Die Programmiersprache C

1. Überblick und Einführung

Vergleich, Motivation, Historie, Technologie

Prof. Dr. sc. Joachim Fischer (Dr. Klaus Ahrens)
Institut für Informatik, Humboldt-Universität zu Berlin
SS 2014

Java versus C

Java

C

Java versus C

- **Java** ist ein experimentelles Fahrzeug auf Luftkissenbasis. Es bewegt sich auf Straßen aller Art, ist allerdings noch schwer zu steuern. Es ist strengstens verboten, Umbauten am Fahrzeug vorzunehmen.

- **C** ist ein offener Geländewagen. Kommt durch jeden Matsch und Schlamm, aber der Fahrer sieht hinterher auch dementsprechend aus.

Eigenschaften von C

Einige Eigenschaften

- kompakte Sprache – Reduktion auf das “Wesentliche”
- extensive Nutzung von Prozeduraufrufen/Funktionen
- schwaches Typ-Konzept (im Gegensatz zu Java, PASCAL)

- aber: strukturierte Programmiersprache
- *low-level*: Bit-orientierte Programmierung ist möglich

- Zeiger (engl. Pointer) Implementation: extensive Nutzung von Zeigern für Speicher, Arrays, Strukturen und Funktionen (später)
- effizienter Code kann erzeugt werden
- Portabilität (durch Sprachstandard): es kann Code für verschiedene Rechner erzeugt werden, aber breite „Grauzone“

Die Stärken von Java

- einfach
- objektorientiert
  Simula-67(1970), Smalltalk(1975), C++(1987), Eiffel(1988)
- architekturunabhängig, interpretativ
  virtuelle Maschine mit Bytecode als Interpreter: P-Code (Pascal, 1974), S-Code(Simula, 1978), M-Code(Modula, 1982)
- getrennte Übersetzung
  Modula-2 (1982)
  C, C++ nur textuelle Inklusion von Header-Files
- typsicher (zuweisungs- und ausdruckskompatibel)
  aber keine Pointer und Adressen (Maschinenorientierung)
  Indextest bei Feldern, "Garbage Collection" von Daten-Objekten

Schwächen von C

- aus softwaretechnischer und sprachtheoretischer Sicht ein gewisser Rückschritt (1978)
- einige unsaubere (unregelmäßige) Sprachkonzepte
- unsichere Sprache (es wird weniger geprüft als in Java), nur unabhängige statt separate Compilation (damit fehleranfälliger)
- Beispiel: **keine** implizite Initialisierung lokaler Variablen

Stärken von C

- flexibel
- Präprozessor (Makros, Include-Files, bedingte Compilation)
- Maschinennähe (*low-level*-Konzepte), damit hohe Laufzeiteffizienz erreichbar
- Betriebssysteme sind (häufig) in C programmiert
- Objektorientierte Erweiterungen existieren (mit Beibehaltung der Stärkung und Verringerung der Schwächen von C)
- *Zero-Overhead*-Prinzip: „Sie zahlen nicht für Dienstleistungen, die Sie nicht in Anspruch nehmen“ ☺

Ein praktischer Vergleich

- Wie gut eignen sich beide Sprachen „große" Probleme „schnell" zu lösen?
- Musterszenario: „Viele" Zeichenketten (im Speicher) lexikographisch sortieren.
- Hier: zufällig erzeugte Strings (aus kleinen Buchstaben) werden (*in situ* mittels Quicksort) sortiert.
- Aufruf:
  - **`[time] java s N`**
  - **`[time] s N`**

Eine Lösung in Java

```
public class s {
  static java.util.Random rand = new java.util.Random();

  static public void main(String[]s)
  {
      if (s.length < 1) return;

      int N = Integer.parseInt(s[0]);

      String[] all = new String[N];

      fill (all, N);

      java.util.Arrays.sort (all);
      //out (all, N);
  }
```

Eine Lösung in Java

```
   static private void fill (String[] v, int n) {
       for (int i=0; i<n; ++i)
             v[i] = randomString();
   }

   static private String randomString() {
       char[] chars = new char [10];

       for (int i=0; i<length; ++i)
             chars [i] = (char)('a'+ rand.nextInt(26));
       return new String (chars);
   }

   static private void out(String[] v, int n) {
       for (int i=0; i<n; ++i)
             System.out.println(v[i]);
   }
}
```

Eine Lösung in C

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define LENGTH 10

void  fill(char**, int);
char* randomString();
void  out(char**, int);
int   compare(const void*, const void*);

int main(int argc, char** argv) {
   int N;
   char** all;
   if (argc < 2) return 0;

   N = atoi(argv[1]);
   all = (char**)malloc(N * sizeof(char*));

   fill (all, N);

   qsort (all, N, sizeof(char*), compare);
   /* out (all, N); */
   return 0;
}
```

Eine Lösung in C

```
void fill (char** v, int n) {
    int i;
    for (i=0; i<n; ++i)
          v[i]=randomString();
}

char* randomString() {
    char* chars = (char*)malloc(LENGTH + 1);
    int i;
    for (i=0; i<LENGTH; ++i)
          chars[i] = ('a' + rand() % 26);
    chars[LENGTH] = 0;

    return chars;
}

void out(char** v, int n) {
    int i;
    for (i=0; i<n; ++i)
          printf(“%p: %s\n“, v[i], v[i]);
}

int compare(const void* x, const void* y) {
    return strcmp(*(char**)x, *(char**)y);
}
```

Vergleich

- Starke strukturelle Ähnlichkeit

- Unterschiede in:
  - Einbettung in die (Bibliotheks-) Umgebung
  - Behandlung von Programmparametern
  - Funktionskontext (Klassen-lokal vs. global)
  - Notwendigkeit von Prototypen (Vorab-Deklarationen)
  - Typ von Zeichenketten
  - Speicherverwaltung

  - Aber vor allem in den Laufzeiteigenschaften !!!

Vergleich der Laufzeiten

50
37,5
25
12,5
0

C (utime) Sekunden
Java (utime) Sekunden

N

1e+06
2e+06
3e+06
4e+06
5e+06

Geschichte der Sprache C

Markante Ereignisse

- UNIX wurde etwa 1969 entwickelt (auf einer DEC PDP-7 in Assembler)
- BCPL – eine Sprache mit mächtigen Entwicklungswerkzeugen
  - Erfahrung: Assembler als Entwicklungssprache zu "langatmig" und fehleranfällig
- Sprache "B" als weiterer Anlauf um 1970
- dritter Anlauf: neue Sprache mit neuen Ansätzen (geprägt durch Pascal/Algol): C als Nachfolger von B (um 1971)
- bis 1973 wurde UNIX fast vollständig in C umgeschrieben
- Ur-C: Kernighan & Ritchie
- heute aktuell ISO(ANSI)-C==C89, C99, C11

Geschichte der Sprache C

COMPUTERWORLD 1 April 1991
CREATORS ADMIT UNIX, C HOAX

In an announcement that has stunned the computer industry, Ken Thompson, Dennis Ritchie and Brian Kernighan admitted that the Unix operating system and C programming language created by them is an elaborate April Fools prank kept alive for over 20 years. Speaking at the recent UnixWorld Software Development Forum, Thompson revealed the following:

"In 1969, AT&T had just terminated their work with the GE/Honeywell/AT&T Multics project. Brian and I had just started working with an early release of Pascal from Professor Nichlaus Wirth's ETH labs in Switzerland and we were impressed with its elegant simplicity and power. Dennis had just finished reading 'Bored of the Rings', a hilarious National Lampoon parody of the great Tolkien 'Lord of the Rings' trilogy. As a lark, we decided to do parodies of the Multics environment and Pascal. Dennis and I were responsible for the operating environment. We looked at Multics and designed the new system to be as complex and cryptic as possible to maximize casual users' frustration levels, calling it Unix as a parody of Multics, as well as other more risque allusions. Then Dennis and Brian worked on a truly warped version of Pascal, called 'A'. When we found others were actually trying to create real programs with A, we quickly added additional cryptic features and evolved into B, BCPL and finally C. **We stopped when we got a clean compile on the following syntax:**

```
for(;P("\n"),R-;P("|"))for(e=C;e-;P("_"+(*u++/8)%2))P("| "+(*u/4)%2);
```

...

Web-Ressourcen

Können wie üblich wandern / verschwinden ☺

Eine Sammlung nützlicher Fragen mit Antworten:

http://www.eskimo.com/~scs/C-faq/top.html

Wertvolle Hinweise zum Programmierstil:

http://www.jetcafe.org/~jim/c-style.html

Eine (inoffizielle, weil Vorab-) Version des C(99)-Standards (in einem File):

http://busybox.net/~landley/c99-draft.html

Eine Sammlung von weiteren Links:

http://www.lysator.liu.se/c/

Coding Style

- In C lassen sich viel einfacher schlechte (unübersichtliche, schwer verständliche, schwer wartbare, nicht portable, ...) Programme schreiben ☹
- `ioccc.org` – The International Obfuscated C Code Contest

```
float o=0.075,h=1.5,T,r,O,l,I;int _,L=80,s=3200;main(){for(;s%L||
(h-=o,T= -2),s;4 -(r=O*O)<(l=I*I)|++ _==L&&write(1,(--s%L?_<L?--_
%6:6:7)+"World! \n",1)&&(O=I=l=_=r=0,T+=o /2))O=I*2*O+h,I=l+T-r;}
```

Coding Style

```
#include <stdio.h>
#include <math.h>
#define E return
#define S for
char*J="LJFFF%7544x^H^XXHZZXHZ]]2#( #@@DA#(.@@%(0CAaIqDCI$IDEH%P@T@qL%PEaIpBJCA\
I%KBPBEP%CBPEaIqBAI%CAaIqBqDAI%U@PE%AAaIqBcDAI%ACaIaCqDCI%(aHCcIpBBH%E@aIqBAI%A\
AaIqB%AAaIqBEH%AAPBaIqB%PCDHxL%H@hIcBBI%E@qJBH#C@@D%aIBI@D%E@QB2P#E@'C@qJBHqJBH\
%C@qJBH%AAaIqBAI%C@cJ%"   "cJ"   "CH%C@qJ%aIqB1I%PCDI`I%BAaICH%KH+@'JH+@KP*@%S@\
3P%H@ABhIaBBI%P@S@PC#",     *j   ,*e;typedef float x;x U(x a){E a<0?0:a>1?1:a; }
typedef struct{x c,a,t;      }   y;y W={1,1,1},Z={0,0,0},B[99],P,C,M,N,K,p,s,d,h
;y G(x t,x a,x c){K.c=t   ;      K.t=c; K.a=a;E K;}int T=-1,b=0,r,F=-111,(*m)(i\
nt)=putchar,X=40,z=5,o,   a,     c,t=0   ,n,R;y A(y a,y b,x c){E G(a.c+b.c*c,a.a
+c*b.a,b.t*c+a.t);}x H=   .5,Y   =.66     ,I,l=0,q,w,u,i,g;x O(y a,y b){E q=a.t*
b.t+b.c*a.c+a.a*b.a;}x Q(){E A(P,M,T       ),O(K,K)<I?C=M,I=q:0;}y V(y a){E A(Z,
a,pow(O(a,a),-H));}x D(y p){S(I=X,P         =p,b=T; M=B[++b],p=B[M.c+=8.8-l*.45,
++b],b<=r;Q())M=p.t?q =M_PI*H,w=atan2(   P.a-M.a,P.c-M.c)  /q,o=p.c-2,a=p.a+1,t=
o+a,w=q*(w>t+H*a?o:    w>t?t:w<o-H*a?t   :w<o?o:w),A(      M,G(cos(w),sin(w),0),
1):A(M,p,U(O(A(P,M,T)    ,p)/O(p,p)));   M=P;M.a=-         .9;o=P.c/8+8;o^=a=P.t
/8+8; M=Q        ()?o&1    ?G(Y,0,0):W   :G(Y,Y,1);E       sqrt        (I)-.45;}
int main(   int    L,char    **k){ S(e   =L>1?1[z=    0,   k]:J    ;*e    &&l<24 ;
++e)S(o=a   =0,j   =J+9;(c=    *++j)&&   !(o&&c<    X&&(q=l+=w)   );o    ?o=*j++/
32,b++[B]   =G(q   +=*j/8&3,*    j&7,0   ),B[r    =b++]=G((c/8&         3)*( o<2?
T:1), (c&   7)+    1e-4,o>2),1:    (o    =(a    =(c-=X)<0?w=c+6   ,t=    a+1:c?(t
?0:m(c),a        ):*++j)==((*e|32        )    ^z)&&1[j]-X));S(z   =3*(   L<3);++
F<110;)S(L=-301;p=Z,++L<300;m(               p.c),m(p.a),m(p.t))S(c=T;++c<=z;)S(h
=G(-4,4.6,29),d=V(A(A(A(Z,V(G(5,0           ,2)),L+L+c/2),V(G(2,-73,0)),F+F+c%2),G
(30.75,-6,-75),20)),g=R=255-(n=z)*64;    R*n+R;g*=H){S(u=i=R=0;!R&&94>(u+=i=D(h=
A(h,d,i)));R=i<.01);S(N=V(A(P,C,                 T)),q=d.t*d.t,s=M,u=1;++i<6*R;u-=
U(i/3-D(A(h,N,i/3)))/pow(                               2,i));s=R?i=pow(U(O(N,V(A(
M=V(G(T,1,2)),d,T))))                                  ,X),p=A(p,W,g*i),u*=U(
O(N,M))*H*Y+Y,g*=                                          n--?Y-Y*i:1-i,s:G(
q,q,1); p=A(p,s                                             ,g*u);h=A(h,N,.1
);d=A(d,N,-2*O                                                  (d,N));}E 0;}
```

IOCCC 2011 winner

Matt Zucker

http://www.ioccc.org/2011/zucker/hint.html

`$ ./zucker > image.ppm`

Coding Style

Gerade deshalb ist es umso wichtiger, gute (übersichtliche, leicht verständliche, gut wartbare, portable, ...) Programme zu schreiben

Wie?

- Übertragen Sie ihren Java-Style auf C!
- Divide et impera! (kleine Funktionen, kleine Quelltexte)
- Formatieren/strukturieren/kommentieren Sie lesbar!
- Nehmen Sie Warnungen ernst!
- Vermeiden Sie Hacks!
- Nutzen Sie verschiedene Compiler auf verschiedenen Plattformen
- Testen Sie abschließend (und nicht zu spät) auf der Referenzplattform

Compilertechnologie

C mit Präprozessor-Anweisungen → C-Präprozessor → C ohne Präprozessor-Anweisungen

- Einfügen von Dateien

```
#include <stdio.h>        (Systemdatei)
#include "stack.h"        (nutzerdefiniert)
```

- Ersetzen von Text(Makros)

```
#define L 5                       (Konstantendefinition)
#define add100(x)  ((x)+100)      (parametrisiert)
```

- bedingte Übersetzung

```
#ifdef TEST
            printf("Testversion");
#else
            printf("Produktionsversion");
#endif
```

C ohne Präprozessor-Anweisungen → Compiler → Assembler → Linker → Maschinen-programm

Compilertechnologie

C mit Präprozessor-Anweisungen → C-Präprozessor → C ohne Präprozessor-Anweisungen

- Einfügen von Dateien
- Ersetzen von Text(Makros)
- bedingte Übersetzung

```
#include <st…                     (Systemdate…
#inc…de "stac…                    (nutzerdefin…

#de…              5               (Konstan… finition)
#define add100(x) …  +100)        (param… rt)

…ef TEST
         pri… (  …versio…
#els…
           …roduktionsversion");
#endif
```

C | Compiler incl. Präprozessor | Assembler

C | Compiler incl. Präprozessor | Assembler

C | Compiler incl. Präprozessor | Assembler

C | Compiler incl. Präprozessor | Assembler

C ohne Präprozessor-Anweisungen → Compiler → Assembler → Linker → Maschinen-programm

Compilertechnologie

- unser Referenzsystem:

```
$ hostname
star
$ uname -a
SunOS star 5.11 11.1 sun4u sparc SUNW,SPARC-Enterprise
$ gcc --version
Copyright (C) 2010 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

- andere Compiler können benutzt werden, machen Sie **immer** auch Tests mit dem Referenzsystem (nicht erst kurz vor Abgabe)

- **gcc** steuert **ALLE** Phasen der Übersetzung: kann beliebige Zwischencodes der Übersetzung erzeugen **UND** weiterverarbeiten

`gcc -c x.c`	- *compile only*, erzeugt (falls fehlerfrei) **x.o**
`gcc -E x.c`	- *preprocess only*, Ausgabe nach stdout
`gcc -S x.c`	- *generate asm*, erzeugt (falls fehlerfrei) **x.s**
`gcc m.c`	- *compile and link*, erzeugt (falls fehlerfrei) ausführbares *binary* **a.out**
`gcc -o prog m.c`	- *compile and link*, erzeugt (falls fehlerfrei) ausführbares *binary* **prog**
`gcc -c x.c y.c z.c`	- *compile only*, erzeugt (falls fehlerfrei) **x.o, y.o, z.o**
`gcc -o p x.o y.o z.o`	- *link only*, erzeugt (falls fehlerfrei) ausführbares *binary* **p**
`gcc -o p x.s y.o z.c`	- *compile and link*, erzeugt (falls fehlerfrei) ausführbares *binary* **p**

Compilertechnologie

weitere Optionen

`-Wall`	- *warn all*, erzeugt (alle) Warnungen
`-Wall -pedantic`	- *warn all, pedantic* erzeugt noch mehr Warnungen
`-g`	- *instrument for debu**g**ging*
`-pg`	- *instrument for **p**rofilin**g***
`-Ox [x=1,2,3,s]`	- *optimize(1) more (2) and yet more (3)* for speed, *optimize(s)* for size
`-ansi`	- *accept ANSI-C* (C89==C90)
`-std=cxx [xx=89,90,99]`	- *accept ANSI-C* (C89==C90), *accept C99*
`-Dmacro[=defn]`	- *define macro* (as if **`#define macro defn`** in source file)
`-Umacro`	- *undefine macro* (as if **`#undef macro`** in source file)
`-Ipath`	- search for headers in *path* also
`-lxyz`	- link with *libxyz.a* (z.B. **`-lm`** manchmal nötig für **`libm.a`**)

und **Unmengen** weitere (**`man gcc, info gcc`**)

Die Programmiersprache C

2. Überblick und Einführung

Typen, Variablen, Funktionen, Operatoren

Prof. Dr. sc. Joachim Fischer (Dr. Klaus Ahrens)
Institut für Informatik, Humboldt-Universität zu Berlin
SS 2014

Sprachelemente

- ein C-Programm hat i. allg. folgenden Aufbau:
  - Präprozessor-Kommando(s)
  - Typdefinition(en)
  - Funktionsdeklaration(en):
    Deklaration von Funktionen mit Ein-und Ausgabeparameter(n) (ohne Definition)
  - Variablendefinition(en) und –deklaration(en)
  - Funktionsdefinition(en)

- Quelltext frei formatiert, Bezeichner wie üblich **`letter {letter | digit}*`** (_ is a letter, don't use in front)
- Kommentare eigentlich klassisch `/* … */` aber inzwischen auch fast überall `// …`
- Zeilenden können mit \ maskiert werden

```
"eine ziemlich lange \
zeichenkette"
```

- jedes ausführbare Programm besitzt genau eine (globale) **`main()`**-Funktion

```
int main() { … }                                         // oder
int main(int someName1, char** someName2)  { … }         // oder
int main(int someName1, char* someName2[]) { … }
```

Keywords

Reservierte Bezeichner in C (89, 99):

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

Typen

Einfache (*built-in*) Typen

nicht in C89, in C99 via **<stdbool.h>**

- fehlt: byte
- vorhanden:

bool	**true/false**
short, int, long	ganze Zahlen unterschiedlicher Länge
float, double	reelle Zahlen unterschiedlicher (endlicher!) Genauigkeit
char	Zeichen: ASCII-Code (1 Byte)
(**void**)	kein wirklicher Typ: Funktionen ohne Resultat, **void***

Zeigertypen (auf beliebige Typen)

Indirektion per Adresse

strukturierte (nutzerdefinierte) Typen

- Aufzählungen
- Felder (Arrays)
- Strukturen, Unions

Einfache Typen

C definiert die folgenden einfachen Typen: [mit **typischen** Implementationsgrößen]

C-Typ	Größe (Byte)	Min-Wert	Max-Wert
bool (C99)	1		
char	1	Plattform- abhängig :-(ASCII nutzt aber nur 7 Bit :-)	
unsigned char	1	0	255
short int	2	-32768	+32767
unsigned short int	2	0	65535
(long) int	4	-2^{31}	2^{31}
float	4	-3.4 x 10 [~ 6 Stellen genau]	+3.4 x 10
double	8	-1.7 x 10 [~ 15 Stellen genau]	+1.7 x 10
long long	8	-2^{63}	2^{63}

auch signed, aber
VORSICHT: Vorzeichen-behandlung u.U. nicht portabel

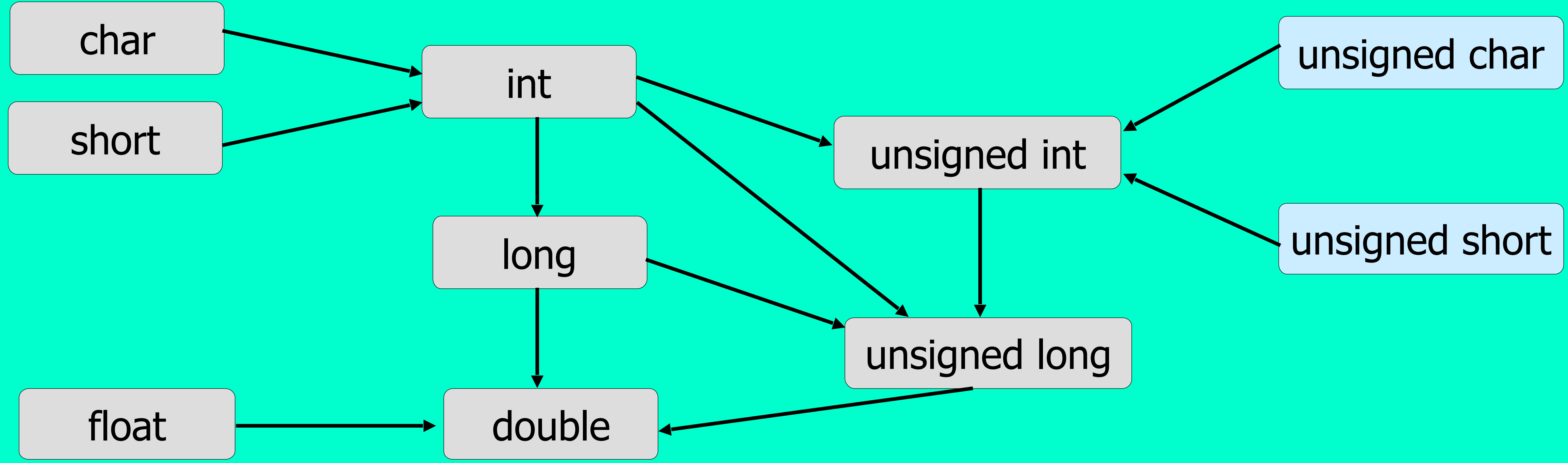
i.allg. zu ungenau
besser immer
double

Typumwandlung - implizit

automatische Typanpassung in Ausdrücken oder Zuweisung `dies = das;`

z.B. 2-stelliger Operator:

Operanden werden immer in Richtung des umfassenderen Typs konvertiert



Typumwandlung - explizit

- durch Cast-Operationen

```
dies = (TypVonDies)das;
```

nur zwischen verwandten Typen erlaubt:

- arithmetische Typen aus der vorigen Folie (u.U. nicht portabel)
- Zeigertypen (u.U. unsicher)

Aufzählungstypen

- Aufzählungstypen setzen sich aus einer Liste von Konstanten zusammen, die auch als Integer-Werte benutzt werden können

  Beispiel:

```
enum days {mon, tues, ..., sun} day;
enum days day1, day2;
```

- wie bei Arrays hat der erste Name den Indexwert 0
  - mon hat Wert 0, tues den Wert 1, usw.
  - `day1` und `day1` sind Variablen
  - die Literale gelten im umgebenden Gültigkeitsbereich !
  - `days` ist **KEIN** eigentlicher Typname (nur *type tag*): `typedef enum days Days;`

- auch andere Werte sind möglich:

```
enum escapes { bell = '\a', backspace = '\b', tab = '\t',
               newline = '\n', vtab = '\v', return = '\r'};
```

- Anfangswert für Index kann auch überschrieben werden:

```
enum months {jan = 1, feb, mar, ......, dec}; // feb==2, mar ==3, …
```

Zeiger

Zeiger oder Pointer

- Das Zeigerkonzept ist ein wesentlicher Teil der Sprache C
- In C werden **Zeiger** intensiv genutzt. **Warum?**
  1. manchmal die einzige Möglichkeit (z.B. dynamische Variablen, siehe malloc/free)
  2. erzeugt kompakten und effizienten Code
  3. erlaubt direkten Zugang zum Hauptspeicher
- C nutzt Zeiger in Zusammenhang mit Feldern, Strukturen, Funktionen

Zeiger

- Ein Zeiger ist eine Variable, die eine Adresse einer anderen Variablen (als Wert) speichert.

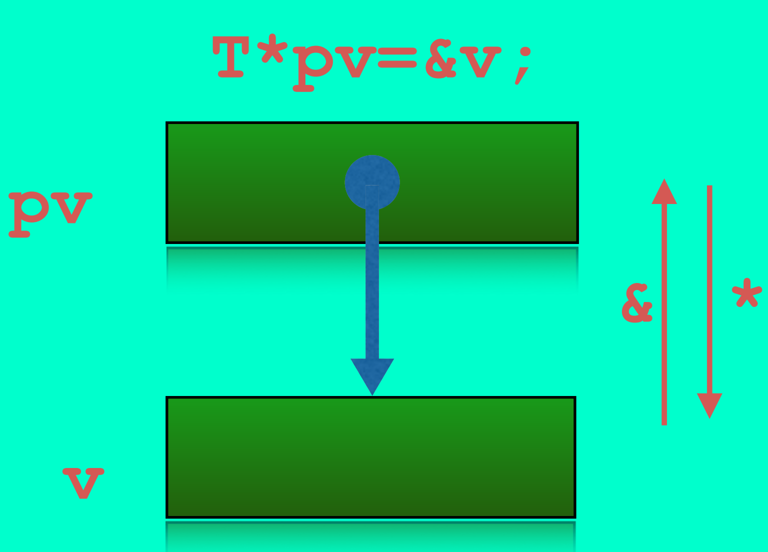
  Zeiger gibt es in C für Variablen beliebigen Typs

Beispiel für die Deklaration eines Zeigers:

```
int *pointer; int*p1; int * p2; int *p3;
int* p1, p2, p3; /* ACHTUNG: nur ein Zeiger, zwei integer */
```

- der monadische Adressoperator „**&**“ gibt die Adresse einer Variablen zurück:
  `T v;` —> **`&v`** hat den Typ **`T*`**

- Der Dereferenzierungsoperator „*“ gibt den „Wert eines Objektes zurück“, auf den der Zeiger zeigt



Zeiger

- ein Zeiger muss mit einem bestimmten Typ assoziiert werden
- nur Zeiger gleichen Typs sind zuweisungskompatibel
- es gibt allerdings einen universellen Zeigertyp **`void*`**, der beliebige andere Zeiger aufnehmen kann:

```
int* pi;                        double* px;
void* pv = pi;                  void* pv = px;

int* pj = (int*)pv;             double* py = (double*)pv;

                py = (double*)pj;
                // wird übersetzt, aber undefined behaviour
```

Zeiger

- Wichtig:
  - wird ein Zeiger deklariert, zeigt er zunächst irgendwo hin
    (nur globale Zeiger sind mit Null initialisiert)
  - Zeiger sollte also immer initialisiert werden (ggf. mit **`NULL`**)

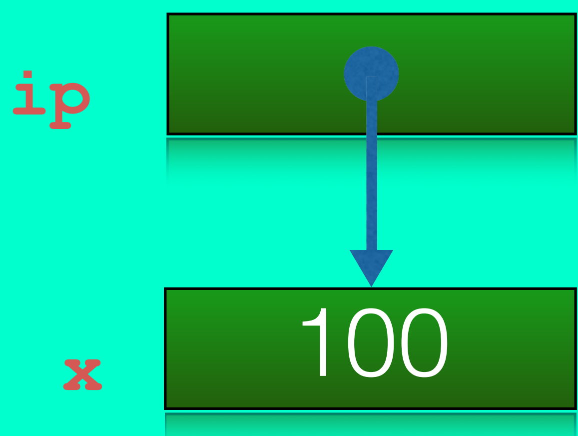
- somit produziert

```
int *ip;   *ip = 100;
```

undefiniertes Verhalten (z.B. Abbruch des Programms oder unbemerktes Überschreiben des Wertes an einer (zufälligen) Adresse oder Formatieren der Festplatte ...).

- Korrektur:

```
int *ip; int x;
ip = &x; *ip = 100;
```



typedef

- kurze Typnamen für Typkonstrukte kann man per

```
typedef Typkonstukt Typname;
```

  definieren

- Benutzung dieser neuen Typen: wie vordefinierte Typen

  Beispiel:

```
/* Typdefinition */
typedef float real;
typedef char letter;
typedef int* intZeiger;
/* Variablendefinition */
real sum = 0.0;
letter nextletter;
```

sizeof

weil die Größen von Objekten im Speicher nicht normativ festgelegt sind, braucht man einen programmatischen Zugang zu dieser Information:

Der Operator **`sizeof`**

- liefert die Speicherplatzgröße (Byteanzahl) als Wert vom Typ **`size_t`** (zumeist unsigned long)
- wird immer zur Compile-Zeit berechnet

- kann in zwei syntaktischen Formen auftreten

  (1) **`sizeof (<ausdruck>)`** beliebiger Ausdruck, Klammern dürfen fehlen

```
int a[10], n;
n= sizeof (a);   /* n= sizeof a; */  Empfehlung: immer Klammern!
```

  (2) **`sizeof (<typ_spezifikation>)`** Klammern dürfen NICHT fehlen

Variablendefinition

Variablendeklaration:

- Typ list_of_variables;
- Beispiel

```
int i,j,k; float x,y,z; char ch;
```

- es gibt keinen Boolean-Typ in (K&R- und ANSI-) C, wohl aber in C99 `#include <stdbool.h>`
- stattdessen wird meist **`int`** benutzt

Definition globaler Variablen

- globale Variablen werden vor dem **`main()`**-Programm wie folgt deklariert/ definiert:

```
short number, sum;
int bignumber, bigsum;
char letter;
int main() { ...  }
```

- nur globale Daten erhalten eine implizite Initialisierung auf Null
- möglich: Initialisierung globaler Variablen mittels Zuweisungsoperator = dann immer eine Definition

- Beispiel:

```
float sum = 0.0;
int bigsum = 0;
char letter = 'A';
int main() {  ...  }
```

Globale Variablen

Speicherverwaltung für globale Variablen

- Speicherplatzreservierung für globale Variablen erfolgt zum Zeitpunkt des Ladens des (übersetzten und verbundenen) Programms,
  Compiler hat bereits die Größe eines zusammenhängenden Speicherbereiches für sämtliche globale Variablen berechnet
- der bereitgestellte Speicherbereich ist mit 0 vorinitialisiert, wenn keine explizite Initialierung erfolgt
- falls nutzerdefinierte Initialisierungen vorgesehen sind, erfolgen diese vor Ausführung der `main()`-Funktion
- Speicherplatzfreigabe erfolgt mit Beendigung des Programms

Definition lokaler Variablen

- lokale Variablen werden innerhalb einer Funktion oder eines lokalen Blockes definiert

```
void foo() {
        short number, sum; /* nicht initialisert ! */
        {
                int sum = 0; /* verdeckt sum aus übergeordnetem Block */
        }
}
```

- unbedingte Empfehlung: Initialisierung lokaler Variablen, wenn erster Zugriff lesend ist

```
void foo() {
        short number = 1, sum = 0; /* initialisert ! */
        {
                int sum = 0; /* verdeckt sum aus übergeordnetem Block */
        }
}
```

Lokale Variablen

Speicherverwaltung für lokale Funktionsvariablen

- Speicherplatzreservierung für lokale Variablen erfolgt zum Zeitpunkt des Funktionsaufrufs im Speicher auf dem Programm-Stack (gehört zum Aktivierungsbereich der Funktion) `auto` (überflüssig)
- eine Initialisierung ist vom Nutzer vorzusehen
- am Ende der Funktion wird Aktivierungsbereich freigegeben (Werte sind verloren)

- Achtung: eine nicht-initialisierte Variable erhält vorheriges (zufälliges) Bitmuster, d.h. der Wert ist undefiniert

- Achtung: eine Adresse einer lokalen Variablen niemals außerhalb des Bezugsrahmens verwenden

Bezugsrahmen von Variablen

Variablen können außerdem zusätzlich als **`static`** ausgezeichnet sein, für globale und lokale mit völlig unterschiedlicher Semantik

- global
  - **`static`** - nur in diesem File sichtbar (nicht Quelltext-übergreifend):
  - ohne **`static`** - (Quelltext-übergreifend) in nur einem File definiert und in anderen [ggf. implizit] als **`extern`** deklariert
- lokal
  - ohne **`static`** - wird bei jeder Ausführung der Funktion/des Blockes neu (auf dem Stack) angelegt, einhält bei fehlender Initialisierung einen undefinierten Wert
  - **`static`** - wird bei jeder Ausführung der Funktion/des Blockes wieder sichtbar, überlebt aber das Ende der Funktion/des Blockes, wird wie globale Daten einmalig auf 0 initialisiert und behält aber den zuletzt hinterlassenen Wert

Konstanten

- ANSI-C erlaubt die Angabe von Konstanten

```
int const a = 1;
const int a = 2;
```

  - Konstantendefinition kann vor oder nach der Typdefinition erfolgen
  - alternativ (aber nicht besser): Definition von Konstanten durch den C-Präprozessor (mehr dazu später)

- `const` wird bei Zeigern (nicht konsequent) berücksichtigt:

```
const int c = 42;
int* p = &c;      ⚠ Initializing 'int *' with an expression of type 'const int *' discards qualifiers
const int* pc = &c;
// (*pc)++;       ❗ Read-only variable is not assignable
```

Ausgabe von Variablen

- C erlaubt formatierte Ausgaben mittels **`printf()`**-Funktion aus der C-Standard-Bibliothek **`#include <stdio.h>`**
- Formatanweisungen werden im ersten Parameter (ein String) kodiert, danach folgen Variablen, die ausgegeben oder eingelesen werden sollen
- **`printf()`** benutzt das spezielle Zeichen % zur Formatierung
  - **`%c`** : characters
  - **`%d`** : integers
  - **`%f`** : doubles (floats werden implizit nach double umgewandelt)
  - **`%s`** : strings, á la "Hallo"
  - **`%p`** : beliebige Zeiger, Adresse hexadezimal
- Beispiel:

```
printf(" %c %d %f \n", ch, i, x);
```

- Wichtig:
  - Der Programmierer ist dafür verantwortlich, dass Formatangaben und Typen der Variablen übereinstimmen,
    sonst undefiniertes Verhalten (z. B. core dump) !!!

Eingabe von Variablen

- C erlaubt formatierte Eingaben mittels **`scanf()`**-Funktion von einfachen Werten und Datenstrukturen
- Formatierung ähnlich zu **`printf`**

```
scanf("%c %d %lf", &ch, &i, &x);
```

- Argumente werden immer per *call by value* übergeben, wie auch Ergebnisse von Funktionen *return by value*
- um Effekte in Argumenten auszulösen braucht man also eine Indirektion (per Zeiger)

Hello World

```
/* sample program */
#include <stdio.h>
int main(){
   printf("Hello, World\n"); return 0;
}
```

- `return` ist ein Statement, das zum Beenden der Funktion führt (in C notwendig! Sonst Warnung und undefiniertes Verhalten!)
  muss gefolgt werden von einem Ausdruck passend zum Rückgabetyp
- C erfordert ein ";" am Ende eines Ausdrucks, um ihn zur Anweisung zu machen (Funktionsruf ist Ausdruck, NICHT Anweisung) !
- "\n" erzeugt ein Zeilenende in der Ausgabe (später mehr)
- Achtung: bei Ausgabe in Dateien ist die ASCII-Kodierung des Zeilenendes abhängig vom Betriebssystem

winDOS: **CR**(015/0xD/,\r')**+LF**(012/0xA/'\n')

Unix: **LF**(012/0xA/'\n')

Funktionen (vorab)

- eine Funktionsdefinition hat folgende Form:

```
type function_name (parameters)
{ local variables   C-Statements    }
```

< C99

- jede Funktion sollte vor ihrem Aufruf per Prototyp deklariert werden !

```
type function_name (parameters);
```

Ansonsten geht der Compiler davon aus, dass die Funktion ein **`int`**-Resultat liefert und beliebige Parameter verarbeitet (was ernsthafte Fehler verursachen kann, wenn dem nicht so ist) !

live DEMO:

Dateioperationen

- Bisher Operationen für Bildschirmein-/Ausgabe
- Datei- Ein- und Ausgabe

  `fscanf` und **`fprintf`** **`f`** wie "File"- Operation

- Datei öffnen und schliessen (im aktuellen Verzeichnis der Programmausführung)

  `fopen (name, modus)` und **`fclose (fileptr)`**
  modus kann sein: "r" (lesend), "w" (schreibend), "a" (anhängend)

```
FILE *fopen(), *fp;

fp = fopen (name, "r");
fscanf(fp, "%d",  &r );
printf ("%d",  r );
fclose ( fp )
…
```

Arithmetische Operationen

- arithmetische Standardoperatoren: + - * / %
- und es gibt noch mehr….
  - ++ und -- mit Variablen in Präfix- und Postfixmodus, also ++x, x++
  - Semantik: erhöhen/reduzieren um den Wert 1

Beispiel

```
int x,y,w;
int main() {   x=((++y) - (w--)) % 100;  return 0; }
```

ist (im Prinzip!) äquivalent zu

```
int x,y,w;
int main() {
    ++y; x=(y-w) % 100; w--;
    return 0;
}
```

ACHTUNG: Die Reihenfolge der Berechnung von Operanden ist in C UNDEFINIERT! (von links nach rechts, umgekehrt, parallel, beliebige Reihenfolge, ...)

Arithmetische Operationen

- Modulo-Operator “%” ist nur für **`int`**-Typ definiert
- Division “/” ist für **`int`** und **`double/float`** definiert
- Achtung:
  - Ergebnis von x = 3 / 2 ist 1, selbst wenn x als **`float`** definiert wurde!!
  - sind beide Argumente von "/" als **`int`** definiert, wird die Operation als integer -Division durchgeführt
- korrekte Spezifikation:
  - **`x = 3.0 / 2`** oder **`x = 3 / 2.0`**
  - oder (besser) **`x = 3.0 / 2.0`**

Berechnung von Operanden

- **ACHTUNG**: Die Reihenfolge der Berechnung von Operanden ist in C UNDEFINIERT!
- Sofern dabei Seiteneffekte möglich sind, hat das Programm undefined behaviour!

Sie müssen lernen zu unterscheiden zwischen:

- Was macht mein C-Programm (beobachtbar) auf einer bestimmten Plattform (Compiler + Betriebssystem-Umgebung) ?
- Was legt der Sprachstandard fest ?

Berechnung von Operanden

Damit ist das Argument:

„Aber auf der Plattform XYZ hat mein Programm funktioniert!"

wertlos !

Neue Qualität von Problemen bei der Programmentwicklung
(die Sie von Java nicht gewohnt sind)

Hilfsmittel:

- profunde Sprachkenntnis (insb. der Teile, die NICHT explizit definiert sind)
- Maximales Warnungslevel im Compiler einstellen und Warnungen ernst nehmen
- Cross-Checks: verschiedene Compiler auf verschiedenen Plattformen nutzen
- Meta-Tools (flex, bison) generieren korrekten Code

Berechnung von Operanden

Ein Beispiel zur Abschreckung:

```
#include <stdio.h>

int foo(int x, int y)
{ return x + y; }

int main() {
      int i=1;

      printf("%d\n", foo(++i, --i));
      return 0;
}
```

Berechnung von Operanden

```
class Foo {
      static int foo(int x, int y)
      { return x + y; }

public static void main(String[] s) {
            int i=1;
            System.out.println(foo(++i, --i));
      }
}
```

Ausgabe ?

3

Berechnung von Operanden

<u>gcc 3.3.3</u>

```
$ gcc -Wall -o foo foo.c
foo.c: In Funktion »main«:
foo.c:11: Warnung: operation on `i' may be undefined
$ foo
1
$
```

Reduziere i (1 -> 0)
Wert von i (0) auf den Stack
Erhöhe i (0 -> 1)
Wert von i (1) auf den Stack
Rufe foo: return 1 + 0

Berechnung von Operanden

vc++ 7.1

```
H:>cl /W4 foo.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 13.10.3077 for 80x86
Copyright (C) Microsoft Corporation 1984-2002. All rights reserved.

Microsoft (R) Incremental Linker Version 7.10.3077
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:foo.exe
foo.obj

H:>foo
2

H:>
```

Erhöhe i (1 -> 2)
Reduziere i (2 -> 1)
Wert von i (1) auf den Stack
Wert von i (1) auf den Stack
Rufe foo: return 1 + 1

Beide Compiler arbeiten korrekt !

Kurzform von Operatoren

- C stellt "elegante" Abkürzungen für Operatoren zur Verfügung
  - Beispiel: `i = i + 3` oder `x = x*(y + 2)`
- Umschreibung in C (generell) in "Kurzform":
  $expression_1$ op = $expression_2$
- Dies ist äquivalent zu (und u. U. effizienter als):
  $expression_1$ = $expression_1$ op $expression_2$
- Beispiel umgeformt:
  `i = i + 3` als `i += 3`
  `x = x*(y + 2)` als `x *= y + 2`

  `x *= y + 2` bedeutet `x = x*(y + 2)` und nicht `x = x*y + 2`

Vergleichsoperatoren

- Test auf Gleichheit: "=="

  Achtung: Bitte "=" nicht mit "==" verwechseln !!!

- zulässig ist auch: **`if ( i = j ) ...`**

  - legales C-Statement (aus syntaktischer Sicht):
    Zuweisung des Wertes von "j" nach "i",
    gleichzeitig Wert des Ausdrucks, der als TRUE interpretiert wird,
    falls j ungleich 0 ist

  - manche Compiler (nicht alle) warnen

Vergleichsoperatoren

- ungleich ist: "!="
- andere Operatoren
  - < (kleiner als)
  - > (größer als)
  - <= (kleiner oder gleich),
  - >= (größer oder gleich)

Logische Operatoren

Die logischen Grundoperatoren sind:

- `&&` für logisches AND (short circuit evaluation!)
- `||` für logisches OR (short circuit evaluation!)
- `!` Für logisches NOT

Achtung: `&` und `|` existieren auch als zweistellige Operatoren, haben aber eine andere Semantik:

  - Bit-orientiertes AND
  - Bit-orientiertes OR (später)

Achtung: `&` ist auch ein einstelliger Operator (Adresse von)

- Verwendung in logischen Ausdrücken (als **`int`** bewertet)

Präzedenzen von Operatoren

- Bedeutung von `a + b * c`
  - Gemeint könnte sein
    - `(a + b) * c`
    - `a + (b * c)`
  - alle Operatoren besitzen einen “Präzedenzwert” (Priorität)
  - Operatoren mit hoher Priorität werden vor Operatoren mit geringerer Priorität evaluiert
  - Operatoren mit gleicher Priorität werden von links nach rechts evaluiert, wenn sie rechts-assoziativ sind:
    - `a - b - c` wird als `( a - b ) - c` evaluiert
- im Zweifelsfall besser ein Klammerpaar zu viel, als eines zu wenig

Präzedenzordnung

- Operatoren in C von hoher bis niedriger Priorität (Präzedenz):
  (sind noch nicht alle eingeführt):

```
() [] -> .
! - * & sizeof cast ++ --   (diese werden von rechts nach links ausgewertet)
/ %
+ -
< <= >= > == !=
&
|
&&
||
?:                                   (rechts nach links)
= += -= ....                         (rechts nach links)
, (comma)
```

- **Beispiel**: `a < 10 && 2 * b < c` wird als
  `( a < 10 ) && ( ( 2 * b ) < c )` interpretiert

- `i = foo(), bar(), 42;  // rufe foo(), rufe bar(), i = 42;`

  Konflikt mit Komma in anderen Kontexten mit Klammern lösbar:
  `int i = (foo(), bar(), 42); baz((foo(), bar(), 42));`

Die Programmiersprache C

3. Anweisungen, Funktionen, Felder, Strukturen

Prof. Dr. sc. Joachim Fischer (Dr. Klaus Ahrens)
Institut für Informatik, Humboldt-Universität zu Berlin
SS 2014

Anweisungen

Algorithmik analog zu Java in Syntax & Semantik

- Zuweisung (auch +=, -=, …)
- **`if, switch`**
- **`while, do-while, for`**
- **`break, continue`**
- Funktionsaufruf (Java: Methodenaufruf)
- **`return`**

```
loop: x=y;
…
goto loop;
```

```
goto skip;
…
skip:
```

nicht vorhanden

throw, try, catch, synchronized

Zuweisungsoperator

- Zuweisung durch "="
- Zuweisung ist KEINE Anweisung, sondern ein Ausdruck (wie in Java auch) !
- C erlaubt Mehrfachzuweisungen (wie Java)
- Beispiel:

```
a=b=c=d=3;
```

- ...dies ist äquivalent zu (aber nicht notwendig effizienter als):

```
d=3; c=3; b=3; a=3;
```

if - Anweisung

- Grundform:
  `if` (expression) statement
  `if` (expression) $statement_1$ **`else`** $statement_2$

```
if (exp1)
    if (exp2) stmt1
    else stm2
```

- Schachtelung möglich:
  `if` (expression) $statement_1$
  `else if` (expression) $statement_2$
  `else` $statement_3$

dangling else
- Bindung an das innerste if
- auch durch Formatierung unterstreichen

- Beispiel:

```
int main() {     int x, y, w, z;
    if (x>0) {... z=w; ... }
    else     {... z=y; ... }
}
```

```
if (exp1)
    {if (exp2) stmt1}
else stm2
```

Operator ? :

- Der »? :«-Operator (»ternary condition«) ist die effizientere Form, um einfache **`if`**-Anweisungen auszudrücken

- syntaktische Form:
  $expression_1$? $expression_2$: $expression_3$

  $expression_2$ und $expression_3$ müssen kompatible Typen haben

- Semantik:
  if $expression_1$ then Wert = $expression_2$ else Wert = $expression_3$

- Beispiel: Zuweisung des Maximums von a und b auf z

```
z = (a > b) ? a : b;
```

  äquivalent zu:

```
if (a > b) z = a; else z = b;
```

switch - Anweisung

- Die **`switch`**- Anweisung erlaubt mehrfache Alternativen einer Selektion auf einer »Ebene«

```
switch (expression) {
        case item1: statement1
        case item2: statement2
        case itemn: statementn
        default  : statement
}
```

- In jeder Alternative muss der Wert von $item_i$ eine Konstante sein, Variablen sind nicht erlaubt

- leere Anweisung durch ein »;« möglich

switch - Anweisung

- Beispiel:

```
switch (letter) {
    case 'A': howmanyAs++; /* fall through */
    case 'E':
    case 'I':
    case 'O':
        case 'U': numberofvowels++; break;
        case ' ': numberofspaces++; break;
        default:  numberofothers++; break;
}
```

switch - Anweisung

- Die Syntax der switch-Anweisung ist in Wahrheit:
  switch (expression) statement
- damit kann man (anders als in Java) überraschende Effekte ausdrücken, Beispiel: *Duffs Device* (loop unrolling)

```
send(to, from, count)
register short *to, *from;
register count;
{
    do
        *to = *from++;
    while (--count>0);
}
```

```
send(to, from, count)
register short *to, *from;
register count;
{
    register n = (count+7)/8;
    switch (count%8) {
        case 0:    do { *to = *from++;
        case 7:         *to = *from++;
        case 6:         *to = *from++;
        case 5:         *to = *from++;
        case 4:         *to = *from++;
        case 3:         *to = *from++;
        case 2:         *to = *from++;
        case 1:         *to = *from++;
        } while (--n>0);
    }
}
```

http://de.wikipedia.org/wiki/Duff's_Device

switch - Anweisung

- Die Syntax der switch-Anweisung ist in Wahrheit:

  switch (expression) statement

- damit kann man (anders a
  Beispiel: *Duffs Device*

```
send(to, from, count)
register short *to, *from;
register count;
{
    do
        *to = *from++;
    while (--count>0);
}
```

im (Ur-) C (auch K&R C) wurde die Typinformation von Parametern nach der Parameterliste spezifiziert

http://de.wikipedia.org/w

switch - Anweisung

- Die Syntax der switch-Anweisung ist in Wahrheit:

  switch (expression) statement

- damit kann man (anders als in Java) überraschende Effekte ausdrücken, Beispiel: *Duffs Device* (loop unrolling)

register ist (immer noch) ein Keyword von C (auch aus K&R-Zeiten) um anzuzeigen, dass Parameter/ lokale Variablen in Registern (statt im Hauptspeicher) angelegt werden sollen - heute *depricated:* der Compiler macht das in eigener Regie

```
send(to, from, count)
register short *to, *from;
register count;
{
    register n = (count+7)/8;
    switch (count%8) {
        case 0:    do {  *to = *from++;
        case 7:          *to = *from++;
        case 6:          *to = *from++;
        case 5:          *to = *from++;
        case 4:          *to = *from++;
        case 3:          *to = *from++;
        case 2:          *to = *from++;
        case 1:          *to = *from++;
        } while (--n>0);
    }
}
```

for - Anweisung

- Die for-Anweisung hat die folgende Form:

  `for` (for-init-statement $expression_1$; $expression_2$) statement

  Erklärung:
  - for-init-statement initialisiert die Iteration
  - $expression_1$ ist der Test zur Beendigung der Iteration
  - $expression_2$ modifiziert eine Schleifenvariable (mehr als nur das Erhöhen eine Schleifenvariablen um 1)
- C99 kennt auch sog. for-Scope: **`for (int i=1;…;…) …`**

  C benutzt **`for`**-Anweisung oft anstelle von **`while`**-Schleifen
- Beispiel:

```
int main() { int x; for (x=3;x>0;x--) { printf("x=%d\n",x); } }
```

… erzeugt als Ausgabe:

```
x=3
x=2
x=1
```

while - Anweisung

- Die while-Anweisung hat die folgende Form:

  `while` (expression) statement

  Beispiel:

```
int main() {int x=3;
   while (x>0) { printf("x=%d\n",x); x--; }
}
```

  ...erzeugt als Ausgabe:

```
x=3
x=2
x=1
```

- legale while-Anweisungen:
  - `while (x--)   ...`
  - `while (x=x+1) ...`
  - `while (x+=5)  ...`
  - `while (1) ...          /* forever,  auch for(;;) ... */`

while - Anweisung

üblich in C: vollständige Ausführung von Operationen im **`while`**-Ausdruck:

```
while (i++ < 10);
while ( (ch = getchar()) != 'q' )
    putchar(ch);
while (*dest++ = *src++); // ????
/* klassisches C-Idiom (strcpy) */
```

do-while - Anweisung

do-while-Anweisung hat die Form:

do statement **while** (expression); <- hier Semikolon explizit !

Beispiel:

```
int main() {
  int x=3;
  do { printf("x=%d\n", x--); }
  while (x>0);
}
```

... erzeugt als Ausgabe:

```
x=3
x=2
x=1
```

break und continue

C enthält zwei Möglichkeiten zur Schleifensteuerung:

- **`break`**: Verlassen der (innersten) Schleife oder **`switch`**-Anweisung.
- **`continue`**: Überspringen einer Schleifeniteration

Beispiel:

```
while (scanf("%d", &value ) == 1 && value != 0) {
  if (value < 0) {
    printf("Illegal value\n"); break; /* Abandon the loop */
  }
  if (value > 100) {
    printf("Invalid value\n"); continue; /* Skip to start loop again */
  }
  /* Process the value read, guaranteed to be between 1 and 100 */
  …;
} /* end while */
```

Funktionsdefinition

- Form

  returntype fn_name (paramdef1, paramdef2, ...)
  { localvariables
  functioncode }

  bei C99 (und C++) nicht zwingend nur am Anfang

- Beispiel Durchschnitt zweier **`float`**-Werte

```
/* besser, weil kompakter: */
float findaverage (float a, float b) {
   return (a+b)/2;
}
```

- Aufruf der Funktion

```
void foo() { float a=5, b=15, result;
             result=findaverage(a,b);
             printf("average=%f\n", result); }
```

Funktionen

Basissyntax (Definition) wie bei Java-Methoden

```
void readvectors (vector v1, vector v2) {
    int i;      /* zu Beginn */
    ...   /* dann Anweisungen */
          /* ab C99 Typ- u. Variablendeklarationen später möglich */
}
```

Besondere Sichtbarkeitsregeln:

```
static void readvectors (v1, v2);
// nur in Übersetzungseinheit (File) sichtbar

extern void readvectors (v1, v2);
// auch nach außen sichtbar, extern ist Standardannahme
```

Funktionen

Verschachtelung von Funktionen

- wie in Java **nicht** erlaubt, in C sind alle Funktionen global,
  obwohl Blockkonzept (Gültigkeitsbereiche für Bezeichner) seit Algol-60 bekannt

- **Gründe**:
  - Leichter und effektiver durch Compiler zu verarbeiten (Compilezeit)
  - Verwaltungsaufwand für Funktionsrufe geringer (Laufzeit)

- **Kritik**: methodischer Nachteil
  Programmstruktur entspricht u.U. nicht der Problemstruktur

Funktionen

Resultattyp von Funktionen

- erlaubt sind: alle Typen auch insbesondere strukturierte Typen (Werte werden als Kopie nach außen gereicht)
- Vorsicht bei der Rückgabe von Adressen:
  - bei der Rückgabe werden nur „Henkel" kopiert, nicht aber die Objekte der „Henkel"
  - werden Adressen lokaler Objekte zurückgegeben, ist die weitere Programmausführung nicht definiert

- Rückgabe von Adressen von globalen Objekten oder dynamisch angelegten Objekten auf der Halde ist dagegen der typische Anwendungsfall (später)

Funktionen

- falls kein Wert zurückgegeben wird
  - sollte der Rückgabewert der Funktion **`void`** sein
  - dann darf eine **`return`**-Anweisung nur ohne Ausdruck benutzt werden
- Beispiel:

```
void squares() {
      for (int loop=1; loop<10; loop++);
            printf("%d \n", loop*loop);
}
```

auch dieses Programm ist (leider) korrekt:

```
int main() {
    squares;    ⚠ Expression result unused
    return 0;}
```

- auch wenn keine Parameter übergeben werden, müssen beim Aufruf Klammern nach dem Funktionsnamen folgen

Rekursion

Beispiel (kein Problem in C):

```
#include <stdio.h>

int fact(int n) {
    /* fact(n) = n*(n-1)*....2*1 */
    if (n == 0) return 1;
    return n * fact(n-1);
}

int main() {
    int n, m;
    printf("Enter a number: "); scanf("%d", &n);
    m = fact(n);
    printf("The factorial of %d is %d.\n", n, m);
    return 0;
}
```

Statische (lokale) Variablen

```
static void flup() {
    int l = 0;
    static int s = 0;
    printf("[1]\tauto: %d\tstatic: %d\n", l++, s++);
    {
        int l = 4;
        static int s = 4;
        printf("[2]\tauto: %d\tstatic: %d\n", l++, s++);
    }
}

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

    for (int i=0; i<3;++i) flup();

    return 0;
}
```

```
[1] auto: 0   static: 0
[2] auto: 4   static: 4
[1] auto: 0   static: 1
[2] auto: 4   static: 5
[1] auto: 0   static: 2
[2] auto: 4   static: 6
```

Felder (Arrays)

- Beispiel:

```
int listofnumbers[50];
```

- Achtung:
  - In C-Arrays beginnt Indizierung bei 0 and endet mit Index, der um eins kleiner ist als seine Größe.
  - Die Feldvariable enthält **KEINE** Information über die Feldgröße (wie in Java)
  - im Beispiel: legaler Index umfasst den Wertebereich 0 bis 49

- auf Elemente des Arrays kann man folgendermaßen zugreifen:

```
thirdnumber = listofnumbers[2];
listofnumbers[5] = 100;
/* Aber leider auch: */ int undefined = listofnumbers[50];
```

Initialisierung von Feldern

```
int felda[5] = {0, 1, 2, 3, 4};
// semantischer Fehler: int felda[5]={0, 1, 2, 3, 4, 5}
// Speicherbereitstellung für Datenobjekt mit 5
// int-Elementen und Initialisierung
```

oder...

```
int feldb[] = {6, 7, 8, 9, 10};
// Speicherbereitstellung für berechnete Größe des
// Datenobjektes und Initialisierung
```

`felda` und `feldb` sind gleich groß

Initialisierung von Feldern

```
int felda[5] = {0, 1};
// Speicherbereitstellung für Datenobjekt mit 5 int-
// Elementen und Initialisierung aller Feldelemente:
// 0, 1, 0, 0, 0
```

aber...

```
int feldb[] = {0, 1};
// Speicherbereitstellung für 2 int-Elemente
// und Initialisierung: 0, 1
```

`felda` und `feldb` sind gleich groß

ohne Dimension und ohne Initialisierung nur als Argumenttyp erlaubt!

Zeiger und Felder

- Zeiger und Felder sind verwandte Konzepte in C

  Beispiel:

```
int a[10], x;
int *pa;
pa = &a[0];  /* pa zeigt auf Adresse von a[0] */
x = *pa;     /* x = Inhalt von pa ( also a[0] in diesem Fall) */
```

0 1 9

a

pa ++pa pa+i

- für Wertezugriffe im Feld werden Zeiger benutzt

  pa + i == & a[i]

- Achtung: Keine Grenzwertüberprüfung beim Zugriff auf das Feld

  es ist leicht, Werte anderer Speicherzellen zu überschreiben

Zeiger und Felder

- legaler Ausdruck:

  Zuweisung:

  `pa = a;` statt `pa = &a[0]`

- äquivalente Ausdrücke:

  &a[i] ≡ a + i /* Achtung: Offset ist: i*sizeof(int) und nicht i */

  a[i] ≡ *(a + i) ≡ ☺ ≡ *(i + a) ≡ i[a]

- Unterschied zwischen Zeiger und Feld:
  - ein Zeiger ist variabel:

    `pa = a;` und **`pa++;`**

  - ein Feld ist **nicht** variabel:

    `a = pa;` und **`a++;`** sind illegale Anweisungen

Zeiger und Felder

```
int f[n], *pint, k;/* ein Feld, ein Zeiger, ein Integer */
```

`f` und `pint` sind vom selben Typ:
Adressen von Integer-Werten (Zeiger auf Integer)

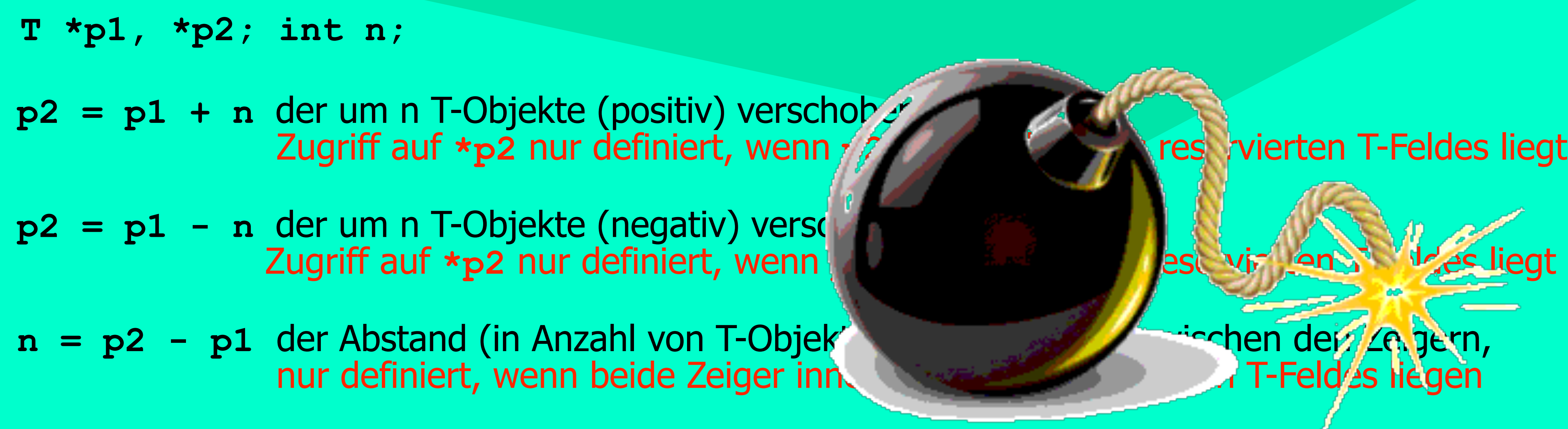
jeweils dasselbe sind:

```
pint = f;          pint = &f[0];
f[0] = 1;          *pint = 1;           /* 0. Element */
f[1] = 1;          *(pint+1) = 1;       /* 1. Element */
f[k] = 1;          *(pint+k) = 1;       /* k-tes Element */
```

Adressarithmetik

Anwendung arithmetischer Operationen auf Zeiger:

```
T *p1, *p2; int n;
```



`p2 = p1 + n` der um n T-Objekte (positiv) verschob[illegible]
Zugriff auf `*p2` nur definiert, wenn [illegible] res[illegible]rvierten T-Feldes liegt

`p2 = p1 - n` der um n T-Objekte (negativ) versc[illegible]
Zugriff auf `*p2` nur definiert, wenn [illegible]

`n = p2 - p1` der Abstand (in Anzahl von T-Objek[illegible] [illegible]ischen den Zeigern,
nur definiert, wenn beide Zeiger inn[illegible] T-Feldes liegen

Achtung:

- Obwohl Zeiger als numerischer (Adress-)Wert realisiert werden, **sind sie keine Integer-Werte**

Darum sind Zeiger typisiert:

- wird ein Zeiger vom Typ `T*` um `n` erhöht, so wird er um `sizeof(T) * n` Bytes erhöht

Adressarithmetik

Beispiele

```
char* ch_ptr = …;
++ch_ptr;
/* erhöht Adresse um sizeof(char), immer 1  */

int *i_ptr = …;
float *f_ptr = …;
++i_ptr; /* erhöht Adresse um sizeof(int),   z.B. 4 */
++f_ptr; /* erhöht Adresse um sizeof(float), z.B. 4 */
```

Zeichenketten (Strings)

- in C werden Strings definiert als Felder von char
- Beispiel: String mit 50 Zeichen

```
char name[50];
```

- C hat »per se« keine String-Operationen
- somit sind folgende Anweisung nicht möglich:

```
char firstname[50], lastname[50], fullname[100];
firstname = "Arnold"; /* Illegal */
lastname = "Schwarzenegger"; /* Illegal */
fullname = "Mr"+firstname +lastname; /* Illegal */
```

- es gibt jedoch eine String-Bibliothek (später)
- um einen String zu drucken, wird "%s" als Formatangabe benutzt:

`printf("%s", name);` **äquivalent zu** `printf(name);` **???**

Zeichenketten (Strings)

- Stringliterale sind vom Typ **`[const] char* bzw. [const] char[]`**
- werden als globale (schreibgeschützte) Daten mit einem abschließenden 0-Byte abgelegt
- können zur Initialisierung von Stringvariablen verwendet werden

```
const char* s1  = "s1";
const char s2[] = "s2";
const char s3[] = {'s', '3', 0};
const char s4[10] = "s4";
const char s5[10] = {'s', '5'}; // 0 terminated ?
const char s6[10] = {'s', '6', '\0', 'm', 'o', 'r', 'e'};

printf("%s\n", s1);
printf("%s\n", s2);
printf("%s\n", s3);
printf("%s\n", s4);
printf("%s\n", s5);
printf("%s\n", s6);
printf("%s\n", s6+3);
```

```
s1
s2
s3
s4
s5
s6
more
```

```
// !!!
char* s = "s1";
s[0] = 'S';
```

Thread 1: EXC_BAD_ACCESS (code=2, address=0x100000f61)

Funktionen und Arrays

- eine Funktion, die ein Feld als Parameter übergeben bekommt, hat keine Information über die Größe des Feldes

```
void foo1(int* f){ // no idea how
    for (int i=0; i<5; ++i)
        printf("%d ", f[i]);
    printf("\n");
}

void foo2(int f[]){ // no idea ho
    for (int i=0; i<5; ++i)
        printf("%d ", f[i]);
    printf("\n");
}

void foo3(int f[5]){ // no idea h
    for (int i=0; i<5; ++i)
        printf("%d ", f[i]);
    printf("\n");
}
```

```
int f1[]={1,2,3,4,5,6,7,8};
int f2[]={1,2,3};

/* all calls compile :-( */
foo1(f1);1 2 3 4 5
foo1(f2);1 2 3 0 1
foo2(f1);1 2 3 4 5
foo2(f2);1 2 3 0 1
foo3(f1);1 2 3 4 5
foo3(f2);1 2 3 0 1
         1 2 3 4 5
```

Funktionen und Arrays

- daher muss einer Funktion, die ein Feld als Parameter übergeben bekommt, die Größe des Feldes explizit übergeben werden

```
void foo4(int f[], int l){ // tell me in l how big the array really is
    if (l<5) {
        printf("some error here\n");
        return;
    }
    for (int i=0; i<5; ++i)
        printf("%d ", f[i]);
    printf("\n");
}
```

```
int f1[]={1,2,3,4,5,6,7,8};
int f2[]={1,2,3};
…
foo4(f1);   1 2 3 4 5
foo4(f2);   some error here
```

Funktionen und Arrays

klassisches C-Idiom: strcpy (so etwa in **string.c**)

kopiere Zeichenkette t nach Zeichenkette s

Voraussetzungen: s ist hinreichend groß, t ist ‚\0' terminiert

```
char* strcpy(char *dst, const char *src) {
      char *d = dst;
      while ( *dst++ = *src++ );
      return d;
}
```

- **while**-Schleife hat keinen Block, alles passiert als Seiteneffekt der Ausdrucksberechnung
- Ausdruck liefert Wert (zugewiesener Wert), der logisch ausgewertet wird
- Zuweisung erfolgt an den Inhalt der Zelle (Indirektion!)
- Erhöhung der Zeigervariablen (Position im String) nach der Zuweisung
- Abbruchbedingung '\0' == 0

Multidimensionale Felder

Multi-dimensionale Arrays können wie folgt definiert werden:

```
int tableofnumbers[50][50];  /* zwei Dimensionen */
```

- für weitere Dimensionen werden weitere »[ ]« hinzugefügt:

```
int bigD[50][50][40][30]......[50];
```

- auf Elemente kann man wie gewohnt zugreifen:

```
anumber = tableofnumbers[2][3];
tableofnumbers[25][16] = 100;
```

Multidimensionale Felder

- 2D-Felder sind in Wirklichkeit 1D-Felder, bei dem jedes Element wiederum ein Feld ist
- mehrdimensionale Felder werden im (sequentiellen :-) Speicher entlang der Indizes (beim letzten beginnend) linearisiert
- dies reflektiert auch die Initialisierung von mehrdimensionalen Feldern

```
int f3[3][2]={{1,2},{3,4},{5,6}};
int f4[3][2]={1,2,3,4,5,6};

int f5[2][3]={1,2,3,4,5,6};
int f6[2][3]={{1,2,3},{4,5,6}};
```

1	2	3	4	5	6	**f3/f4**
1	2	3	4	5	6	**f5/f6**

- werden 2D-Felder an Funktionen übergeben, so kann deren (Zeilen-)Struktur explizit beschrieben werden, dies steuert, wie in der Funktion auf das Feld zugegriffen wird
  - die Anzahl der Zeilen ist zusätzlicher Parameter bereitzustellen
  - Grund: C muss “wissen” , wie viele Spalten es gibt, um von einer Zeile zur nächsten springen zu können

Multidimensionale Felder

- auch hier kann die „äußerste“ Dimension als Parameter übergeben werden
- ob auch wirklich „passende“ Felder übergeben werden, kann zur Laufzeit **NICHT** geprüft werden!

1	2	3	4	5	6	f3/f4
1	2	3	4	5	6	f5/f6

```
void foo5(int f[][2]) {
   for (int i=0; i<3;++i) {
      for (int j=0; j<2; ++j)
         printf(" %d", f[i][j]);
      printf("\n");
   }
}
…
foo5(f3);
foo5(f4);
foo5(f5);
foo5(f6);
```

⚠ Incompatible pointer types passing 'int [2][3]' to parameter of type 'int (*)[2]'
⚠ Incompatible pointer types passing 'int [2][3]' to parameter of type 'int (*)[2]'

alle Aufrufe geben aus:

```
1 2
3 4
5 6
```

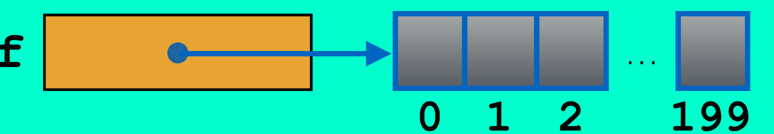
Multidimensionale Felder

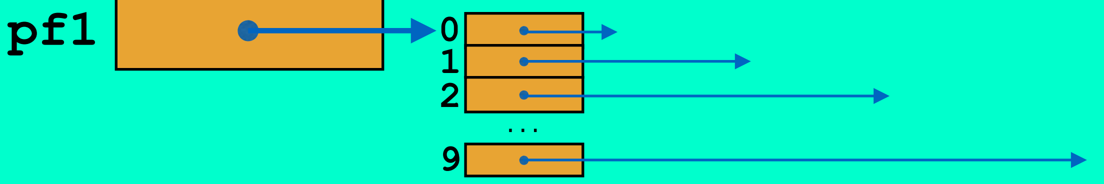
- alternativ können Felder von Zeigern verarbeitet werden

```
char *pf0[10];  // 10 (undef.)Zeiger auf char(-Felder unbekannter Länge)
char f[10][20]; // 200 Byte sequentiell im Speicher mit 2-dim. Zugriff
char *pf1[10]={f[0],f[1],f[2],f[3],f[4],f[5],f[6],f[7],f[8],f[9]};
```

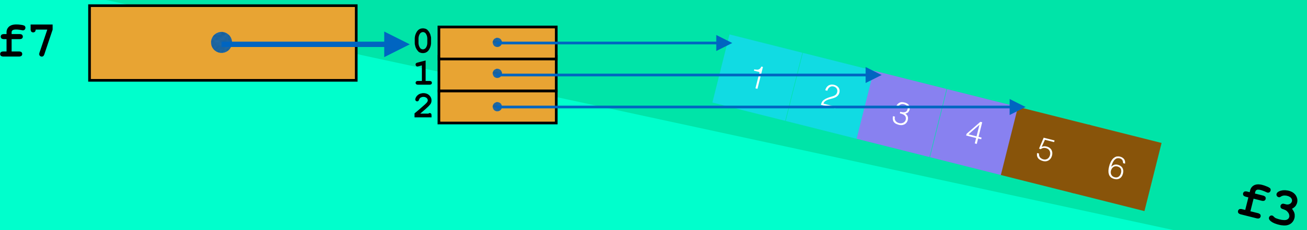
Achtung:

- `pf0` ist ein Feld mit 10 Zeigerelementen (ohne Speicher für Objekte)
- `f` ist ein 200-elementiges 2D-Feld vom Typ `char` (mit Speicher für Objekte)
- `pf1` ist ein Feld mit 10 Zeigerelementen (mit Speicher für Objekte)





Multidimensionale Felder



- die Verarbeitung muss dann natürlich mit passenden Funktionen erfolgen

```
void foo6(int* f[3]) {
   for (int i=0; i<3;++i) {
      for (int j=0; j<2; ++j)
         printf(" %d", f[i][j]);
      printf("\n");
   }
}

int* f7[3]={f3[0], f3[1], f3[2]};

// oder   void foo7(int* f[])
// besser void foo8(int* f[], int n)
```

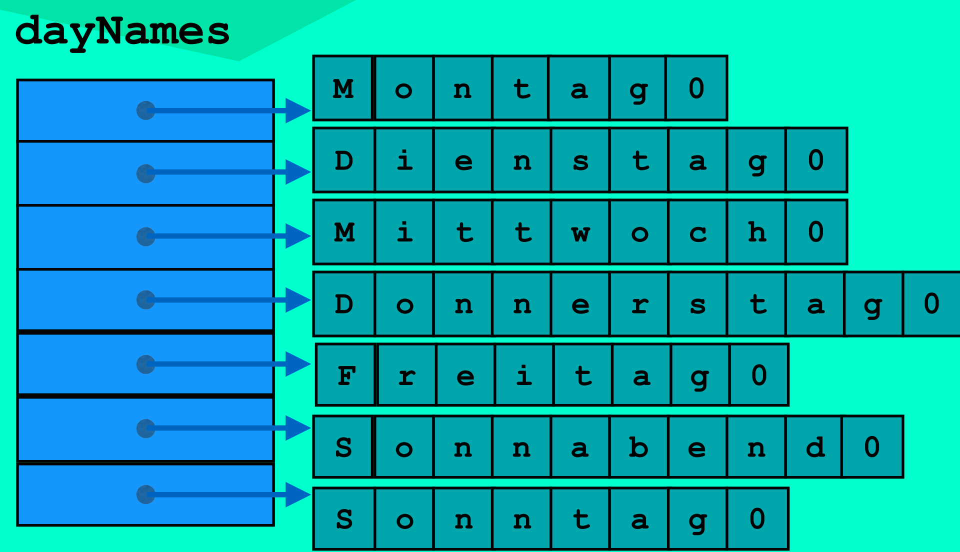
```
foo6(f7);
```

```
1 2
3 4
5 6
```

Multidimensionale Felder

- die Übergabe als Feld von Zeigern fordert nicht, dass die referenzierten Felder alle gleiche Länge haben, allerdings muss dann die individuell Länge anderweitig zugänglich sein - typischer Anwendungsfall: Strings mit impliziter Längeninformation (0-Byte)

```
const char* dayNames[] = { "Montag", "Dienstag",
                           "Mittwoch", "Donnerstag",
                           "Freitag", "Sonnabend",
                           "Sonntag" };

void printAllDays(char* f[]) {
    for (int i = 0; i<7; ++i)
        printf("%d: %s\n", i, f[i]);
}

int main(int argc, const char * argv[])
{
    printAllDays(dayNames);
    return 0;
}
```




```
0: Montag
1: Dienstag
2: Mittwoch
3: Donnerstag
4: Freitag
5: Sonnabend
6: Sonntag
```

Multidimensionale Felder

- die Übergabe als Feld von Zeigern fordert nicht, dass die referenzierten Felder alle gleiche Länge haben, allerdings muss dann die individuell Länge anderweitig zugänglich sein - typischer Anwendungsfall: Strings mit impliziter Längeninformation (0-Byte)

```
const char* dayNames[11] = { "Montag", "Dienstag",
                             "Mittwoch", "Donnerstag",
                             "Freitag", "Sonnabend",
                             "Sonntag" };

void printAllDays(char* f[]) {
    for (int i = 0; i<7; ++i)
        printf("%d: %s\n", i, f[i]);
}

int main(int argc, const char * argv[])
{
    printAllDays(dayNames);
    return 0;
}
```

M	o	n	t	a	g	0	0	0	0	0
D	i	e	n	s	t	a	g	0	0	0
M	i	t	t	w	o	c	h	0	0	0
D	o	n	n	e	r	s	t	a	g	0
F	r	e	i	t	a	g	0	0	0	0
S	o	n	n	a	b	e	n	d	0	0
S	o	n	n	t	a	g	0	0	0	0

```
0: Montag
1: Dienstag
2: Mittwoch
3: Donnerstag
4: Freitag
5: Sonnabend
6: Sonntag
```

Multidimensionale Felder

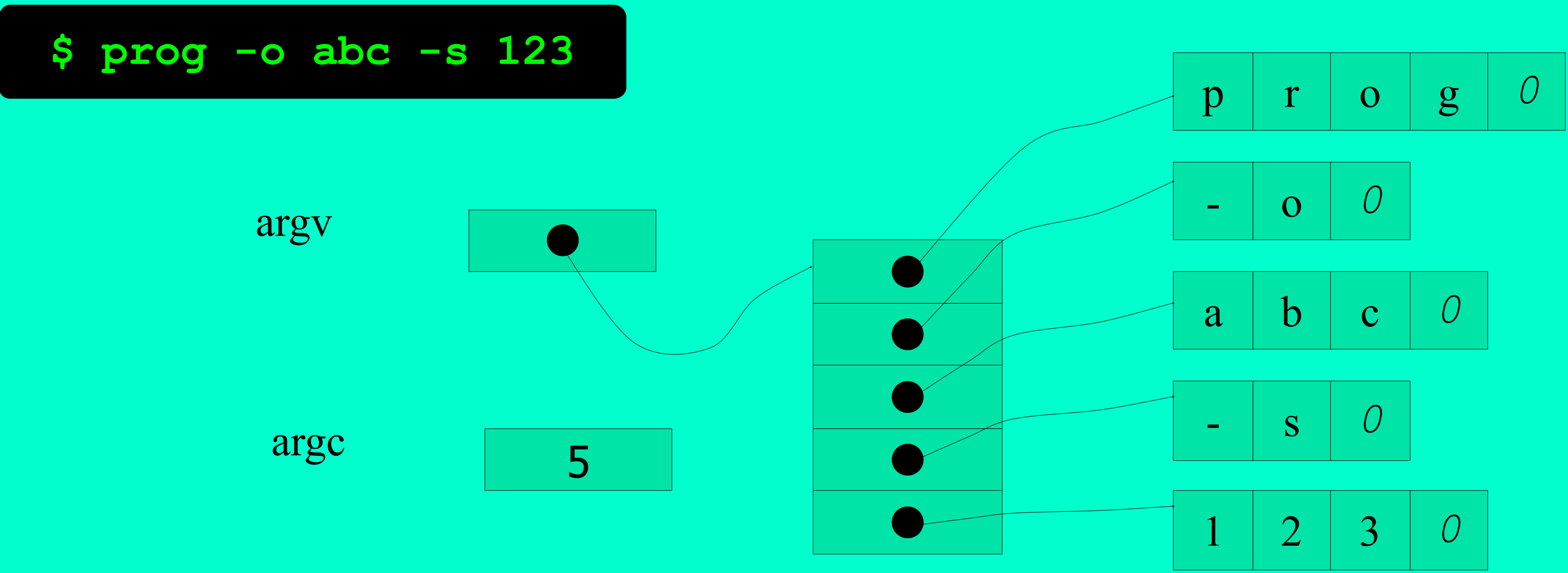
- auch die Länge des Zeigerfeldes wird häufig implizit bereitgestellt:

```
const char* dayNames[] = { "Montag", "Dienstag",
                           "Mittwoch", "Donnerstag",
                           "Freitag", "Sonnabend",
                           "Sonntag", 0};

void printAllDays(char** f) {
    int i = 0;
    while (*f)
        printf("%d: %s\n", i++, *f++);
}

int main(int argc, const char * argv[])
{
    printAllDays(dayNames);
    return 0;
}
```

```
0: Montag
1: Dienstag
2: Mittwoch
3: Donnerstag
4: Freitag
5: Sonnabend
6: Sonntag
```

Zeiger und Funktionen

- Parameterübergabe an das Hauptprogramm:
  `main(int argc, char* argv[])` oder auch
  `main(int argc, char** argv)`



Zeiger und Funktionen

- es gibt auch Zeiger auf Funktionen:

```
void f1(int i){
    printf("f1(%d)\n", i);
}
void f2(int i){
    printf("f2(%d)\n", i);
}

int main(int argc, const char * argv[])
{
    void (*fp)(int)=f1;    // von innen nach außen lesen: fp ist ein Zeiger auf
                           // eine Funktion von int nach void, f1 und f2 passen
    fp(12);  // am Aufruf ist nicht zu erkennen, ob fp ein Funktionsname
    fp=f2;   // oder ein Funktionszeiger ist, Deklaration und Benutzung nicht
    fp(22);  // konseqent benutzt
    return 0;
}
```

```
f1(12)
f2(22)
```

Zeiger und Funktionen

- es gibt auch Zeiger auf Funktionen:

```
void f1(int i){
    printf("f1(%d)\n", i);
}
void f2(int i){
    printf("f2(%d)\n", i);
}

int main()
{
    void (*fp)(int)=&f1;

    (*fp)(12);   // besser: konsequente Verwendung der Indirektion
    fp=&f2;
    (*fp)(22);
    return 0;
}
```

```
f1(12)
f2(22)
```

Zeiger und Funktionen

- Funktionszeiger manchmal schwer lesbar: **`man signal`**

```
NAME
     signal -- simplified software signal facilities

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <signal.h>

     void (*signal(int sig, void (*func)(int)))(int);

     or in the equivalent but easier to read typedef'd version:

     typedef void (*sig_t) (int);

     sig_t
     signal(int sig, sig_t func);
```

Strukturen

- heterogene Datensätze mit benannten Feldern

- Beispiel:

```
struct toy {
    char name[50];                          // nicht: char name[];
    int price;
    float size;
};

struct toy myToy;
```

Deklaration einer Nutzer-Struktur `toy` und Erzeugung einer Instanz dieser Struktur `myToy`

- `toy` ist nur ein **type tag (Bezeichnung)** für die Struktur zur späteren Referenzierung in weiteren Deklarationen und KEIN Typname

```
typedef struct toy Toy; // sogar identische Namen möglich
```

Strukturen

- Variablen können auch direkt in einer struct-Deklaration definiert werden:

```
struct toy {
      char name[50];
      int price;
      float size;
} yourToy, hisToy;
```

- Strukturvariablen können während der Definition (passend) initialisiert werden:

```
struct toy herToy = {"gameboy", 30, 3.5};
```

- Um auf ein Element einer Struktur zuzugreifen, wird der ».«-Operator benutzt

```
myToy.price = 100;
```

Union-Typen

- ein Union-Typ ist eine Struktur,
  von dem eine Variable (zu verschiedenen Zeiten) Werte unterschiedlichen Typs und Größe speichern kann
- jeder Komponente des Typs wird derselbe Speicherplatz zugeordnet
- Speicherplatzgröße wird durch die »längste« Komponente bestimmt

```
struct S {
     int i;
     double x;
} s;
```

s: i / x

```
union U {
     int i;
     double x;
} u;
```

u: i/x

Union-Typen

Beispiel:

```
union operand { /* Speicher, der Bitmuster als Werte */
                /* besitzt, als integer, long integer*/
                /* oder double interpretiert werden   */
                /* können                             */
    int    i;
    long   l;
    double d;
} o;
```

deklariert eine **`union`** mit Namen **`operand`** (auch nur Typ-Tag) und eine Instanz namens **`o`**

- auf Elemente kann wie bei **`structs`** zugegriffen werden:
  ```
  printf("%ld \n", o.l);
  ```
- die jeweils aktuelle korrekte Interpretation des Speicherplatzes muss in einem separaten Diskriminator [zumeist von **`enum`**-Typ] explizit verwaltet werden

Union-Typen

```
enum optype {INT, LONG, FLOAT};

struct operand {
  enum optype mytype;
  union variante {
      int i;
      float f;
      long d;
  } value;
} op;
```

```
// Initialisierung von op
...
switch (op.mytype) {
  case INT:
            printf("%d", op.value.i);
            break;
  case FLOAT:
            printf("%f", op.value.f);
            break;
  case LONG:
            printf("%ld", op.value.l);
            break;
  default:
            printf ("ERROR\n");
            break;
}
```

Zeiger und Strukturen

- Beispiel: Punkt

```
struct Punkt {
      float x, y, z;
} p;

struct Punkt *p_ptr;
p_ptr = &p; /* Adresse von p */
```

- Operator -> erlaubt den Zugriff auf Elemente einer Struktur, auf die der Zeiger zeigt:

```
p_ptr->x = 1.0;
/* Kurznotation für (*p_ptr).x= 1.0, Klammerung notwendig */
p_ptr->y -= 3.0;
```

Zeiger und Strukturen

- rekursive Datenstrukturen
  - Beispiel: Verkette Liste

```
struct ELEMENT {
      int value;
      struct ELEMENT *next;
};

struct ELEMENT n1, n2;
/* ohne struct-Angabe ist ELEMENT kein gültiger Typname */

n1.next = &n2;
```

value | *next → value | *next

n1 n2

Zeiger und Strukturen

- besser mit **`typedef`**

```
typedef struct ELEMENT {
        int value;
        struct ELEMENT *next; // hier ELEMENT noch kein Typname
} ELEMENT;                    // ab hier ist ELEMENT ein Typname

ELEMENT n1, n2;

n1.next = &n2;
```

Definition neuer Datentypen

- alle Typkonstrukte von C können beliebig tief verschachtelt verwendet werden
- ein (sinnloses, aber übersetzbares) Beispiel:

```
struct S {
    int* i;
    char n[12];
    union L {
        double x;
        int (*f)(int);
    } l, *pl;
};

struct S s[6];
union  L l;                                          // type scopes do not nest!

int main() {
                s[2].pl = &s[4].l;
                s[2].pl->x = 1.23;
                (*l.f)(3);     Thread 1: EXC_BAD_ACCESS (code=1, address=0x0)

                return 0;
}
```

Buffer Overflows

Am Beispiel `strcpy(char* dst, const char* src)`:

Voraussetzung: `dst` ist hinreichend groß,

Aber wie kann `dst` dimensioniert werden, wenn man mit potentiell beliebigen Quellstrings (z.B. aus dem Netz) rechnen muss !?

```
char buf[ichdenkedaswirdschonreichen]; /* ? */
strcpy(buf, source); /* danger ! */

/* dangerous: strcpy, strcat, sprintf, scanf, getwd, gets, ... */
```

Buffer Overflows

Immer zuvor die Länge des Quellstrings auswerten:

```
#include <stdio.h>
#define BUFSIZE 20
#define MAX_DYNAMIC_BUFFER_I_AM_WILLING_TO_ALLOCATE 100
void process(char[]);

int main(int c, char** v){
    if (c<2) return 0;
    char buf[BUFSIZE];
    char *bufptr=0;
    int l = strlen(v[1])+1;
    if (l <= BUFSIZE) {
          strcpy (buf, v[1]);
          process(buf);
    } else if (l <= MAX_DYNAMIC_BUFFER_I_AM_WILLING_TO_ALLOCATE) {
          bufptr = malloc(MAX_DYNAMIC_BUFFER_I_AM_WILLING_TO_ALLOCATE);
          strcpy(bufptr, v[1]);
          process(bufptr);
          free(bufptr);
    } else {
          fprintf(stderr, "cannot handle this length string\n");
          exit(-1);
    }
    return 0;
}
void process (char buffer[]) { printf ("%s\n", buffer); }
```

Die Programmiersprache C

4. Dynamische Objekte, Präprozessor, Bitoperationen, Bibliothek

Prof. Dr. sc. Joachim Fischer (Dr. Klaus Ahrens)
Institut für Informatik, Humboldt-Universität zu Berlin
SS 2014

Dynamische Objekte

C benutzt für die Allokierung von Objekten zur Laufzeit einen *Heap* wie Java

Aber:

- die bereitgestellten Operationen sind viel elementarer, man operiert direkt mit physischen Adressen und dem Hauptspeicher
- allokierter Speicher ist bestenfalls auf 0 initialisiert (`calloc`), sonst uninitialisiert (es gibt keine Konstruktoren)
- es gibt keine *garbage collection*, auch um die Freigabe von Speicher muss man sich selbst kümmern —> sonst entstehen sog. *memory leaks*
- man kann viel mehr falsch machen :-(

Dynamische Objekte

```
#include <stdlib.h>     /*Systembibliothek: Speicherverwaltung u. mehr */

struct datum {
  int jahr;          /* 4 Bytes */
  char monat[3];     /* 3 Bytes */
  int tag;           /* 4 Bytes */
};
```

Schritte von malloc:
1. sizeof: Größe des Objekts
2. Speicherplatzreservierung
3. Rückgabe eines typlosen Zeigers (**`void*`**) (zeigt auf Anfang des reserv. Bereichs)
4. entstehender Wert erhält einen Typ **`struct datum *`** (implizite Typumwandlung)

danach kann Speicherplatz benutzt werden

```
struct datum *t1 = malloc(sizeof(struct datum));

/* t1 zeigt auf eine nichtini                        auf der Halde */
/* wie groß ist die Speich                           herausrichtung */
```

ein Cast auf **`(struct datum*)`** [den man häufig sieht] ist hier **NICHT** erforderlich (Compiler „kennt“ **`malloc`** & Co.)

Dynamische Objekte

Speicherplatz anfordern mit `malloc`

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

- `malloc` liefert einen Zeiger auf einen **uninitialisierten** Speicherblock der Größe `size` auf dem *Heap* **ODER** `NULL`
- Anwendung:

```
int *ip = malloc(sizeof(int));
if (ip) … // Speicherplatz benutzen
else …    // Fehlerbehandlung
```

Dynamische Objekte

Speicherplatz anfordern mit `calloc`

```
void *calloc (size_t count, size_t size);
```

- **`calloc`** liefert einen Zeiger auf einen durchgängig mit 0 initialisierten Speicherblock der Größe **`count*size`** auf dem *Heap* **ODER** **`NULL`**
- Anwendung:

```
int *ifp = calloc(100, sizeof(int));
if (ifp) … // Speicherplatz benutzen
else …     // Fehlerbehandlung
```

Dynamische Objekte

Speicherplatz anfordern mit `realloc`

```
void *realloc (void *ptr, size_t size);
```

- **`realloc`** liefert einen Zeiger auf einen Speicherblock der Größe **`size`** und platziert dort das Objekt, welches zuvor über den (ebenfalls dynamisch allokierte) Zeiger **`ptr`** erreichbar war auf dem *Heap* **ODER `NULL`**
  - der Inhalt des Speicherblocks wird u.U. (partiell) kopiert
  - ein vormals mit `calloc` angeforderter Block wird beim Vergrößern **nicht** mit 0 aufgefüllt
- Anwendung:

```
// Vergroessern
ifp = realloc(ifp, 200);
if (ifp) … // ok
else …     // error
```

```
// Verkleinern
ifp = realloc(ifp, 50);
```

Dynamische Objekte

Speicherplatz freigeben mit **`free`**

```
void free (void *ptr);
```

- **`free`** gibt einen einen (zuvor dynamisch allokierten Speicherblock) der über **`ptr`** erreichbar ist, an den *Heap* zurück
- Anwendung:

```
free(ifp);

// don't use
ifp = NULL;
```

`free` auf eine Adresse die zuvor nicht dynamisch allokiert wurde:

mehrfaches **`free`**:

Zugriff über den Zeiger nach **`free`**:

Bitoperationen

&	AND
|	OR
^	XOR
~	Einer-Komplement
<<	Shift left
>>	Shift right

- Nicht verwechseln: & und && - & ist bitorientiertes AND, && logisches UND.
- Ähnlich für | und ||

- Unärer Operator
- Die Schiebeoperatoren schieben den linken Operanden bitweise um den Wert des rechten Operanden
- Der rechte Operand muss positiv sein. Die 'neuen' Bits werden mit '0' aufgefüllt (keine Vorzeichen-ausdehnung, kein *wrap around*)

Bitoperationen

- Beispiel:
  - Falls x = 00000010 (binär) oder 2 (dezimal):
    - **`x>>=2`** : x = 00000000 oder x = 0 (dezimal)
    - **`x<<=2`** : x = 00001000 oder x = 8 (dezimal)

- *shift left* ist (für vorzeichenlose Zahlen) zur Multiplikation mit 2 äquivalent
- *shift right* ist (für vorzeichenlose Zahlen) zur Division durch 2 äquivalent
- *shift* ist wesentlich schneller als die Multiplikation oder Division

Präprozessor

- Motivation
  - Substitution von sich häufig wiederholenden Mustern
  - Substitution wird vor der Compilation ausgeführt
  - Sollte Lesbarkeit des Codes verbessern und Code vereinfachen (aber nicht übertreiben !)
- Direktiven beginnen immer mit einem **#**

Präprozessor

```
#define <macro> <re
```

- definiert ein Makro:
  String <macro> soll durch <r … den

die Ersetzung erfolgt „blind“:

- ohne Kenntnis der Regeln von C (manchmal Vorteil, meist Nachteil)
- ohne Kontextwissen: Vorrang? Seiteneffekte?
- Fehlerausschriften des Compilers beziehen sich auf den substituierten Quelltext!

- Beispiele:

```
#define begin {
#define end }
#define max(A, B) ( (A) > (B) ? (A):(B))
```

  - dies definiert keine Funktion, sondern nur Text
  - A and B werden durch “aktuelle” Parameter ersetzt:

```
max(a,3)  —> max(a,3) ( (a) > (3) ? (a):(3))
```

Präprozessor

`#undef` `<name>` macht die Makrodefinition <name> rückgängig

`#include` fügt eine Datei mit in den Code ein

- zwei mögliche Formen:
  - **`#include`** <file> oder **`#include`** "file"
  - <file> Systemheader: suche in einem dem Compiler bekannten Verzeichnis (üblich bei **`gcc`** unter Unix ist **`/usr/include`**)
  - "file" sucht nach der Datei im aktuellen Verzeichnis (und entlang des Pfades aus der Option **`-I`**)
- Include-Dateien enthalten normalerweise C-Deklarationen und <u>keinen</u> (algorithmischen) C-Code
- es gibt jedoch KEINERLEI Vorgaben über Inhalt und Struktur von Include-Dateien

Präprozessor

`#if` <Conditional> <inclusion>

- **`#if`** evaluiert einen konstanten Integer-Ausdruck:
  0: lasse <inclusion> im Quelltext aus
  !0: schließe <inclusion> in den Quelltext ein
- es wird ein **`#endif`** benötigt, um das Ende zu signalisieren
- else Teil: **`#else`** and **`#elif`** -- else if

spezielle Form von **`#if`**

```
#ifdef <Symbol>                #if defined <Symbol>
```
-- if defined
```
#ifndef <Symbol>               #if !defined <Symbol>
```
-- if not defined

Präprozessor

Weitere Direktiven:

```
#error text of error message
```

- Abbruch der Übersetzung mit entsprechender Fehlermeldung

```
#if !defined __GNUC__
#error compile with GNU gcc
#endif
```

```
// leider nicht:
#if !sizeof(int)==4
#error need 32 bit
#endif
```

```
#line number "string"
```

- informiert den Präprozessor, dass die Zeilennummer der nächsten Zeile number sein soll und in welcher Datei mit Namen string diese Zeile auftritt

Präprozessor

even more magic :-)

[häufig in generiertem C-Code]

Stringization **#** macht in **#define**-Direktiven aus #X eine Zeichenkette "X"

```
#define O(X) #X

printf("%s\n",O(irgendwas auch {[$% () nur geschachtelt ));
```

```
irgendwas auch {[$% () nur geschachtelt
```

Token Pasting **##** kann in **#define**-Direktiven Teile zusammenfügen

```
#define GLUE(A, B) A##B

int GLUE(xyz,4) = 25;
printf("%d\n",xyz4);
printf("%d\n",GLUE(xyz,4));
```

```
25
25
```

Standardbibliotheken

Viele vorprogrammierte Funktionen sind verfügbar:

- `stdio.h`: Standard- Ein-/Ausgabe-Bibliothek

  Terminal- & Dateifunktionen `fprintf, scanf, sprintf, sscanf,` …
- `math.h`: mathematische Funktionen `sin, cos, log` …
  und Konstanten `M_E, M_PI, M_SQRT2` …
- `string.h`: Zeichenkettenoperationen
- `time.h`: Datum, Uhrzeit, Zeitmessung
- `assert.h`: Assertions `assert(i==0);`
- `errno.h`: Fehlercodes von Bibliotheksfunktionen

falls erfüllt: keine Wirkung
falls nicht erfüllt: Programmabbruch

```
Assertion failed: (i==0), function main,
file /Users/klaus/Xcode Sources/memerrors/
memerrors/main.c, line 126.
```

```
errno=ENOMEM;   // normalerweise nicht so, sondern als Effekt von malloc
perror("Systemfehler");
```

```
Systemfehler: Cannot allocate memory
```