelNummer);
```

30.Typberatung für Individualisten

Neben den sattsam bekannten generischen Typen ermöglicht das Keyword *typedef* die Vereinbarung beliebiger eigener Typen. Dabei können auch strukturierte Typen mit einfachen Namen belegt werden:

```
typedef short INT16;
typedef long  INT32;
typedef struct
      {
      int nX;
      int nY;
      int nZ;
      } T_VEC_3D;

INT16     iVariable; // Deklaration
T_VEC_3D  ZielRichtung = {0,0,1};
```

Dies dient einerseits zur Verbesserung der Lesbarkeit, andererseits auch zur Verbesserung der Plattformunabhängigkeit von Code (durch Definition eigener Metatypen, Beispiel LINUX).

31.Überlass nichts dem System, was Du selbst noch schlechter kannst – Memory management

Arbeitsspeicher ist auf jedem Computer ein rares Gut und will deshalb sorgsam verwaltet werden. Deshalb ist das eine Aufgabe, an der ein C-Programmierer unbedingt selbst scheitern will. Dafür bietet die *libc* von ANSI C ungefähr zwei Funktionen:

```
void *malloc(size_t size);
void free(void *ptr);
```

In der Theorie ist der Umgang mit diesen Funktionen denkbar einfach. Mit einem simplen Aufruf von adresse=*malloc(<size>);* wird ein Speicherbereich der Größe <size> reserviert und deren Startadresse als *generischer pointer (void-Pointer)* zurückgeliefert. Wird der Speicherbereich nicht mehr gebraucht, so muss er mit einem Aufruf von *free(adresse)* wieder freigegeben werden. Beispiel:

```
struct STUDIERENDER *pAktuellerDepp;
pAktuellerDepp = malloc(sizeof(struct STUDIERENDER));
[...]
printf("Der Depp heisst %s %s\n",
        pAktuellerDepp->szVorname, pAktuellerDepp->szNachname);
```

Sollte der Aufruf von *free()* unterbleiben, so bleibt ein so genanntes S*peicherleck (memory leak)*, das mit fortdauernder Laufzeit des Programms immer mehr Arbeitsspeicher belegt und über kurz oder lang zum Absturz des Programms (oder such des Betriebssystems) wegen Speichermangel führt.

Richtige Programmiersprachen (wie Smalltalk oder Java) verwalten allozierten Speicher selbst und verfügen über einen *garbage collector*, der nicht mehr referenzierten Speicher automatisch recycled. Damit werden die meisten memory leaks sicher vermieden. Dennoch gibt es auch hier Daten, über deren Lebensdauer nur der Programmierer entscheiden kann und diese bleiben weiterhin anfällig.

Nur zur Relativierung eventueller spöttischer Kommentare: Praktisch jedes ernsthafte Softwareprojekt (Applikation, Service oder Betriebssystem) weist Speicherlecks auf. Die Kunst besteht vorwiegend darin, diese Lecks so weit zurückzudrängen, dass sie im Normalbetrieb während der üblichen Laufzeit nicht störend ins Gewicht fallen.

32.Locker von der Platte gehobelt

Der Wichtigtuer spricht von *Objektpersistenz*, wenn es darum geht, wichtige Daten unauslöschlich (*grins...*) auf dem Massenspeicher festzuhalten. Dazu benötigt man die Hilfe der File-Funktionen aus der libc. Die wesentlichen gepufferten Filefunktionen sind:

```
FILE *fopen(const char *path, const char *mode);
int fclose(FILE *stream);
int feof(FILE *stream);

int fgetc(FILE *stream);
int putc(int c, FILE *stream);

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
int fprintf(FILE *stream, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
```

Die Funktion *fopen()* öffnet das unter *path* angegebene File in dem als *mode* übergebenen Modus:

- „r“ Nur Lesezugriff, beginnend am Anfang des Files
- „r+“ Schreib- und Lesezugriff, beginnend am Anfang des Files
- „w“ File löschen und zum Schreiben freigeben
- „w+“ File löschen und zum Schreiben und Lesen freigeben
- „a“ File zum Schreiben öffnen, beginnend am Ende des Files
- „a+“ File zum Schreiben und Lesen öffnen, beginnend am Ende des Files
- „b“ Darf angehängt werden, um auch unter Windows Binärfiles (z.B. Bilder) zu verarbeiten.

Der Returnwert vom Typ *FILE** muss gespeichert werden, weil alle folgenden Funktionen ihn als Referenz auf das File benötigen.

Die Funktion *fclose()* schließt das unter *stream* angegebene File und gibt die FILE Struktur frei. **Besonders wichtig** ist *fclose()* nach **Schreibzugriffen**, weil die Änderungen meist erst mit dieser Funktion wirklich auf den Datenträger geschrieben werden.

Mit der *feof()* Funktion kann ermittelt werden, ob noch Daten im File vorhanden sind. Die Funktion liefert 0, wenn noch Daten in Leserichtung vorhanden sind

Die Funktionen *fgetc()* und *fputc()* lesen und schreiben jeweils genau einen char aus dem File bzw in das File.

Mit den Funktionen *fread()* und *fwrite()* können ganze Speicherblöcke aus dem File gesesen bzw. in das File geschrieben werden.

Die Funktionen *fscanf()* und *fprintf()* schliesslich erweitern die sattsam bekannten Funktionen *scanf()* und *printf()* auf die Interaktion mit Files.

33.Ich helf' mir selbst – und das immer wieder: Rekursion

Im Zuge der Erläuterung von Funktionen wurde bereits angedeutet, dass innerhalb jeder Funktion *jede* andere Funktion aufgerufen werden darf. Diese Möglichkeit geht sogar so weit, dass innerhalb einer Funktion diese Funktion *selbst* aufgerufen werden darf. In einem solchen Fall spricht man von eimen *rekursiven Aufruf*. Ein ebenso beliebtes wie sinnloses Beispiel ist die Berechnung der Fakultät einer Zahl.

Die Fakultät der Zahl n berechnet sich als das Produkt aller natürlichen Zahlen von 1 bis einschliesslich n. Wir haben diese Problematik bereits in Übungsaufgabe zwei mithilfe einer Schleife *iterativ* gelöst. Sinngemäß sah der Code der Funktion etwa so aus:

```
int fakultaet(int n)
  {
  int i, fac;

  for( i=1, fac=1; i<=n; i++)
      fac *= i;

  return fac;
  }
```

Um die Sache nicht so einfach erscheinen zu lassen, wie sie ist, haben Mathematiker auch einen rekursiven Ansatz entwickelt. Dieser definiert die Fakultät von n zu:

n! = n * (n-1)! Für n > 1 und 1! = 1

damit ergibt sich folgende rekursiv arbeitende Funktion:

```
int fakultaet(int n)
  {
  if (n <= 1)
      return 1;
  else
      return n * fakultaet(n-1);
  }
```

Das bestechende an dieser Funktion ist, dass sie ohne eine eigene Iteration auskommt. In vielen Fällen ist eine rekursive Implementierung schlanker und eleganter als eine iterative, sie muss aber mindestens eine Bedingung unbedingt einhalten:

Satz: *Der rekursive Selbstaufruf einer Funktion muss **immer** von einer eigenen Bedingung abhängen, damit sichergestellt ist, dass die Rekursion überhaupt **terminiert**.*

Im Fall unseres Beispiels wird die Fakultät von 9 so berechnet:

```
 fakultaet(9) =  9 * fakultaet(8)
  fakultaet(8) =  8 * fakultaet(7)
   fakultaet(7) =  7 * fakultaet(6)
    fakultaet(6) =  6 * fakultaet(5)
     fakultaet(5) =  5 * fakultaet(4)
      fakultaet(4) =  4 * fakultaet(3)
       fakultaet(3) =  3 * fakultaet(2)
        fakultaet(2) =  2 * fakultaet(1)
         fakultaet(1) =  1               // hier wird der if-Zweig gewählt
```

und jetzt wird der Aufrufstapel abgebaut

```
          1
         2*1
        3*2
       4*6
      5*24
     6*120
    7*720
   8*5040
  9*40320
 362880
```

Und das ist das erwartete Ergebnis. Zum Zeitpunkt der Ermittlung von fakultaet(1) hat sich die Funktion 8 mal selbst aufgerufen und keiner dieser Aufrufe hat ein Ergebnis gebracht. Erst wenn mit fakultaet(1) das konstante Ergebnis 1 zurückgegeben wird, kann der Aufruf fakultaet(2) mit dem Ergebnis 2 zurückkehren und daraufhin fakultaet(3) mit dem Ergebnis 6 und so weiter.

Die rekursive Lösung ist elegant, weist aber einen erheblichen Overhead auf. Für jede Rekursionsebene muss ein eigener Funktionsaufruf durchgeführt werden, der nicht unerhebliche Mengen an Rechenzeit und Stackspeicher verbraucht.

Beim Demontieren und Traversieren von listen und Bäumen wird uns die Rekursion dennoch als willkommenes Werkzeug zu Hilfe kommen.

34.Was Pointer treiben wenn es dunkel ist

Der Pointer kennt drei wichtige Anwendungen:

- Referenzübergabe
- dynamische Daten
- Verkettungen

Insbesondere die Verkettungsmöglichkeiten eröffnen neue Bereiche der Leistungsfähigkeit und Gemeinheit. Ausgangspunkt ist beispielsweise eine generische Knotenstruktur mit circa drei Pointern:

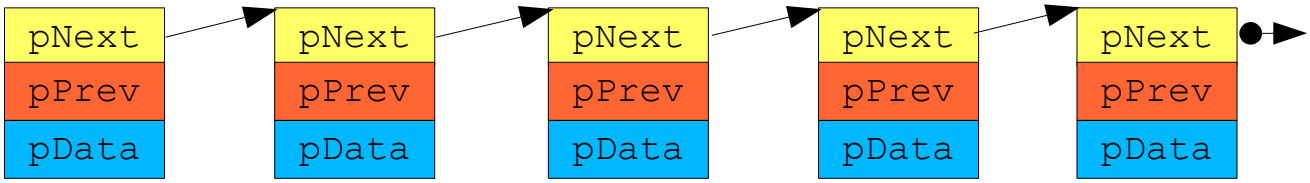
```
typedef struct NODE
  {
  struct NODE *pNext;
  struct NODE *pPrev;
  void        *pData;
  } T_NODE;
```

Mit einer größeren Menge dieser Knoten kann eine strukturierte Speicherung erheblicher Datenmengen variabler Größe erfolgen. Wie geht das? Ganz einfach! (Wer's glaubt...)

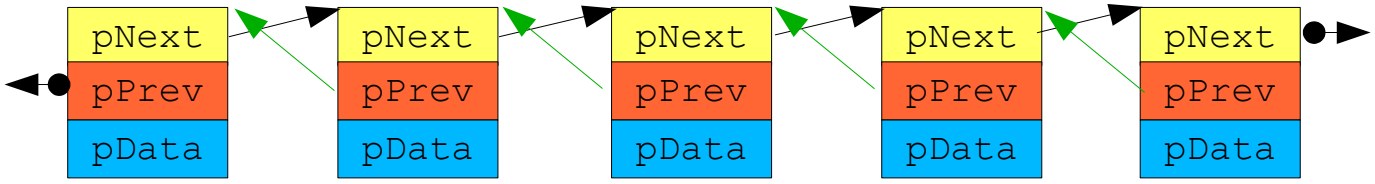
Der Knoten besitzt zwei Pointer auf weitere Knoten gleichen Typs, über die sich diese Knoten *verketten* lassen. Die Knoten selbst dienen nur zur Strukturierung der Daten, die Inhalte (also quasi die Werte) werden dynamisch (z.B. via *malloc()*) alloziert und über den generischen Pointer angebunden. Als Organisationsstruktur sind zwei Anordnungen üblich:

- Listen (einfach und doppelt verkettet) und
- Bäume

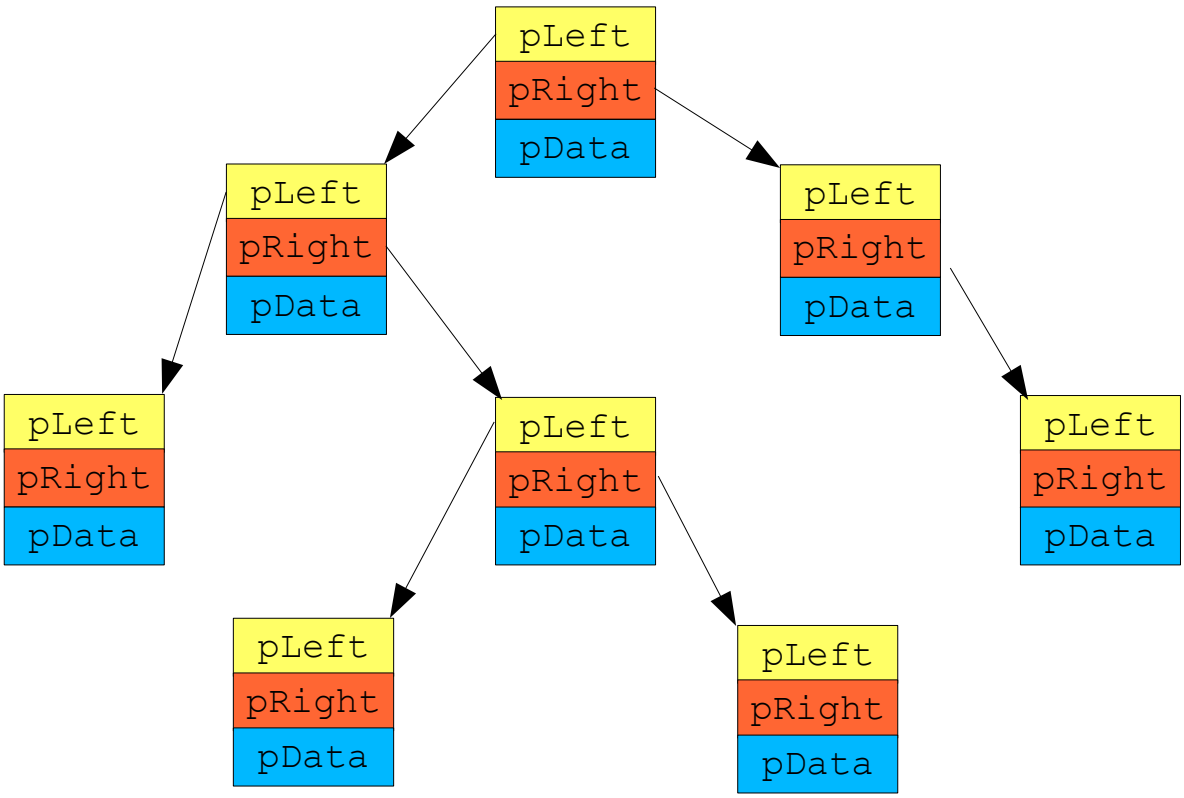
Eine einfach verkettete Liste verwendet den pPrev-Pointer nicht. Die Knoten werden in einer Reihe miteinander verlinkt. Das Ende der Liste wird entweder durch einen Nullpointer oder eine Selbstreferenz markiert. Diese Liste kann in aufsteigender Richtung durchlaufen (*traversiert) werden.*



Eine doppelt verkettete Liste verwendet den pPrev-Pointer um auf den vorherigen Knoten zu verlinken. Beide Enden der Liste werden entweder durch einen Nullpointer oder eine Selbstreferenz markiert. Diese Liste kann in beide Richtungen traversiert werden.



Der Binärbaum ist eine wesentlich interessantere Struktur, die beiden Pointer verweisen auf einen linken oder rechten Unterknoten. Mit Binärbäumen können besonders einfach und effizient sortierende Speicherstrukturen aufgebaut werden. Ein simpler *Binärbaum* könnte so aussehen:



Ein simpler *Birnbaum* dagegen sieht beispielsweise so aus:

Mehr zum aufregenden Paarungsverhalten der Binärbäume demnächst hier.

35.Die ich schuf die Liste werd' ich nun nicht los

Wie bereits angedeutet, kann eine verkettete Liste entweder iterativ oder rekursiv wieder aus dem Speicher entfernt werden. Zur iterativen Entfernung eignet sich ein Algorithmus der linearen Traversierung:

```
void freemem1(T_NODE *pRoot)
      {
      T_NODE *pTemp, *pNode;

      pTemp = pNode = pRoot;

      while (pNode)
            {
            pTemp = pNode->pNext;
            free(pNode);
            pNode = pTemp;
            }
      }
```

Zur vollständigen und sicheren Freigabe des gesamten Speichers sind *zwei* Hilfspointer erforderlich, da der pNext-Pointer des Blocks zur weiteren Verwendung gerettet werden muss, bevor der ihn enthaltende Block freigegeben werden kann.

Wesentlich eleganter sieht hingegen die rekursive Variante aus.

```
void freemem2(T_NODE *pNode)
      {
      if (pNode->pNext)
            {
            freemem2(pNode->pNext);
            }

      free(pNode);
      }
```

Diese Version kommt ohne Hilfsvariablen und Schleifenkonstrukte aus, benötigt aber für grosse Listen auch viel Rechenzeit und Stackspeicher.

36.Tausend Fallstricke für den ambitionierten Masochisten

Die Sprache C bietet in einem kompakten System mehr gemeingefährliche Fallen und Bosheiten als fünf richtige Programmiersprachen zusammen. Um dem angehenden C-Programmierer die Möglichkeit zu geben, in möglichst viele Fettnäpfchen zu treten, sei hier eine kleine Übersicht über die ewigen Klassiker gegeben.

Auf die einfachen Fälle soll nur ein kleiner Rückblick gegeben sein:

Beispiel 1: Die unerklärliche Endlosschleife:

```
int main (int argc, char **argv)
      {
      int nCount;
      int anArray[32];

      for (nCount=0; nCount<40; nCount++)
            {
            printf("Init anArray[%d]\n",nCount);
            anArray[nCount] = 0;
            }
      return 0;
      }
```

Aufgrund des buffer overflows wird dieses Programm auf den meisten Systemen nie terminiueren, da bei einer Bereichsüberschreitung des Arrays die *davor* deklarierten Variablen – in diesem Fall der Zähler selbst - getroffen und mit 0 belegt werden. Bei einer ungekehrten Deklarationsreihenfolge der Variablen würde der Stack und ggf. auch die dort liegende Rücksprungadresse überschrieben, was einen *Verlust der Programmkontrolle* bedeutet.

Beispiel 2: So wichtig kann eine Null sein

Da in C bekanntlich kein Stringtyp existiert, ist der ersatzweise gebastelte C-String nicht nur Ziel für Spott und Hohn, sondern auch für eigentümliche Fehler und Attacken. Die Arrayeigenschaft macht den C-String anfällig für buffer overflows, weshalb einige Funktionen erfunden wurden, welche mit Längenbegrenzung arbeiten – beispielsweise die Funktion *strncpy()*.

```
int main (int argc, char **argv)
      {
      int nVar=0xdeadbeef;
      char szString[10];

      strncpy(szString, "Das ist viel zu lang",10);
      printf ("szString enthaelt: <%s>\n", szString);
      return 0;
      }
```

Dieses Programm gibt etwas in dieser Art aus:

```
szString enthaelt: <Das ist viÿ¿c@pHÀ½>
```

Die wundersame Verlängerung des Strings entsteht aus dem Wegfall der terminierenden Null durch die so genannte *Längenbegrenzung*. In einen folgenden Verarbeitungsschritt wird dieser vermeintlich begrenzte String erst recht zu einem buffer overflow führen.

Beispiel 3: Dein char-pointer dangled!

Eine weitere Unart des C-Stings ist seine Zuweisungskompatibilität mit einem gewöhnlichen char-pointer:

```
int main (int argc, char **argv)
      {
      char *pcString;

      strncpy(pcString, "Wo soll das enden?",10);
      printf ("szString enthaelt: <%s>\n", pcString);
      return 0;
      }
```

Dieses Programm gibt etwas in dieser Art aus:

```
szString enthaelt: <Wo soll da>
Speicherzugriffsfehler
```

Dieses Programm führt meist zu einer Zugriffsverletzung, die das Betriebssystem veranlasst, das Programm umgehend zu beenden und aus dem Speicher zu werfen. Warum? Die Variable pcString ist ein so genannter *dangling pointer,* sie zeigt also auf eine beliebige Speicheradresse. Und genau dort hin werden die 10 Bytes des Strings geschrieben. Dass so etwas *meistens* nicht gut geht, liegt auf der Hand, die Auswirkungen können aber, abhängig vom Programmaufbau, durchaus subtil (z.B. spontane Veränderung von Variablen anderer Module) sein

.

Beispiel 4: Die Referenz der Verstorbenen

Ein subtiler Fehler ist die Rückgabe einer Referenz auf ein temporäres Objekt. Dabei wird als *return value* einer Funktion die Adresse einer *lokalen Variable* dieser Funktion verwendet. Diese lokale Variable existiert aber nach Ende des Programms nicht mehr.

```
char *func(void);

int main (int argc, char **argv)
      {
      char *pcString;

      pcString = func();
      printf ("szString enthaelt: <%s>\n", pcString);
      return 0;
      }

char *func(void)
  {
  char szTest[] = "Ein kleiner Test";

  return szTest;
  }
```

Dieses Programm gibt etwas in dieser Art aus:

```
szString enthaelt: <Ein ÿ±Cner Test>
```

Einige Compiler warnen in diesem offensichtlichen Fall zwar, der Code wird aber dennoch erzeugt. Der Stackspeicher, an dem der C-String szTest gespeichert war, steht nach dem Ende der Funktion func() wieder allgemein zur Verfügung, weshalb zum Zeitpunkt der Ausgabe der Inhalt dieses Speicherbereiches bereits teilweise überschrieben wurde.

37.Glossar mit Wachstumspotential

- Anweisung: Ein Ausdruck, gefolgt von einem Semikolon oder eine mir geschweiften Klammern gruppierte Folge von Ausdrücken. Ein Semikolon allein ist eine leere Anweisung.
- Assembler: Ein Programm, das aus einem Assemblertext ein Objektfile erzeugt.
- Ausdruck: Etwas, dem ein Wert zugeordnet (nicht zugewiesen!) werden kann.
- Bedingung: Eine Kontrollstruktur, die den Programmablauf in Abhängigkeit von einem booleschen Ausdruck in zwei verschiedene Pfade aufteilt.
- Call: Übergabe des Programmablaufs in eine Funktion. Das Programm wird nach dem Return mit der nächsten Anweisung nach dem call fortgesetzt.
- CamelCase: ZusammenSchreibung von Worten, bei der an jeder WortGrenze ein GroßBuchstabe steht. Häufig für VariablenNamen verwendet.
- Coding Style: Eine Vereinbarung über *Strukturierung* und *Nomenklatur* der Programmtexte, die zu einer besseren Wartbarkeit des Codes führen soll.
- Compiler: Ein Programm, das aus einem Hochsprachen-Programmquelltext ein Objektfile erzeugt. Oft auch synonym für ein Paket aus Compiler, Assembler und Linker verwendet.
- Editor: Werkzeug zum Erstellen und Verändern von Texten
- Kontrollstruktur: Ein Konstrukt der Programmiersprache, das die Ablaufreihenfolge der Anweisungen abweichend vom streng linearen Ablauf gestattet.
- Linker: Ein Programm, das aus mehreren Objektfiles und Bibliotheken durch statisches Binden ein ausführbares Programm erzeugt.
- Literal: Ein Ausdruck, der direkt einen konstanten Wert repräsnetiert. z.B.: 7 oder “Text“
- Makro: Ein Text, der durch den Präprozessor expandiert wird
- Objektfile: Eine Datei, die Anweisungen in Maschinensprache und Symbolinformationen enthält.
- Overflow: Überschreitung des gültigen Wertebereichs. Kann entweder ein numerischer Overflow sein, der zu unsinnigen Rechenergebnissen führt, oder ein Buffer Overflow, der auf nicht dafür zugewiesenen Speicher zugreift.
-
- Preprocessor: Ein Programmteil des Compilers, der vor der Übersetzung noch Ersetzungen im Quelltext vornimmt.
- Quelltext: Eine Datei, die in Textform ein Programm einer höheren Programmiersprache enthält.
- Return: Ein Operator zur Rückkehr an die aufrufende Funktion. Ein return aus main() beendet die Applikation.
- Return value
- Schleife: Eine Kontrollstruktur, welche die Anweisung(en) des Schleifenrumpfes so lange wiederholt, wie die Laufbedingung gültig ist.
- Typ: Eigenschaft einer Variablen oder eines Ausdrucks, die angibt, welchen Informationsgehalt und Speicherbedarf der Wert aufweist.
- Whitespace: Der wichtigste Inhalt eines Quelltextes, alle nicht sichtbaren Zeichen (Space, Tab, Newline), die zur Strukturierung dienen.
- Zuweisung: Die Belegung einer Variablen mit dem Wert eines Ausdrucks.