
Block 3 (4.12.2023: NUR Diskussion/Fragen)

Schauen Sie sich vorab die Videos an!!

3.6 Strings

Strings = Array von `char`. Ende eines String: (Code-) Zahl 0 (nicht das Zeichen `'0'`, das hat Code-Zahl 48!).

```
char name[100];            /* up to 99 characters */
```

Stringfunktionen in `string.h` deklariert.
Bsp.:

```
strcpy(name, "Robert Smith");                /* copies text into string */
printf("length(%s)=%d\n", name, strlen(name));         /* prints length */
```

[Selbsttest]

Mit welcher Funktion können Sie einen String an den anderen hängen? Schauen Sie mit `man string` nach und geben Sie einen Beispielaufruf.

Nützlich: `sprintf(string, <format string>, ....)` (definiert in `stdlib.h`) geht wie `printf` aber schreibt in String anstatt auf Standardausgabe.

Hinweis: `valgrind` findet Speicherfehler, wie z.B. Schreiben über das Ende des reservierten Bereichs bei einem String/Array.

3.7 Strukturen, selbst-definierte Datentypen

Gruppiere mehrere Elemente in einen Datentyp:
Bsp.:

```
struct particle
{
  double        mass;         /* in kg                          */
  int         charge;         /* in units of e                  */
  double position[3];         /* position in space. in meters */
}
```

Variablen-Deklaration:

```
struct particle particle1;
```

Zugriff

```
particle1.mass = 9.109e-31;
particle1.charge = 1;
particle1.position[0] = -2.3e-3;
```

Bequem: eigene Datentypen. Schreibe: `typedef` gefolgt von einer "normalen" Deklaration, z.B.:

```
typedef double vector_t[3];                    /* new type 'vector_t' */
typedef struct particle particle_t;          /* new type 'particle_t' */
...

vector_t velocity;                  /* velocity is of type 'vector_t' */
particle_t electron;  /* variable 'electron' is of type 'particle_t' */
```

Konvention: sammle alle Typen in extra Header (.h) File.

Mit Strukturen kann man leicht Funktionen implementieren, die mehrere Daten zurückgeben, z.B.

```
particle_t initialise_atom()
{
  particle_t atom;
  atom.mass = 1;
  ...
  return(atom);
}
```

3.8 Pointer

Pointer (Zeiger) = Adresse einer Variable im Speicher.

Deklaration: `<type> *ptr` erzeugt `ptr` als Adresse einer Variablen des Typs `<type>`.

&-Operator gibt Adresse der Variablen: `& <variable>`.

`*ptr` = Inhalt der Variablen auf die `ptr` zeigt, d.h. man kann den Inhalt setzen durch `*prt= <expression>`. Bsp:

```
int number, *address;

number = 50;
address = & number;
*address = 100;
printf("%d\n", number);
```

ergibt: 100.
Arrays = Pointer, `int value[10]` ⇒ `value`= Adresse des Anfangs des Arrays, d.h. von `value[0]`. Beides `int value[10]` und `int *value2` = Pointer auf `int`, aber für `value` wird Array der Länge 10 reserviert und `value` zeigt auf den Array Anfang. `value2` erhält anfangs KEINEN Wert.

Zugriff: `value[5]` ist äquivalent zu `*(value+5)`.

Fall: Zeiger auf Struktur zeigt: Zugriff auf Elemente durch`->` Operator.

```
struct particle *atom;
...
atom->mass = 2.0;
```

(äquivalent `(*atom).mass=2.0`)
Pointer: verwendbar um komplexe Beziehungen zwischen Variablen herzustellen, z.B. um komplexe Datentypen zu bauen (Listen oder Bäume, s.u.).

Pointer: verwendbar um aus Funktion Daten ohne `return` zurückzugeben. (Nützlich, falls viele Variablen zurückgegeben werden)

```
void add_numbers(int n1, int n2, int *result_p)
{
  *result_p = n1+n2;
}
```

Hinweis: Pointer `result_p` kann in `add_numbers()`, nur den Inhalt des Speichers ändern, auf den `result_p` zeigt, nicht aber den (externen) Wert von `result_p`.

3.9 Dynamische Speicher Allozierung

Oft weiß man die Größe von Arrays nicht bei Compilieren oder bei Aufruf einer Funktion.
⇒ Arrays dynamisch mit `malloc(<number of bytes>)` (definiert in `stdlib.h`) allozieren. Benutze `sizeof(<data type>)` um Array-Größe festzulegen.
Bsp:

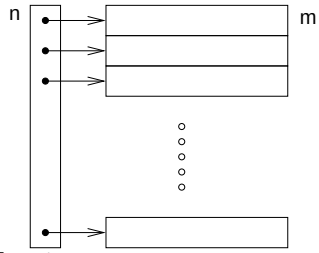
```
struct particle *atom2;
int num_atoms;
...
atom2 = (struct particle *) malloc(num_atoms*sizeof(struct particle));
```

Nun kann `atom2` wie normales Array benutzt werden.
Wenn Array nicht mehr benötigt: Freigabe

```
free(atom2);
```

Niemals vergessen, andernfalls kann Programm riesig anwachsen.

Allozierung von Matrizen variabler Größe in 2 Schritten.
Grunprinzip $n \times m$ Matrix: Erst Array von n Pointern allozieren, dann n mal Array von m `double` allozieren.



Beispiel: Anlegen einer `num_rows` $\times$ `num_columns` Matrix:

```
int num_rows, num_columns, row;
double **matrix;
...
matrix = (double **) malloc(num_rows*sizeof(double *));
for(row=0; row<num_rows; row++)
  matrix[row] = (double *) malloc(num_columns*sizeof(double));
```

Nie vergessen (!): Freigeben:

```
for(row=0; row<num_rows; row++)
  free(matrix[row]);
free(matrix);
```

[Selbsttest]

Wie deklarieren und legen Sie ein Array `name` von Strings an. Es soll `num_names` Namen mit jeweils maximaler Länge `name_length` geben?

4 Debugging

4.1 Software Engineering

Schreiben Sie Programme so, dass sie möglichst wenig Fehler enthalten

- Erst auf Papier planen:

  Problem in Teilprobleme zerlegen

  Datenstrukturen entwerfen

  Grundlegende Aufgaben (Subroutinen) identifizieren

- Programmierstil:

  Keine globalen Variablen

  Übersichtlicher Code (Einrücken, nicht mehr als 80 Zeichen/Zeile)

  "Sprechende" Variablennamen (`energy` und `force` statt `x1` und `x2`)

  Genügend Kommentare (Funktions, Variabel und Blockkommentare)

  Erst Spezialfälle programmieren, dann allgemeine Unterroutinen.

- Testen:

  Nach erfolgreichem Compilen: jedes Programmsegment nochmal durchlesen

  Jede Unterroutine einzeln Testen: mit Debugger nachvollziehen; mit Normal-, aber auch Grenz- und Sonderfälle aufrufen

  Unterroutinen Stück für Stück zu Hauptprogramm zusammensetzen

  Messgrößen beobachten (z.B. Gesamtenergie)

  Für bekannte Spezialfälle Vergleich der Ergebnisse

4.2 Debugger

Linux: *gdb*: a source-code debugger: Programm bei Ausführung "zusehen", jederzeit anhalten, alle Variablen inspizier und änderbar.

Beispielprogramm:

```
26 int main(int argc, char *argv[])
27 {
28     int t, *array, sum = 0;
29
30     array = (int *) malloc (100*sizeof(int));
31     for(t=0; t<100; t++)
32         array[t] = t;
33     for(t=0; t<100; t++)
34         sum += array[t];
35     printf("sum= %d\n", sum);
36     free(array);
37     return(0);
38 }
```

Zum Compilieren: Option `-g`

```
cc -o gdbtest -g gdbtest.c
```

Starten des Debuggers: In Shell `gdb <Programm>`:

```
gdb gdbtest
```

Kommandos durch Texteingabe.

Befehl `list` (kurz: `l`): Source code an aktueller Stelle ansehen:

```
(gdb) l
19
20        for (t=0; t<n; t++)
21          sum += array[t];
22        return(sum);
23      }
24
25
26      int main(int argc, char *argv[])
27      {
28          int t, *array, sum = 0;
(gdb)
```

Mehr Infos: `help line`.

Setzen von Breakpoints `break` (kurz `b`) + Zeilennumer oder Funktionsname

```
(gdb) b 33
Breakpoint 1 at 0x8048443: file gdbtest.c, line 33.
```

Breakpoints löschen: `delete`.

Programm starten: `run` (oder `r`), ggf. mit Programmargumenten.

```
gdb) r
Starting program: /home/hartmann/book4/programs/debugging/gdbtest 100

Breakpoint 1, main (argc=2, argv=0xbfdfcc94) at gdbtest.c:33
33          for(t=0; t<100; t++)
```

Variablen mit `print` (oder `p`) ansehen:

```
(gdb) p array
$1 = (int *) 0x8049680
(gdb) p array[99]
$2 = 99
```

Dauerhafte Anzeige: `display`
Wert Ändern: `set variable` (oder `set var`)

```
(gdb) set var array[99]=98
```

Programm schrittweise fortsetzen: `next` (oder `n`):

```
(gdb) n
34              sum += array[t];
```

Funktionsaufrufe = eine Zeile.
Um in Funktion hineinzuschauen: `step` (oder `s`)
Ausführung bis zum nächsten Breakpoint/Ende: `continue` (oder `c`)

```
(gdb) c
Continuing.
sum= 4949

Program exited normally.
```

Breakpoints mit Bedingung (sinnvoll innerhalb Schleifen): `condition`

```
(gdb) delete 1
(gdb) b 34
Breakpoint 2 at 0x8048477: file gdbtest.c, line 34.
(gdb) condition 2 (t==50)
(gdb) r
```

```
Starting program: /home/hartmann/book4/programs/debugging/gdbtest

Breakpoint 2, main (argc=1, argv=0xbff8fe34) at gdbtest.c:34
34              sum += array[t];
(gdb) print t
$1 = 50
```

Ausgeben eigener Datenstrukturen oder bestimmte Tests: dazu Funktion implementieren und jederzeit mit `call` aufrufen.

```
int chksum(int n, int *array)
{
  int sum=0, t;
  for (t=0; t<n; t++)
    sum += array[t];
  return(sum);
}
```

```
(gdb) call chksum(100, array)
$2 = 4950
```

Umfassende Informationen: `info`, z.B. `info break` `info registers`, `info float`, `info variables`, `info types`.

Graphischer Debugger: Breakpoint durch anklicken etc: `ddd`.

4.3 Memory Checker

Einige Fehler: haben manchmal eine Auswirkung (z.B. `Segmentation fault`) manchmal nicht, Verhalten unvorhersehbar:
Grund: Fehler beim Zugriff auf Speicher, z.B.:

- Zugriff hinter Array Grenzen
- Zugriff auf bereits freigegeben Speicher
- Vergessen mit `malloc` allozierten Speicher wieder freizugeben ("Memory Leak")

Beispiel (`memerror1.c`)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char *argv[])
```

```
5  {
6      int t, *array, sum = 0;
7
8      array = (int *) malloc (99*sizeof(int));
9      for(t=0; t<100; t++)
10         array[t] = t;
11     for(t=0; t<100; t++)
12         sum += array[t];
13     printf("sum= %d\n", sum);
14     free(array);
15     return(0);
16 }
```

```
cc -o memerror1 memerror1.c -g
```

Program läuft aber komplett durch:

```
sum= 4950
```

Aber:

```
1  [hartmann@comphy01 debugging]$ valgrind memerror1
2   Memcheck, a memory error detector.
3   Copyright (C) 2002-2005, and GNU GPL'd, by Julian Seward et al.
4   Using LibVEX rev 1575, a library for dynamic binary translation.
5   Copyright (C) 2004-2005, and GNU GPL'd, by OpenWorks LLP.
6   Using valgrind-3.1.1, a dynamic binary instrumentation framework.
7   Copyright (C) 2000-2005, and GNU GPL'd, by Julian Seward et al.
8   For more details, rerun with: -v
9
10  Invalid write of size 4
11     at 0x8048423: main (memerror1.c:10)
12   Address 0x402A1B4 is 0 bytes after a block of size 396 alloc'd
13     at 0x4004405: malloc (vg_replace_malloc.c:149)
14     by 0x80483FF: main (memerror1.c:8)
15
16  Invalid read of size 4
17     at 0x8048447: main (memerror1.c:12)
18   Address 0x402A1B4 is 0 bytes after a block of size 396 alloc'd
19     at 0x4004405: malloc (vg_replace_malloc.c:149)
20     by 0x80483FF: main (memerror1.c:8)
21 sum= 4950
22
23  ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 13 from 1)
24  malloc/free: in use at exit: 0 bytes in 0 blocks.
```

```
25  malloc/free: 1 allocs, 1 frees, 396 bytes allocated.
26  For counts of detected errors, rerun with: -v
27  All heap blocks were freed -- no leaks are possible.
28 [hartmann@comphy01 debugging]$
```

Man kann Debugger aufrufen lassen, wenn Fehler auftaucht: `--db-attach=yes`

Programm ohne `free` Kommando (`memerror2.c`)
→ Memory Leak
→

```
[hartmann@comphy01 debugging]$ valgrind  memerror2
   .....

 searching for pointers to 1 not-freed blocks.
 checked 51,748 bytes.

 LEAK SUMMARY:
    definitely lost: 400 bytes in 1 blocks.
      possibly lost: 0 bytes in 0 blocks.
    still reachable: 0 bytes in 0 blocks.
         suppressed: 0 bytes in 0 blocks.
 Use --leak-check=full to see details of leaked memory.
[hartmann@comphy01 debugging]$
```

TAKE HOME WORK

Besorgen Sie sich (falls noch nicht vorhanden) das des C Tutorials vom StudIP, und lesen sie die restlichen Seiten. Vollziehen sie die Programmteile praktisch nach!