Hochschule Bochum
Bochum University
of Applied Sciences

Angewandte Informatik

C im Schnelldurchgang

WS 2011/12, ab 1. Dezember 2011

Prof. Dr. Peter Gerwinski

Inhaltsverzeichnis

Stand: 3. Januar 2012

1 Grundlagen

1.1 Hallo, Welt!

Beispiel: Textausgabe – hello.c

```
#include <stdio.h>

int main (void)
{
  printf ("Hello, world!\n");
  return 0;
}
```

Präprozessor: System-Include-Datei

Zeilenschaltung

- Hauptprogramm: Funktion `main()` vom Type `int` (ganzzahlig)
- Rückgabewert: 0 = Erfolg

Programm compilieren:

```
gcc -Wall -O hello.c -o hello
```

alle Warnungen aktivieren

Optimierung

Quelltext

Ausgabedatei

- `-O2`, `-O3`: stärker optimieren
- `-E`: Präprozessor-Ausgabe
- `-S`: Assembler-Ausgabe

1.2 Variablen, Zuweisungen, Zeiger, formatierte Ein- und Ausgabe

Beispiel: Zeitangaben umrechnen – time.c

```
#include <stdio.h>

int main (void)
{
  int s, m, h;
  printf ("Sekunden: ");
  scanf ("%d", &s);
  h = s / 3600;
  s %= 3600;
  m = s / 60;
  s %= 60;
  printf ("%02d:%02d:%02d\n", h, m, s);
  return 0;
}
```

Adresse der Variablen

Ganzzahl-Division (mit Abrunden)

`%`-Operator: Rest bei Division („modulo“)

Format: Dezimalzahl, 2 Stellen, mit Nullen auffüllen

kein Format: Doppelpunkt wird direkt ausgegeben

- Format-Spezifikationen innerhalb des Ausgabe-Strings
  werden durch entsprechend formatierte Parameterwerte ersetzt.

  `%d`: Dezimalzahl, linksbündig
  `%3d`: Dezimalzahl, rechtsbündig in Feld der Länge 3
  `%03d`: dasselbe, von links mit Nullen aufgefüllt

- Format-Spezifikationen bei der Eingabe:
  Eingabe in bestimmtem Format erwarten
  und auf den Variablen ablegen, auf die die Parameter zeigen
- Ein- und Ausgabe: Die Bedeutung der Parameter hängt von der Format-Spezifikation ab.
- Variablendeklaration: erst der Typ, dann Name(n)
- „=“: Zuweisung des Ausdrucks auf der rechten Seite an die linke Seite
- „`%`=“: Anwendung des Operators (hier: `%`), danach Zuweisung an die linke Seite:
  `a += b;` entspricht `a = a + b;`
  `a %= b;` entspricht `a = a % b;`
- Jede Zuweisung ist selbst ein (zuweisbarer) Ausdruck:
  `a = b = c;`

1.3 Funktionen

Beispiel: Funktionen, Wert- und Variablenparameter – functions.c

```
#include <stdio.h>

#define SOME_NUMBER 42

void set_value (int *x)
{
  *x = SOME_NUMBER;
}

void print_value (int x)
{
  printf ("%d\n", x);
}

int main (void)
{
  int y;
  set_value (&y);
  print_value (y);
  return 0;
}
```


- Funktionsdeklaration: Rückgabetyp, Name, Parameter in Klammern
- C kennt nur Wert-, keine Variablenparameter.
  Stattdessen übergibt man Zeiger auf Variablen als Werte.
- Rückgabetyp „`void`“: Funktion, die „nichts“ zurückgibt

1.4 Verzweigungen und Schleifen

Beispiel: Primzahltest – prime.c

```
#include <stdio.h>

int main (void)
{
  int p, n;
```

```
  printf ("Ihre Zahl: ");
  scanf ("%d", &p);
  n = 2;
  while (n < p)     <-- oder:  for (n = 2; n < p; n++)
    {
      if (p % n == 0)
        {
          printf ("%d ist keine Primzahl.\n", p);
          return 0;
        }
      n++;
    }
  printf ("%d ist eine Primzahl.\n", p);
  return 0;
}
```

- `while`, `if` und `for` beziehen sich jeweils nur auf einen nachfolgenden Befehl.
  Mehrere Befehle muß man mit geschweiften Klammern zu einem einzigen (Block) zusammensetzen.
- „`n++`“: Ausdruck mit dem Wert `n`; anschließend erhöhe `n` um 1
  „`++n`“: erhöhe `n` um 1; *danach* Ausdruck mit dem (neuen) Wert `n`;
- Verbesserungspotential: Zahlen kleiner als 2 korrekt behandeln
  (1 ist *keine* Primzahl!)

Beispiel: Zinseszinsen – interest.c

```
#include <stdio.h>

int main (void)
{
  double k, z;     <-- Fließkommazahlen (doppelt genau)
  int lz, j;
  printf ("Kapital in EUR: ");
  scanf ("%lf", &k);
  printf ("Zinsen in %%: ");
  scanf ("%lf", &z);
  printf ("Laufzeit in Jahren: ");
  scanf ("%d", &lz);
  for (j = 1; j <= lz; j++)
    {
      k += k * z / 100.0;
      printf ("Kapital nach %d Jahr(en): %.2lf\n", j, k);
    }
  return 0;
}
```

- Prozentzeichen ausgeben: im Format-String verdoppeln
- Eingabe von Fließkommazahlen: `%lf` für doppelt genau, `%f` für einfach genau (`float`)
- Ausgabe von Fließkommazahlen:
  `%lf`: linksbündig
  `%.2lf`: linksbündig, 2 Nachkommastellen
  `%10.2lf`: rechtsbündig in Feld der Länge 10, 2 Nachkommastellen

Beispiel: Kredit-Rückzahlung loan.c

```
#include <stdio.h>

int main (void)
{
  double k, z, m, s;
  int j;
  printf ("Kapital in EUR: ");
  scanf ("%lf", &k);
  printf ("Zinsen in %%: ");
  scanf ("%lf", &z);
  printf ("Jährliche Rate: ");
  scanf ("%lf", &m);
  s = 0.0;
  for (j = 1; k > 0.01; j++)
    {
      double dk = m - k * z / 100.0;
      if (dk >= k)
        {
          printf ("Schlußrate nach %d Jahr(en): %.2lf\n", j, k);
          s += k;
        }
      else
        s += m;
      k -= dk;
      if (k > 0)
        printf ("Kapital nach %d Jahr(en): %.2lf\n", j, k);
    }
  printf ("Insgesamt gezahlt: %.2lf\n", s);
  return 0;
}
```

- Die `for`-Schleifenabbruchbedingung ist unabhängig von Initialisierung und Iteration.
- Verbesserungspotential: Endlosschleife vermeiden, indem man abbricht, sobald das Restkapital über das Zehnfache des ursprünglichen Kapitals angestiegen (statt abgesunken!) ist

1.5 Logische Operationen

Beispiel: Rechnen mit exakten Brüchen – fractions.c

```
#include <stdio.h>

int main (void)
{
  int a, b, negative;
  printf ("Zähler: ");
  scanf ("%d", &a);
  printf ("Nenner: ");
  scanf ("%d", &b);
  negative = b < 0;        <-- Wert des Vergleich-Ausdrucks: 0 oder 1
  if (negative)            <-- Bedingung trifft zu, wenn ≠ 0.
    {
      a = -a;
      b = -b;
    }
  if (b)                   <-- oder: if (b != 0)
    {                      <-- Zuweisung innerhalb des Ausdrucks
      int r;
      if ((r = a % b) && a > b)
        printf ("Der Bruch lautet: %d %d/%d.\n", a / b, r, b);
      else if (!r)         <-- Verneinung: 0 wird 1, alles ≠ 0 wird 0
        printf ("Das ist eine ganze Zahl: %d.\n", a / b);
      else
        printf ("Der Bruch lautet: %d/%d.\n", a, b);
    }
  else
    printf ("Division durch Null ist nicht zulässig!\n");
}
```

- Und-Verknüpfung: `&&`
  Bedingung erfüllt, wenn beide Teilbedingungen zutreffen.
- Oder-Verknüpfung: `||`
  Bedingung erfüllt, wenn mindestens eine der Teilbedingungen zutrifft.
- Vergleiche: `==`, `!=`, `<=`, `>=`, `<`, `>`, ...
- Keine Vergleiche, sondern Zuweisungen: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, ...

1.6 Der Verzweigungsoperator ?:

Beispiel: max.c

```
#include <stdio.h>

int main (void)
{
  int a, b, max;
  printf ("Erste Zahl: ");
  scanf ("%d", &a);
  printf ("Zweite Zahl: ");
  scanf ("%d", &b);
  max = a > b ? a : b;
  printf ("Die größere der beiden Zahlen ist %d.\n", max);
}
```

- Äquivalent zu `if` mit `else`
- Da immer ein Wert zurückgegeben werden muß, gibt es keine Form ohne `else`-Zweig.
- Kann auch geschachtelt werden

1.7 Feldvariablen

Eine Feldvariable (Array) ermöglicht es, mehrere gleichartige Variaben über einen gemeinsamen Namen plus einen Index ansprechen.

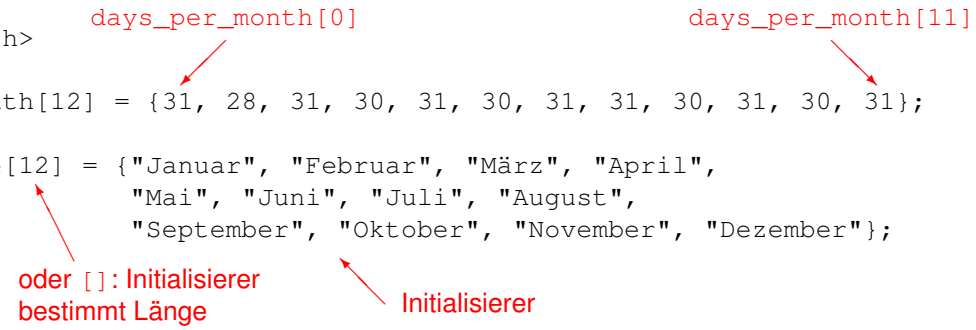
Beispiel: year.c

```
#include <stdio.h>

int days_per_month[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

char *month_name[12] = {"Januar", "Februar", "März", "April",
                        "Mai", "Juni", "Juli", "August",
                        "September", "Oktober", "November", "Dezember"};

int main (void)
{
  int day, d, month;
  printf ("Tag Nr.: ");
  scanf ("%d", &day);
  d = day;
  month = 0;
  while (d > days_per_month[month])
    {
      d -= days_per_month[month];
      month++;
    }
  printf ("Der %d. Tag im Jahr ist der %d. %s.\n",
          day, d, month_name[month]);
  return 0;
}
```



- Aufgabe: Schaltjahre miteinbeziehen
  Musterlösung: year2.c
- Aufgabe: unzulässige Eingaben abfangen

1.8 Literatur

- `printf`, `scanf`: Dokumentation des Compiler-Herstellers,
  z. B. für GCC unter Debian GNU/Linux:

  ```
  sudo apt-get install manpages-dev
  man 3 printf
  ```

- C-Programmierung: z. B. WikiBooks: http://de.wikibooks.org/wiki/C-Programmierung

2 Zahlensysteme und Bit-Operationen

2.1 Dezimalsystem

- Basis: 10
- Gültige Ziffern: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

```
137
```

Einer: $7 \cdot 10^0$
Zehner: $3 \cdot 10^1$
Hunderter: $1 \cdot 10^2$

```
  137
+  42
 ----
  179
```

$137_{10} = 1 \cdot 10^2 + 3 \cdot 10^1 + 7 \cdot 10^0 = 100 + 30 + 7 = 137$

2.2 Hexadezimalsystem

- Basis: 16
- Gültige Ziffern: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

```
137
```

$7 \cdot 16^0$
$3 \cdot 16^1$
$1 \cdot 16^2$

```
  A380
+ B747
  1----
 15AC7
```

$137_{16} = 1 \cdot 16^2 + 3 \cdot 16^1 + 7 \cdot 16^0 = 256 + 48 + 7 = 311$

- Schreibweise in C: `0x137`

2.3 Oktalsystem

- Basis: 8
- Gültige Ziffern: 0, 1, 2, 3, 4, 5, 6, 7

```
137
```

$7 \cdot 8^0$
$3 \cdot 8^1$
$1 \cdot 8^2$

```
  137
+  42
 -111-
  201
```

$137_8 = 1 \cdot 8^2 + 3 \cdot 8^1 + 7 \cdot 8^0 = 64 + 24 + 7 = 95$
$42_8 = 4 \cdot 8^1 + 2 \cdot 8^0 = 32 + 2 = 34$
$201_8 = 2 \cdot 8^2 + 0 \cdot 8^1 + 1 \cdot 8^0 = 128 + 1 = 129$

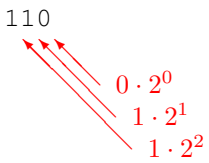
- Schreibweise in C: `0137`
- Rechner für beliebige Zahlensysteme: GNU bc

```
$ bc
ibase=8
137
95
obase=10
137 + 42
201
```

Eingabe zur Basis 8
Ausgabe zur Basis 10
Eingabe zur Basis 8 ($10_8 = 8$)
Ausgabe zur Basis 8

2.4 Binärsystem

- Basis: 2
- Gültige Ziffern: 0, 1




```
  110
+ 1100
------
  11
 10010
```

$110_2 = 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 4 + 2 + 0 = 6$

- Binär-Zahlen ermöglichen es, elektronisch zu rechnen ...
- und mehrere „Ja/Nein“ (Bits) zu einer einzigen Zahl zusammenzufassen.
- Oktal- und Hexadezimal-Zahlen lassen sich ziffernweise in Binär-Zahlen umrechnen.

000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

$1101011_2 = 153_8 = 6\mathrm{B}_{16}$

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

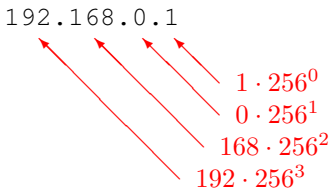
Beispiel: Oktal-Schreibweise für Unix-Zugriffsrechte

`-rw-r-----` $= 0\,110\,100\,000_2 = 640_8$ `$ chmod 640 file.c`

`-rwxr-x---` $= 0\,111\,101\,000_2 = 750_8$ `$ chmod 750 subdir`

2.5 IP-Adressen (IPv4)

- Basis: 256
- Gültige Ziffern: 0 bis 255, getrennt durch Punkte
- Kompakte Schreibweise für Binärzahlen mit 32 Ziffern (Bits)



$192.168.0.1_{256} = 11000000\,10101000\,00000000\,00000001_2$

2.6 Bit-Operationen

```
  0110       0110       0110       0110                  0110
+ 1100     | 1100     & 1100     ^ 1100     ~ 1100   >>    2
 -----      -----      -----      -----      -----      -----
 10010      1110       0100       1010       0011       0001
```

Addition | Oder | Und | Exklusiv-Oder | Negation | Bit-Verschiebung

```
  01101100           01101100           01101100
| 00000010         & 11110111         ^ 00010000
 ---------          ---------          ---------
  01101110           01100100           01111100
```

Bit gezielt setzen | Bit gezielt löschen | Bit gezielt umklappen

- Bits werden häufig von rechts und ab 0 numeriert (hier: 0 bis 7), um die Maskenerzeugung mittels Schiebeoperatoren zu erleichtern.
- Die Bit-Operatoren (z. B. `&` in C) wirken jeweils auf alle Bits der Zahlen. Die logischen Operatoren (z. B. `&&` in C) prüfen die Zahl insgesamt auf $\neq 0$. Nicht verwechseln!

```
6 & 12 == 4
6 && 12 == 1
```

Anwendung: Bit 2 (also das dritte Bit von rechts) in einer 8-Bit-Zahl auf 1 setzen:

```
  00000001           01101100
<<        2        | 00000100
 ---------          ---------
  00000100           01101100
```

Maske für Bit 2 | Bit gezielt setzen

- Schreibweise in C: `a |= 1 << 2;`

Anwendung: Bit 2 in einer 8-Bit-Zahl auf 0 setzen:

```
  00000001                              01101100
<<        2          ~ 00000100       & 11111011
 ---------          ---------          ---------
  00000100           11111011           01101000
```

Maske zum Löschen von Bit 2 erzeugen | Bit gezielt löschen

- Schreibweise in C: `a &= ~(1 << 2);`

Anwendung: Bit 2 aus einer 8-Bit-Zahl extrahieren:

```
  00000001           01101100           00000100
<<        2        & 00000100        >>        2
 ---------          ---------          ---------
  00000100           00000100           00000001
```

Maske für Bit 2 | Bit 2 isolieren | in Zahl 0 oder 1 umwandeln

- Schreibweise in C: `x = (a & (1 << 2)) >> 2;`

Beispiel: Netzmaske für 256 IP-Adressen

```
  192.168.  1.123   <-- IP-Adresse eines Rechners
& 255.255.255.  0   <-- Netzmaske: 255 = 11111111_2
 ---------------
  192.168.  1.  0   <-- IP-Adresse des Sub-Netzes
```

Beispiel: Netzmaske für 8 IP-Adressen

```
  192.168.  1.123   <-- IP-Adresse eines Rechners      01111011
& 255.255.255.248   <-- Netzmaske                   & 11111000
 ----------------                                    ---------
  192.168.  1.120   <-- IP-Adresse des Sub-Netzes     01111000
```

2.7 Bit-Arrays

Der kleinste C-Datentyp, `char`, ist laut Sprachstandard mindestens 8 Bit groß. (Meistens ist er genau 8 Bit groß; auf einigen Signalprozessoren (DSP) ist er größer.)

Wenn man ein großes Array von Variablen benötigt, die nur die Werte 0 oder 1 annehmen können, empfiehlt es sich, die in einem `char` enthaltenen Bits einzeln anzusprechen, anstatt für jede 0 oder 1 ein komplettes Byte zu reservieren.

Beispiel: bit_array.c

Die folgenden „Setter-“ und „Getter“-Funktionen illustrieren, wie man 800 Bits in einem Array von 100 `char`-Variablen unterbringt und über einen von 0 bis 799 laufenden Index anspricht.

```
#include <limits.h>  <-- enthält: #define CHAR_BIT 8

#define BUF_SIZE 100

unsigned char buffer[BUF_SIZE];

void set_bit (int index)
{
  buffer[index / CHAR_BIT] |= 1 << index % CHAR_BIT;
}

void clear_bit (int index)
{
  buffer[index / CHAR_BIT] &= ~(1 << index % CHAR_BIT);
}

int get_bit (int index)
{
  return (buffer[index / CHAR_BIT] & (1 << index % CHAR_BIT)) != 0;
}
```

2.8 Zweidimensionale Bit-Arrays

C kennt keine zweidimensionalen Arrays. Stattdessen verwendet man Arrays von Arrays:

```
char buffer[60][100];

buffer[59][99] = 1;
```

Achtung: Die Schreibweise `buffer[59, 99]` ist zwar ebenfalls zulässiges C, bedeutet aber etwas völlig anderes als `buffer[59][99]`.

Das Beispielprogramm bit_array_2.c illustriert eine Möglichkeit, das oben behandelte eindimensionale Bit-Array zu einem zweidimensionalen zu erweitern: Wir ersetzen den Bit-Puffer, ein Array von `char`-Variablen, durch ein Array von Bit-Puffern, also ein Array von Array von `char`-Variablen.

Alternativ kann man auch die Einträge der zweidimensionalen „Tabelle“ eindimensional durchnumerieren. Man ersetzt also Indexpaare

(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)
(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)
(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	(3,8)
(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)	(4,8)
(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)	(5,8)
(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)	(6,8)

durch einen einzigen fortlaufenden Index:

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48

Wenn wir die Indices nicht bei 1, sondern – wie in C üblich – bei 0 beginnen lassen, erhalten wir den fortlaufenden Index i aus den zweidimensionalen Indices x und y durch die folgende Formel:

$$i \;=\; y \cdot \text{Zeilenbreite} + x$$

Die Umkehrung dieser Formel geschieht übrigens durch Division mit Rest:

$$\begin{aligned} y &= i \,/\, \text{Zeilenbreite} \\ x &= i \,\%\, \text{Zeilenbreite} \end{aligned}$$

Dieser Mechanismus ist genau derselbe wie der, mit dem wir die einzelnen Bits in einem Byte ansprechen (mit `CHAR_BIT` anstelle der Zeilenbreite). So gesehen können wir also den eindimensionalen Bit-Speicher auch als ein zweidimensionales Array mit den Indizes „Byte-Nr.“ und „Bit-Position innerhalb des Bytes“ auffassen.

Die folgenden Setter- und Getter-Funktionen sprechen auf diese Weise ein eindimensionales Array als ein zweidimensionales an:

```
#define ROWS 60
#define COLUMNS 100

unsigned char buffer[ROWS * COLUMNS];

void set_bit (int x, int y, char value)
{
  buffer[y * COLUMNS + x] = value;
}

char get_bit (int x, int y, int index)
{
  return buffer[y * COLUMNS + x];
}
```

Das Beispielprogramm bit_array_2a.c verwendet diese alternative Möglichkeit, um das oben behandelte eindimensionale Bit-Array zu einem zweidimensionalen zu erweitern: Wir schreiben zweidimensionale Setter und Getter für das bereits vorhandene eindimensionale Bit-Array.

2.9 Anwendung: Monochrom-Grafikformate

Beispiel: Das PBM-Grafikformat: draw_image.c, test.pbm

Zum Abspeichern von monochromen Bildern, die also nur aus schwarzen und weißen Pixeln bestehen, gibt es zahlreiche Dateiformate. Eins der einfachsten ist das binäre PBM-Format (PBM = portable bitmap), bei dem die Bildinformationen i. w. genauso gespeichert werden wie in dem oben beschriebenen Bit-Array.

Der Inhalt einer binären PBM-Datei ist

- die Kennung „`P4`“,
- Trennzeichen (Leerzeichen, Tabulatoren oder Zeilenschaltungen),
- danach die Breite als Dezimalzahl,
- weitere Trennzeichen,
- danach die Höhe als Dezimalzahl,
- genau 1 Trennzeichen
- und zuletzt die eigentlichen Bilddaten, binär abgespeichert.

Bzgl. der Bytes geschieht dies in demselben Format wie in unserer Variablen `buffer`: (x, y) entspricht `Index = y * Breite + x`, dieser wiederum dem Byte Nr. `Index / 8`.

Die Bits innerhalb der Bytes werden von links nach rechts abgespeichert, also genau in der umgekehrten Reihenfolge wie in unserem Bit-Array. Um ein einzelnes Bit anzusprechen, verwenden wir also nicht `1 << Index % 8`, sondern `(1 << 7) >> Index % 8`.

(Zum Vergleich enthalten test.png, und test.jpg dasselbe monochrome Bild in gebräuchlicheren Datenformaten.)

Beispiel: Das XBM-Grafikformat: read_image.c, test.xbm

Die Datei enthält einen C-Quelltext, zunächst die Breite und Höhe als `#define`s, danach die eigentlichen Bilddaten, als Quelltext eines initialisierten C-Arrays abgespeichert und in demselben Format wie in unserem Buffer

Die Bits innerhalb der Bytes werden von rechts nach links abgespeichert. Um ein einzelnes Bit anzusprechen, verwenden wir `1 << Index % 8`.

3 Vom Quelltext zum ausführbaren Programm

3.1 Der Präprozessor

Der erste Schritt beim Compilieren eines C-Programms ist das Auflösen der sogenannten Präprozessor-Direktiven und -Macros.

```
#include "stdio.h"
```

bewirkt, daß aus Sicht des Compilers anstelle der Zeile der Inhalt der Datei `stdio.h` im C-Quelltext erscheint. Dies ist zunächst unabhängig von Bibliotheken und auch nicht auf die Programmiersprache C beschränkt.

Beispiel: Die Datei maerchen.txt enthält:

```
Es war einmal
#include "hexe.txt"
Die lebte in einem Wald.
```

Die Datei hexe.txt enthält:

```
eine kleine Hexe.
```

Der Aufruf

```
cpp -P maerchen.txt
```

produziert die Ausgabe

```
Es war einmal
eine kleine Hexe.
Die lebte in einem Wald.
```

`cpp` ist der C-Präprozessor. Die Option `-P` unterdrückt Herkunftsangaben, die normalerweise vom Compiler verwendet werden, um Fehlermeldungen den richtigen Zeilen in den richtigen Dateien zuordnen zu können. Ohne das `-P` lautet die Ausgabe:

```
# 1 "maerchen.txt"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "maerchen.txt"
Es war einmal
# 1 "hexe.txt" 1
eine kleine Hexe.
# 3 "maerchen.txt" 2
Die lebte in einem Wald.
```

(Alternativ zu `cpp` kann man auch `gcc -xc -E` verwenden. Das `-E` bewirkt, daß nur der Präprozessor, aber nicht der Compiler aufgerufen wird. Mit `-xc` teilen wir `gcc` mit, daß er die .txt-Datei wie einen C-Quelltext behandeln soll. Ohne `-xc` hält `gcc` die .txt-Datei für eine Bibliothek, die an den Linker übergeben werden soll (siehe unten), jedoch nicht an den Präprozessor übergeben werden darf – auch nicht bei `-E`.)

Nichts anderes geschieht, wenn man das klassische hello.c compiliert:

```
#include <stdio.h>

int main (void)
{
  printf ("Hello, world!\n");
  return 0;
}
```

Die Datei stdio.h ist wesentlich länger als hexe.txt in dem o. a. Beispiel, und sie ruft weitere Include-Dateien auf, so daß wir insgesamt auf über 15000 Zeilen Quelltext kommen. Die spitzen Klammern anstelle der Anführungszeichen bedeuten, daß es sich um eine Standard-Include-Datei handelt, die nur in den Standard-Include-Verzeichnissen gesucht werden soll, nicht jedoch im aktuellen Verzeichnis.

3.2 Externe Funktionen

Tatsächlich ist von den über 15000 Zeilen aus stdio.h nur eine einzige relevant, nämlich:

```
extern int printf (__const char *__restrict __format, ...);
```

Dies ist die Deklaration einer Funktion, die sich von einer „normalen“ Funktionsdefinition nur wie folgt unterscheidet:

- Die Parameter `__const char *__restrict __format, ...` heißen etwas ungewöhnlich.
- Der Funktionskörper `{ ... }` fehlt. Stattdessen folgt auf die Kopfzeile direkt ein Semikolon `;`.
- Der Deklaration ist das Schlüsselwort `extern` vorangestellt.

Dies bedeutet für den Compiler: „Es gibt diese Funktion und sie sieht aus, wie beschrieben. Benutze sie einfach, und kümmere dich nicht darum, wer die Funktion schreibt.“

Wenn wir tatsächlich nur `printf()` benötigen, können wir also die Standard-Datei stdio.h durch eine eigene ersetzen, die nur die o. a. Zeile `extern int printf (...)` enthält. (Dies ist in der Praxis natürlich keine gute Idee, weil nur derjenige, der die Funktion `printf()` geschrieben hat, den korrekten Aufruf kennt. In der Praxis sollten wir immer diejenige Include-Datei benutzen, die gemeinsam mit der tatsächlichen Funktion ausgeliefert wurde.)

3.3 Präprozessor-Makros

Mit `#define` kann man sog. Makros definieren, die bei Benutzung durch einen Text ersetzt werden.

Beispiel: rechnen.c

```
#include "stdio.h"

#define VIER 2 + 2

int main (void)
{
  printf ("Drei mal vier = %d\n", 3 * VIER);
  return 0;
}
```

Genau wie bei `#include` nimmt der Präprozessor auch bei `#define` eine rein textuelle Ersetzung vor, ohne sich um den Sinn des Ersetzten zu kümmern. Hier z. B. sieht man mit `gcc -E rechnen.c`, daß die Ersetzung des Makros `VIER` wie folgt lautet:

```
  printf ("Drei mal vier = %d\n", 3 * 2 + 2);
```

Der C-Compiler wendet die Regel „Punktrechnung geht vor Strichrechnung“ an und erfährt überhaupt nicht, daß das `2 + 2` aus einem Makro enstanden ist.

Um derartige Effekte zu vermeiden, setzt man arithmetische Operationen innerhalb von Makros in Klammern:

```
#define VIER (2 + 2)
```

(Es ist in den meisten Situationen üblich, Makros in `GROSSBUCHSTABEN` zu benennen, um darauf hinzuweisen, daß es sich eben um einen Makro handelt.)

3.4 Compiler und Linker

Beispiel: philosophy.c, answer.c, answer.h

Das Programm philosophy.c verwendet eine Funktion `answer()`, die in der Datei answer.h extern deklariert ist.

Der „normale“ Aufruf

```
gcc -Wall -O philosophy.c -o philosophy
```

liefert die Fehlermeldung:

```
/tmp/ccr4Njg7.o: In function 'main':
philosophy.c:(.text+0xa): undefined reference to 'answer'
collect2: ld returned 1 exit status
```

Diese stammt nicht vom Compiler, sondern vom Linker. Das Programm ist syntaktisch korrekt und wird auch korrekt in eine Binärdatei umgewandelt (hier: /tmp/ccr4Njg7.o). Erst beim Zusammenbau („Linken“) der ausführbaren Datei (philosophy) tritt ein Fehler auf.

Tatsächlich wird die Funktion `answer()` nicht innerhalb von philosophy.c, sondern in einer separaten Datei answer.c, einer sog. „Bibliothek“ definiert. Es ist möglich (und üblich), Bibliotheken separat vom Hauptprogramm zu compilieren. Dadurch spart man sich das Neucompilieren, wenn im Hauptprogramm etwas geändert wurde, nicht jedoch in der Bibliothek.

Mit der Option `-c` weisen wir `gcc` an, nur zu compilieren, jedoch nicht zu linken. Die Aufrufe

```
gcc -Wall -O -c philosophy.c
gcc -Wall -O -c answer.c
```

produzieren die Binärdateien philosophy.o und answer.o, die sogenannten Objekt-Dateien (daher die Endung .o oder .obj).

Mit dem Aufruf

```
gcc philosophy.o answer.o -o philosophy
```

bauen wir die beiden bereits compilierten Objekt-Dateien zu einer ausführbaren Datei philosophy (hier ohne Endung) zusammen.

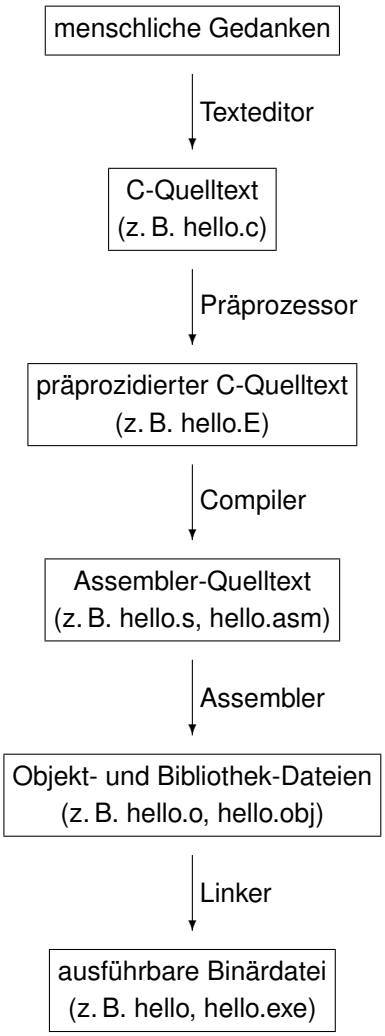
Es ist auch möglich, im Compiler-Aufruf gleich beide C-Quelltexte zu übergeben:

```
gcc -Wall -O philosophy.c answer.c -o philosophy
```

In diesem Fall ruft `gcc` zweimal den Compiler auf (für jede C-Datei einmal) und anschließend den Linker.

3.5 Die Toolchain

Wir können nun die Werkzeuge, die vom Schreiben des Quelltextes bis zur Ausführung des Programms verwendet werden, zu einer Kette, der „Toolchain“, zusammenfassen:



Manche dieser Werkzeuge können zu einer einzigen Software zusammengefaßt sein:

- Das Programm `gcc` faßt Präprozessor, Compiler, Assembler und Linker in einem einzigen Aufruf zusammen. Was jeweils aufgerufen wird, entscheidet das Programm anhand der Endungen der übergebenen Dateien.
- Eine „Entwicklungsumgebung“ (z. B. Eclipse) umfaßt typischerweise Texteditor, Präprozessor, Compiler, Assembler und Linker plus weitere Werkzeuge.

3.6 Standard-Pfade

Wenn eine Bibliothek regelmäßig von vielen Programmierern benutzt wird, wird sie üblicherweise an einem Standard-Ort abgelegt, z. B. in dem Verzeichnis /usr/lib.

`gcc` erwartet, daß die Namen von Bibliotheksdateien mit lib beginnen und die Endung .a oder .so haben. (.a steht für „Archiv“, da eine .a-Datei mehrere .o-Dateien enthält. .so steht für „shared object“ und bezeichnet eine Bibliothek, die erst zur Laufzeit eingebunden wird und von mehreren Programmen gleichzeitig benutzt werden kann. Andere übliche Bezeichnungen sind .lib anstelle von .a und .dll anstelle von .so.)

Mit der Option `-lfoo` teilen wir `gcc` mit, daß wir eine Datei libfoo.a aus einem der Standardverzeichnisse verwenden möchten. („foo“ ist eine metasyntaktische Variable und steht für ein beliebiges Wort.) Auch der Aufruf `-lm` zum Einbinden der Mathematik-Bibliothek ist nichts anderes. Tatsächlich gibt es eine Datei libm.a im Verzeichnis /usr/lib.

```
gcc test.c -lm -o test
```

ist somit dasselbe wie

```
gcc test.c /usr/lib/libm.a -o test
```

Mit der Option `-L /foo/bar` können wir ein Verzeichnis /foo/bar dem Suchpfad hinzufügen. („bar“ ist eine weitere metasyntaktische Variable.)

```
gcc test.c -L /home/joe/my\_libs -lmy -o test
```

compiliert test.c und linkt es mit einer Bibliothek libmy.a, nach der nicht nur in den Standardverzeichnissen (/usr/lib, /usr/local/lib u. a.), sondern zusätzlich im Verzeichnis /home/joe/my_libs gesucht wird.

Auf gleiche Weise kann man mit `-I /foo/bar` Verzeichnisse für Include-Dateien (s. o.) dem Standardsuchpfad hinzufügen.

3.7 Das Programm `make`

In größeren Projekten ruft man den Compiler (und Präprozessor und Linker) nicht „von Hand“ auf, sondern überläßt dies einem weiteren Programm namens `make`.

`make` sucht im aktuellen Verzeichnis nach einer Datei Makefile (ohne Dateiendung). (Normalerweise gibt es nur ein Makefile pro Verzeichnis. Falls es doch mehrere gibt, kann man die Datei, z. B. Makefile.1, mit `-f` auch explizit angeben: `make -f Makefile.1`.)

Ein Makefile enthält sog. Regeln, um Ziele zu erzeugen. Eine Regel beginnt mit der Angabe des Ziels, gefolgt von einem Doppelpunkt und den Dateien (oder anderen Zielen), von denen es abhängt. Darunter steht, mit einem Tabulator-Zeichen eingerückt, der Programmaufruf, der nötig ist, um das Ziel zu bauen.

```
philosophy.o: philosophy.c answer.h
        gcc -c philosophy.c -o philosophy.o
```

Achtung: Ein Tabulator-Zeichen läßt sich optisch häufig nicht von mehreren Leerzeichen unterscheiden. `make` akzeptiert jedoch nur das Tabulator-Zeichen.

Die o. a. Regel bedeutet, daß jedesmal, wenn sich philosophy.c oder answer.h geändert hat, `make` das Programm `gcc` in der beschriebenen Weise aufrufen soll.

Durch die Kombination mehrerer Regeln lernt `make`, welche Befehle in welcher Reihenfolge aufgerufen werden müssen, je nachdem, welche Dateien geändert wurden.

Beispiel: fileMakefile.1

Der Aufruf

```
make -f Makefile.1
```

bewirkt beim ersten Mal:

```
gcc -c philosophy.c -o philosophy.o
gcc -c answer.c -o answer.o
gcc philosophy.o answer.o -o philosophy
```

Beim zweiten Aufruf stellt `make` fest, daß sich keine der Dateien auf der rechten Seite der Regeln (rechts vom Doppelpunkt) geändert hat und ruft keine Programme auf:

```
make: »philosophy« ist bereits aktualisiert.
```

Um wiederkehrende Dinge (typischerweise: Listen von Dateinamen oder Compiler-Optionen) nicht mehrfach eingeben zu müssen, kennt `make` sog. Macros:

```
PHILOSOPHY_SOURCES = philosophy.c answer.h
```

Um den Macro zu expandieren, setzt man ihn in runde Klammern mit einem vorangestellten Dollarzeichen. Die Regel

```
philosophy.o: $(PHILOSOPHY_SOURCES)
        gcc -c philosophy.c -o philosophy.o
```

ist also nur eine andere Schreibweise für:

```
philosophy.o: philosophy.c answer.h
        gcc -c philosophy.c -o philosophy.o
```

Beispiel: Makefile.2

Makefile.2 verwendet Macros, um letztlich dasselbe zu erreichen wie Makefile.1. Zusätzlich führt es eine neue Regel `clean` ein. Diese bewirkt üblicherweise, daß alle automatisch erzeugten Dateien gelöscht werden:

```
clean:
        rm -f $(PHILOSOPHY_OBJECTS) philosophy
```

Rechts vom Doppelpunkt nach `clean:` befinden sich keine Abhängigkeitsdateien. Dies hat zur Folge, daß die Aktion (`rm -f ...`) bei `make clean` grundsätzlich immer ausgeführt wird, also nicht nur, wenn sich irgendeine Datei geändert hat.

(Ebenfalls üblich ist eine weitere Regel `install:`, die bewirkt, daß die Zieldateien (hier: die ausführbare Datei philosophy) an ihren endgültigen Bestimmungsort kopiert werden.)

3.8 Fazit: 3 Sprachen

Um in C programmieren zu können, muß man also tatsächlich drei Sprachen lernen:

- C selbst,
- die Präprozessor-Sprache
- und die `make`-Sprache.

Durch Entwicklungsumgebungen wie z. B. Eclipse läßt sich der `make`-Anteil teilweise automatisieren. (Eclipse z. B. schreibt ein Makefile; andere Umgebungen übernehmen die Funktionalität von `make` selbst.) Auch dort muß man jedoch die zu einem Projekt gehörenden Dateien verwalten.

Wenn sich dann eine Datei nicht an dem Ort befindet, erhält man u. U. wenig aussagekräftige Fehlermeldungen, z. B.:

```
make: *** No rule to make target 'MeinProgramm.elf', needed by 'elf'.  Stop.
```

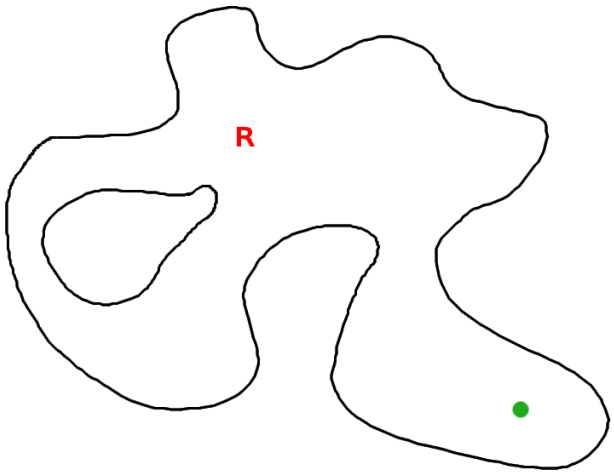
Wenn man dann um die Zusammenhänge weiß („Welche Bibliotheken verwendet mein Programm? Wo befinden sich diese? Und wie erfährt der Linker davon?“), kann man das Problem systematisch angehen und ist nicht auf Herumraten angewiesen.

4 Programmierung eines Roboters: Orientierung in einem Labyrinth

4.1 Aufgabenstellung

In diesem Projekt lautet die Aufgabenstellung, einen Roboter (Modell: Robby RP6, Hersteller: Arexx) so zu programmieren, daß er sich autark in einem Labyrinth orientiert und insbesondere einen im Labyrinth plazierten Zielpunkt zuverlässig findet.

In der nebenstehenden Zeichnung ist der Roboter durch ein rotes „R“ dargestellt, und der Zielpunkt durch einen großen grünen Punkt. Wände sind schwarz eingezeichnet.

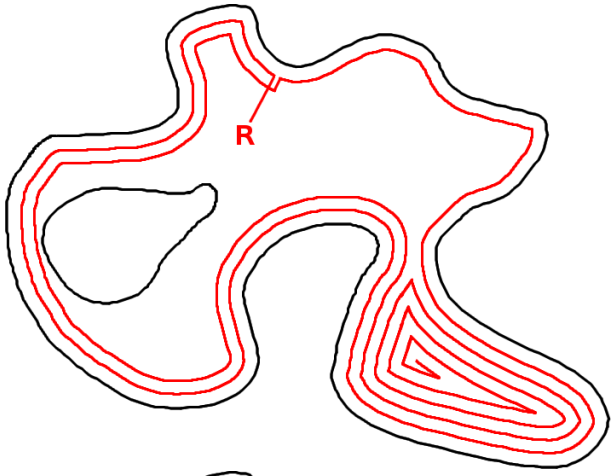


Tatsächlich ist das Finden des Zielpunktes nicht die eigentliche Aufgabe. Da sich der Zielpunkt irgendwo befinden kann, kann der Roboter ihn nur dann zuverlässig finden, wenn er das gesamte zugängliche Gebiet erkundet. De facto lautet also die Aufgabe: „Fahre jeden zugänglichen Punkt des Gebietes mindestens einmal an.“

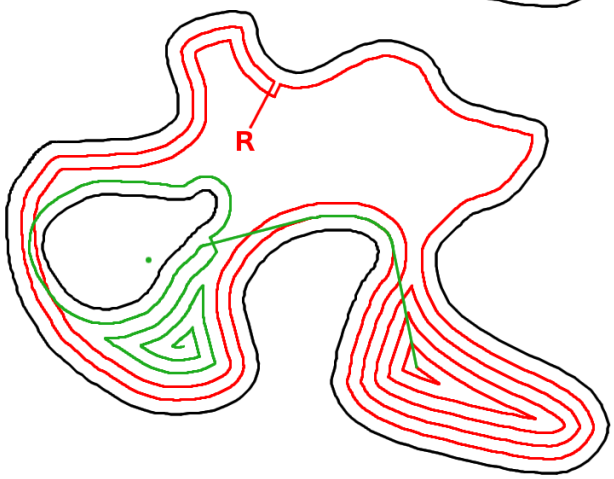
Das hier zu entwickelnde Programm eignet sich insofern auch für einen automatischen Staubsauger.

4.2 Verschiedene Lösungswege

Lösungsweg 1: Spiralen

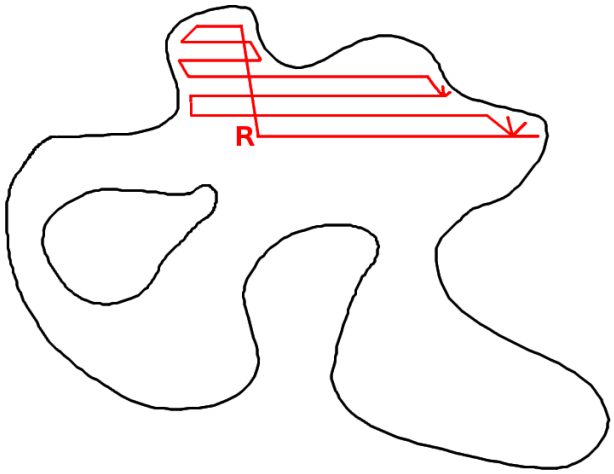


- Der Roboter muß über 2 Sensoren (rechts und links) verfügen.
- Der Roboter fährt zunächst den Rand des Gebietes ab und merkt sich die Geometrie.
- Wenn sich die Kurve schließt, fährt der Roboter den nächstgelegenen noch unbekannten Punkt im Inneren an und umfährt den Rand des noch unbekannten Gebiets. Auf diese Weise spiralt er sich nach innen (rote Linie).
- Sobald eine Spirale gefüllt ist, zielt der Roboter auf einen noch unbekannten Punkt (kleiner grüner Punkt innerhalb der Insel), z. B. mit Hilfe des Dijkstra-Algorithmus’ (grüne Linie).
- Wenn der Roboter auf ein Hindernis trifft, fährt er dessen Rand ab und setzt danach die Spirale fort (grüne Linie).
- Dies geschieht so lange, bis es keine unbekannten Gebiete mehr gibt.



- Vorteil: mathematisch „sauberer“ Algorithmus
- Nachteil: viele Fließkomma- und Vektor-Operationen, dadurch evtl. zu aufwendig für den Mikro-Controller
- Nachteil: viele Kurvenfahrten, dadurch evtl. Verlust an Genauigkeit bei der Positionsbestimmung des Roboters

Lösungsweg 2: Vektororientierter iterativer Flood-Fill-Algorithmus



- Der Roboter fährt zunächst in irgendeine Richtung, bis er auf ein Hindernis stößt.
- Er fährt dann am Rand entlang, bis er einen vorher definierten Abstand zum vorherigen Weg einnimmt. Danach schwenkt er auf einen Kurs entgegen dem vorherigen Weg ein.
- Der Hin-und-her-Kurs wird wiederholt, bis der Roboter eine Ecke des Bereichs gefunden hat.
- **?** Danach fährt der Roboter an Punkte neben den bereits untersuchten Linien und wiederholt das Bisherige.
- Vorteil: „saubere“ Vektor-Operationen
- Nachteil: Fließkomma- und Vektor-Operationen, dadurch evtl. zu aufwendig für den Mikro-Controller
- Problem: Für den oben mit „**?**“ markierten Schritt fehlt noch eine umsetzbare Ausarbeitung.

Lösungsweg 3: Pixelorientierter iterativer Flood-Fill-Algorithmus

- Der Roboter unterteilt das Gebiet in „Pixel“ (z. B. Quadrate von 25 cm Kantenlänge)
- Die Pixelgenauigkeit wird hinreichend fein gewählt, daß der Zielpunkt zuverlässig gefunden wird.
- Wir setzen voraus, daß wir einen Algorithmus schreiben können, der den Roboter von Pixel (x_1, y_1) nach Pixel (x_2, y_2) fahren läßt.
- Während sich der Roboter bewegt, speichert er eine „Karte“ des Gebiets. Pixel können „unbekannt“, „erreichbar“, „bekannt“ oder „Wand“ sein. Am Anfang sind alle Pixel „unbekannt“.
- Immer wenn der Roboter auf einem Pixel steht, markiert er diesen als „bekannt“ und die Nachbarpixel als „erreichbar“.
- In jedem „Spielzug“ fährt der Roboter einen „erreichbaren“ Pixel an. Wenn er dabei auf eine Wand trifft, markiert er den Ziel-Pixel als „Wand“, ansonsten als „bekannt“. Bereits bekannte Wände müssen dabei umfahren werden.
- Der Algorithmus endet, sobald es keine „erreichbaren“ Pixel mehr gibt.

Illustration:

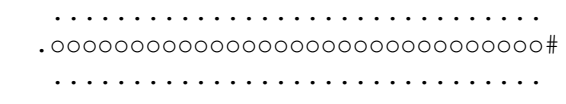
- Startsituation: Der Roboter steht auf einem Feld. Wir markieren es mit dem Symbol „○“ als „bekannt“, und wir markieren die Nachbarfelder mit dem Symbol „.“ als „erreichbar“.

```
 .
.○.
 .
```

- Der Roboter fährt nun einen erreichbaren Punkt an, z. B. den rechten. Solange er auf kein Hindernis stößt, fährt er weiter und markiert jeweils den erreichten Punkt als bekannt und die Nachbarpunkte als erreichbar.

```
. . . . . .
.○○○○○○○○○○○○○○○○○○○○○○.
. . . . . .
```

- Irgendwann stößt der Roboter auf eine Wand. Er markiert sie mit dem Symbol „#“.



- Er fährt zum letzten bekannten Punkt zurück und von da aus zum nächstgelegenen erreichbaren Punkt, z. B. dem oberen.

  Nehmen wir an, hier ist ebenfalls eine Wand. Diese wird markiert.

- Wir kehren an den letzten bekannten Punkt zurück und fahren von dort aus den nächstgelegenen erreichbaren Punkt an, jetzt also den unteren.

- Dies wird fortgesetzt, bis es keine als erreichbar markierten Punkte mehr gibt. (Es gibt dann nur noch bekannte Punkte und Wände sowie unbekannte Punkte, die hinter den Wänden liegen, also unerreichbar sind.)

- Nachteil: Der Algorithmus begrenzt die erreichbare Genauigkeit auf die Größe eines Pixels.
- Vorteil: Der Roboter bewegt sich überwiegend geradeaus.
  Dies sollte die mechanische Genauigkeit erhöhen.
- Vorteil: Die „Weltkarte“ wird in Form von Pixeln mit nur jeweils 3 Zuständen abgespeichert, was 2 Bits pro Punkt entspricht. Dies verbraucht erheblich weniger Speicherplatz als in Gestalt von Fließkomma-Vektoren abgespeicherte Kurven.

Lösungsweg 3 ist einfach genug, um ihn für den RP6 zu implementieren, und soll im folgenden ausgearbeitet werden.

4.3 Die `robosim`-Bibliothek

Damit wir uns nicht während der Entwicklung des Algorithmus mit den Besonderheiten der Software-Entwicklung für Mikro-Controller befassen müssen, entwickeln wir den Algorithmus zunächst für einen „virtuellen“ Roboter, der in der Grafik eines Personal-Computers simuliert wird.

(Parallel dazu wird in einer anderen Vorlesung auf die Besonderheiten der Software-Entwicklung für Mikro-Controller eingegangen. Das Übertragen des in der Vorlesung entwickelten Algorithmus' auf den Mikro-Controller des realen Roboters ist Bestandteil der zum Bestehen der Prüfung gestellten Projektaufgabe.)

Design-Ziele:

- aussagekräftige graphische Darstellung eines simulierten Roboters
- Aus Benutzersicht (= Sicht des Programms, das die Bibliothek nutzt)
  soll `robosim` der Bibliothek zur Steuerung des realen Roboters konzeptionell ähnlich sein.
- möglichst einfach zu verstehen

Funktionen:

- Bewegung starten
- Bewegung stoppen
- Abfrage, ob z. Zt. eine Bewegung stattfindet
- Abfrage der zurückgelegten Strecke
- Drehung bis Richtung
- Abfrage, ob eine Wand oder der Zielpunkt gefunden wurde
- Tastatur-Callback
- Timer-Callback

4.4 Installation der OpenGL- und PNG-Bibliotheken

Die Bibliothek `robosim` nutzt zur grafischen Ausgabe und zum Lesen von PNG-Dateien die OpenGL- und PNG-Bibliotheken. Um diese in den virtuellen Maschinen (Ubuntu) des Rechner-Pools nutzen zu können, müssen sie dort nachinstalliert werden:

- Terminal öffnen: Applications $\longrightarrow$ Accessoires $\longrightarrow$ Terminal
- `sudo su -`
  `Passwort:` (Beim Tippen wird *nichts* auf dem Bildschirm angezeigt. Auch keine Punkte.)
  `export http_proxy=http://cache.hs-bochum.de:8080`
  `apt-get install libglut3-dev libpng12-dev`
- Fragen mit `y` beantworten.
- Auf manchen Rechnern des Pools schlägt die Installation von `libglut3-dev` fehl. Auf diesen Rechnern installieren wir stattdessen `freeglut3-dev`.
- Die Angabe des Proxy muß auch dann erfolgen, wenn dieser bereits in Firefox eingestellt wurde.

4.5 Grundsätzliche Benutzung der `robosim`-Bibliothek

Vorab ein wichtiger Hinweis:

Niemals als Systemadministrator (`root`) Software entwickeln!

Während der Software-Entwicklung ist es unvermeidlich, fehlerbehaftete Software testweise laufen zu lassen. Wer als Administrator arbeitet, riskiert, daß sich selbst harmloseste Fehler fatal auf das System auswirken.

Beispiel: Ein Makefile enthält die Regel

```
clean:
        rm -rf $(TMP_DIR)/*
```

Dies soll bewirken, daß mit `make clean` irgendwelche zwischendurch angelegten temporären Daten gelöscht werden können.

Nehmen wir nun an, der Programmierer des Makefiles hat vergessen, den Macro `TMP_DIR` zu setzen oder vertippt sich dabei (z. B. `TEMP_DIR`). In dem Fall würde der Befehl `rm -rf /*` ausgeführt. (Bedeutung: lösche den gesamten Inhalt aller angeschlossenen Festplatten.)

- Aufruf als Normalbenutzer: Es passiert überhaupt nichts. Wegen der Option `-f` bekommt man noch nicht einmal eine Fehlermeldung.
- Aufruf als Administrator: Der Inhalt sämtlicher angeschlossenen Festplatten und Wechseldatenträger wird ohne Rückfrage und ohne optische Anzeige gelöscht.

Bevor wir uns also den nachfolgenden Beispielen zuwenden, stellen wir sicher, daß unser Terminal als normaler Benutzer läuft und nicht als Systemadministrator. Dies erkennt man normalerweise an dem Zeichen „`#`“ in der Eingabeaufforderung (Prompt). Im Zweifel kann man sich mit dem Unix-Kommando `id` vergewissern und nötigenfalls die `root`-Sitzung mit dem Kommando `exit` beenden.

Beispiel 1: robosim-test-1.c

```
#include "robosim.h"

int main (int argc, char **argv)
{
  robosim_init (argc, argv, "parcour.png", 426, 493, -90);
  robosim_start_move ();
  robosim_run ();
  return 0;
}
```

Dieses Programm illustriert die grundsätzliche Art der Benutzung der Bibliothek.

- Compilieren:

  ```
  gcc -Wall robosim-test-1.c -o robosim-test-1 \
      -lGL -lGLU -lglut -lpng robosim.c
  ```

  (Der Backslash am Ende steht für eine Fortsetzung des Befehls in der nächsten Zeile.)

  Die Standard-Bibliotheken werden mit der Option `-l` (ein Bindestrich und ein kleines L für „library“) eingebunden. Abweichend davon wird die robosim-Bibliothek `robosim.c` direkt als Quelltext mit angegeben.

- Aufrufen:

  ```
  ./robosim-test-1
  ```

  (Angabe des Dateinamens mit vollem Pfad – hier: im aktuellen Verzeichnis, symbolisiert durch einen Punkt.)

- Zeile im Programm:

  ```
  robosim_init (argc, argv, "parcour.png", 426, 493, -90);
  ```

  Initialisierung der Bibliothek, Laden des Hintergrundbildes, Anfangs-Positionierung des Roboters

  Dies ist die einzige Stelle, an der Bildschirm-Koordinaten verwendet werden.

- Zeile im Programm:

  ```
  robosim_start_move ();
  ```

  Dem Roboter wird das Signal gegeben, loszufahren.

- Zeile im Programm:

  ```
  robosim_run ();
  ```

  Die Hauptschleife der Bibliothek wird aufgerufen. Diese Hauptschleife ist eine Endlosschleife; das Programm erhält die Kontrolle nicht zurück. Hierfür müssen sog. Callbacks installiert werden.

  Dieses Konzept ist typisch für viele Bibliotheken. Alternativ besteht auch die Möglichkeit, daß der Programmierer die (endlose) Hauptschleife selbst schreibt und von dort aus `task`-Funktionen aufruft. (Dies ist z. B. in der RP6-Bibliothek der Fall.)

Dieses Programm läßt den (simulierten) Roboter in die aktuelle Richtung losfahren. Da kein Code vorgesehen ist, um Wände zu erkennen, ignoriert der Roboter die Wände und fährt stur geradeaus, bis man das Programm abbricht.

Um dies zu ändern, benötigen wir Callbacks. Diese sind Gegenstand des nächsten Beispiels.

Beispiel 2: robosim-test-2.c

```
#include <stdlib.h>
#include "robosim.h"

void key_handler (char which_key)
{
  switch (which_key)
    {
    case 'l':
      robosim_rotate (-90);
      break;
    case 'r':
      robosim_rotate (90);
      break;
    case ' ':
      if (robosim_moving ())
        robosim_stop_move ();
      else
        robosim_start_move ();
      break;
    case 'q':
      exit (0);
    }
}

int main (int argc, char **argv)
{
  robosim_init (argc, argv, "parcour.png", 426, 493, -90);
  robosim_on_key_pressed (key_handler);
  robosim_run ();
  return 0;
}
```

- Vor dem Aufruf der Hauptschleife wird die selbstgeschriebene Funktion `key_handler()` als sogenanntes Callback installiert.
- Bei jedem Tastendruck wird die Hauptschleife (`robosim_run()`) unterbrochen und `key_handler()` aufgerufen.
- Wenn `key_handler()` zurückkehrt, wird die Hauptschleife fortgesetzt.

Die Funktion `key_handler()` bewirkt, daß der Roboter

- bei Betätigung der Taste „l“ nach links dreht,
- bei Betätigung der Taste „r“ nach rechts dreht,
- bei Betätigung der Leertaste losfährt bzw. anhält
- und bei Betätigung der Taste „q“ das Programm beendet.

Dieses Programm ist weiterhin nicht in der Lage, Wände zu erkennen. Dies geschieht im nächsten Beispiel.

Beispiel 3: robosim-test-3.c

Dieses Programm illustriert die Benutzung des Timer-Callbacks.

Wir erweitern robosim-test-2.c um eine Funktion:

```
void timer_handler (void)
{
  robosim_sensor_result sensor = robosim_query_sensor ();
  if (sensor == found_wall)
    robosim_rotate (180);
}
```

- Vor dem Aufruf der Hauptschleife wird sowohl ein Keyboard-Callback `key_handler()` als auch ein Timer-Callback `timer_handler()` installiert.
- Zehnmal pro Sekunde (so festgelegt in der Bibliothek) wird die Hauptschleife (`robosim_run()`) unterbrochen und `timer_handler()` aufgerufen.
- Innerhalb von `timer_handler()` fragen wir den (simulierten) Sensor ab. Wenn dieser eine Wand detektiert, drehen wir den Roboter auf der Stelle um 180 Grad.
- Wenn `timer_handler()` zurückkehrt, wird die Hauptschleife fortgesetzt.

Es ist insbesondere nicht möglich, von außerhalb der Callback-Handler auf den sich bewegenden Roboter Einfluß zu nehmen.

4.6 Elementare Bewegungen des Roboters

Aufgabe: Schreiben Sie ein Programm, das den Roboter im Quadrat fahren läßt.

Musterlösung: robosim-uebung-1.c

- Im Timer-Callback wird die bisher zurückgelegte Entfernung gemessen. Wenn diese einen bestimmten Wert (`SQUARE_SIZE`) erreicht hat, wird der Roboter um 90° gedreht und der Entfernungsmesser auf 0 zurückgesetzt.
- Der Vergleich `>=` anstelle von `==` wird wichtig, sobald wir es mit echten Entfernungsmessungen zu tun bekommen. Kein realer Roboter wird auf 15 Nachkommastellen genau „40 cm“ als Meßergebnis für die zurückgelegte Strecke zurückliefern.
- Der Timer-Callback ist der einzige Ort, an dem die Entfernungsmessung stattfinden kann. Dies kann deswegen nicht im Hauptprogramm erfolgen, weil sich einerseits der Roboter nur innerhalb von `robosim_run()` bewegt, andererseits `robosim_run()` eine Endlosschleife darstellt, aus der heraus kein anderer Programmteil aufgerufen wird – es sei denn über Callbacks.
- Der Tastatur-Callback ist weiterhin aktiv. Insbesondere muß der zunächst stillstehende Roboter mit der Leertaste gestartet werden, und er kann während der Bewegung per Tastendruck gedreht oder gestoppt werden.

Beispiel 4: robosim-test-4.c

Dieses Programm illustriert, wie man den Roboter eine komplexere Bewegungsaufgabe lösen läßt.

- Bewegungsaufgabe für den Roboter: „Bewege Dich um eine vorgegebene Distanz (`PIXEL_SIZE`) vorwärts. Wenn Du unterwegs auf ein Hindernis triffst, kehre an den Ausgangspunkt zurück.“
- Genau wie in robosim-test-3.c wird der Sensor vom Timer-Handler aus abgefragt. Wenn eine Wand detektiert wird, dreht sich der Roboter um 180°.
- Zusätzlich zu robosim-test-3.c mißt der Roboter bei Berührung der Wand die zurückgelegte Entfernung, speichert sie in der Variablen `distance_from_home` und merkt sich, daß er sich nun auf dem „Rückweg“ befindet (`direction = DIRECTION_BACKWARD;`).
- Eine Vorwärtsbewegung wird nach der gemessenen Entfernung `PIXEL_SIZE` beendet, eine Rückwärtsbewegung nach `distance_from_home`.
- Beim nächsten Losfahren (ausgelöst durch die Leertaste) merkt sich der Roboter die Bewegung wieder als „Hinweg“ (`direction = DIRECTION_FORWARD;`).

4.7 Ein Gebiet kartographieren: das „Gedächtnis“ des Roboters

Beispiel 1: robosim-parcour-1.c

Wir erweitern nun unseren simulierten Roboter um ein „Gedächtnis“. Der Roboter soll sich merken,

- wo er bereits war,
- welche Punkte aufgrund von Hindernissen nicht erreicht werden konnten
- und welche Punkte für ihn zwar unbekannt, aber erreichbar sind.

Um sich dies merken zu können, unterteilen wir für den Roboter die Welt in ein „Raster“ aus „Kacheln“ (oder „Pixeln“) mit fester Kantenlänge `PIXEL_SIZE` und speichern in einem zweidimensionalen Array `char mark_buffer[]` für jede Kachel einen von vier Zuständen:

- unbekannt (`MARK_UNKNOWN`),
- bekannt (`MARK_KNOWN`),
- durch Hindernis versperrt (`MARK_WALL`)
- oder unbekannt, aber erreichbar (`MARK_REACHABLE`).

Der `mark_buffer[]` ist gewissermaßen die vom Roboter aufgezeichnete „Weltkarte“.

(Bemerkung: Ein `char` hat mindestens 8 Bits, kann also Werte von 0 bis 255 speichern, von denen wir aber nur vier brauchen. Diese „Verschwendung“ sollte für einen realen Roboter mit Mikro-Controller mittels „Bit-Basteleien“ umschifft werden.)

- Das Array `mark_buffer[]` wird nicht direkt geschrieben und gelesen, sondern über sog. „Getter“ (`char get_mark_buffer()`) und „Setter“ (`void set_mark_buffer()`). Diese sind so geschrieben, daß sie erkennen, sobald der gültige Bereich des „Gedächtnisses“ verlassen wird und das Programm mit einer Fehlermeldung kontrolliert beenden.
- Auch die Setter werden nicht direkt aufgerufen, sondern über Hilfsfunktionen `mark_as_known()`, `mark_as_wall()` und `mark_as_reachable()`. Letztere enthält eine Prioritätsprüfung: Ein Punkt wird nur dann als „erreichbar“ markiert, wenn er nicht bereits als „Wand“ oder als „bekannt“ markiert ist.
- Eine weitere Hilfsfunktion `mark_neighbours_as_reachable()` ruft viermal `mark_as_reachable()` auf, um die Nachbarpunkte eines vorgegebenen Koordinatenpaares zu markieren.
- Parallel zu der „realen“ Bewegung des (simulierten) Roboters führen wir eine „logische“ Bewegung durch den `mark_buffer[]` mit: Wir merken uns auf dem Variablenpaar `rx, ry` die aktuelle Position des Roboters innerhalb des `mark_buffer[]`s und aktualisieren sie nach jeder Bewegung.
- Dies gilt auch für Drehungen: Parallel zu der „realen“ Drehung (`robosim_rotate()`) lassen wir einen logischen Richtungsvektor `dx, dy` mitdrehen. Dies geschieht in der Funktion `rotate()`, die ihrerseits `robosim_rotate()` aufruft.
- Die Anfangsposition für `rx, ry` wird in die Mitte des `mark_buffer[]`s gesetzt, weil wir anfangs nichts über die „Welt“ des Roboters wissen und in allen Richtungen gleich viel Platz zur Verfügung haben wollen.
- Zur Initialisierung (im Hauptprogramm vor dem Aufruf der Hauptschleife `robosim_run()`) wird der `mark_buffer[]` in den Zustand „alles unbekannt“ versetzt, anschließend die aktuelle Position als „bekannt“ markiert und die Nachbarpunkte als „erreichbar“.
- Dasselbe geschieht innerhalb des `timer_handler()`s, immer wenn sich der Roboter weiterbewegt hat: Die aktuelle Position wird als „bekannt“ markiert und die Nachbarpunkte als „erreichbar“.
- Der `keyboard_handler()` wurde um die Taste ’b’ ergänzt, mit der man sich im laufenden Programm den Inhalt des `mark_buffer[]`s auf den Bildschirm ausgeben lassen kann.

Mit dem Programm robosim-parcour-1.c können wir den manuell (mit Leertaste, „l“ und „r“) gesteuerten Roboter seine Welt erkunden lassen. Der nächste Schritt lautet, den Roboter autark die gesamte Gegend erfassen zu lassen.

4.8 Jeden Punkt erreichen: Floodfill

Um die autarke Bewegung des Roboters vorzubereiten, befassen wir uns mit dem Floodfill-Algorithmus aus der Grafikbearbeitung: Wie füllt man eine farbig umrandete Fläche in einer anderen Farbe aus?

Aufgabe: Füllen Sie, vom momentanen Standpunkt des Roboters ausgehend, den durch den Roboter als erreichbar bekannten Bereich in `mark_buffer` mit dem Zeichen „`*`“ aus.

Mit „erreichbarer Bereich“ sind alle Punkte gemeint, die entweder als „bekannt“ oder als „erreichbar“ markiert sind. Unbekannte Punkte oder Wände gelten in diesem Zusammenhang als „nicht erreichbar“.

Hinweis: Wikipedia-Seite zu Floodfill

Musterlösung: robosim-uebung-2.c

- Wir schreiben eine Funktion `flood_fill()`, die ein Koordinatenpaar als Parameter erwartet, den `mark_buffer[]` an dieser Stelle mit „`*`“ markiert und anschließend sich selbst für jeden der vier Nachbarpunkte einmal aufruft („Rekursion“).
- Die Funktion wird nur aktiv, wenn der übergebene Punkt als „bekannt“ oder als „erreichbar“ markiert ist. In anderen Fällen bricht sie ab.
- Um die Funktion aufzurufen, kann man z. B. eine weitere Taste in Betrieb nehmen (hier: „f“). Später wird es so sein, daß der Roboter die Funkion selbständig aufruft, sobald er einen neuen Weg sucht – also z. B., wenn er auf ein Hindernis gestoßen und zum Ausgangspunkt der Bewegung zurückgekehrt ist.

Bemerkung 1:

```
if (get_mark_buffer (x, y) == MARK_KNOWN || MARK_REACHABLE)
  ...;
```

hat eine völlig andere Bedeutung als

```
if (get_mark_buffer (x, y) == MARK_KNOWN
    || get_mark_buffer (x, y) == MARK_REACHABLE)
   ...;
```

Die zweite Version liefert das, was wir haben wollen. Die erste Version sieht aus wie eine kürzere Version der zweiten, ist es aber nicht!

Die obere `if`-Anweisung ist zwar korrektes C, aber für unsere Zwecke falsch. Die Bedingung
`get_mark_buffer (x, y) == MARK_KNOWN || MARK_REACHABLE`
ist die Logische Oder-Verknüpfung der Bedingung `get_mark_buffer (x, y) == MARK_KNOWN` mit der Konstanten `MARK_REACHABLE`. `MARK_REACHABLE` hat den Wert `'.'`, ist also immer ungleich Null. Daher ist auch die Oder-Verknüpfung immer ungleich Null, und die von dem `if` abhängige Anweisung wird grundsätzlich immer ausgeführt – unabhängig davon, was `get_mark_buffer (x, y)` zurückliefert.

Bemerkung 2: Wen man den `mark_buffer[]` anstatt mit '*' mit 'o' oder mit '.' füllt, läuft das Programm in eine unendliche Rekursion, weil die markierten Punkte dann nicht als „erledigt“ betrachtet werden, sondern als „noch zu bearbeiten“. Die `flood_fill()`- Funktion wird diese Punkte also wieder und wieder beschreiben, bis der CPU-Stack volläuft und das Betriebssystem das Programm mit einer Fehlermeldung abbricht. (Wenn das Programm auf einem Mikro-Controller läuft, der keinen derartigen „Wächter“ hat, würde es sich dann übrigens völlig unkontrolliert verhalten.)

Bemerkung 3: Wenn es nur darum ginge, in `mark_buffer[]` alle `'o'` und alle `'.'` durch `'*'` zu ersetzen, könnte man das natürlich auch ganz ohne Floodfill durch eine simple Doppelschleife erreichen:

```
int x, y;
for (x = 0; x < MARK_BUFFER_WIDTH; x++)
  for (y = 0; y < MARK_BUFFER_HEIGHT; y++)
    if (get_mark_buffer (x, y) == MARK_KNOWN
        || get_mark_buffer (x, y) == MARK_REACHABLE)
      set_mark_buffer (x, y, '*');
```

Da es uns aber darum geht, einen Weg von der aktuellen Position des Roboters zu einem erreichbaren Punkt zu finden, ist es für uns zielführender, sich von Nachbarpunkt zu Nachbarpunkt zu hangeln – eben so, wie Floodfill es macht.

Das nächste Beispiel-Programm identifiziert zumindest schon mal den für den Roboter nächstgelegenen als „erreichbar“ markierten Punkt.

Beispiel 2: robosim-parcour-2.c

Unser simulierter Roboter bekommt nun eine zweite „Landkarte“, in der er für jeden Punkt einzeichnet, wie „teuer“ es ist, ihn zu erreichen. Diese Landkarte ist das zweidimensionale Array `path_buffer[]`, das genauso viele Punkte speichert wie `mark_buffer[]`, aber jedem Punkt eine ganze Zahl zuordnet, nämlich die „Kosten“, um dorthin zu gelangen.

Um diese „Kosten-Landkarte“ (`path_buffer[]`) zu ermitteln, verwenden wir eine leichte Abwandlung des oben erprobten Floodfill-Algorithmus'.

- Wir definieren eine Konstante `PATH_UNKNOWN` als eine möglichst große ganze Zahl. Dies dient dazu, noch unbekannten oder nicht erreichbaren Punkten „unendliche Kosten“ zuordnen zu können. Als „möglichst große“ Zahl verwenden wir die Konstante `INT_MAX` aus der Standard-Include-Datei limits.h. (Auf PCs hat sie den Wert $2^{31} - 1 = 2147483647$.)
- Wir schreiben die üblichen Getter- und Setter-Funktionen für den `path_buffer[]` sowie eine Funktion `erase_path_buffer()`, um ihn auf den Wert `PATH_UNKNOWN` zu initialisieren.
- Der Zeitpunkt, an dem wir sinnvollerweise die „Kosten-Landkarte“ berechnen, ist immer am Ende einer Bewegung. Dies sind zwei Stellen in `timer_handler()`. Dort rufen wir eine neue Funktion `aim_at_closest_reachable_pixel()` auf.
- `aim_at_closest_reachable_pixel()` löscht den `path_buffer[]`
  und ruft anschließend eine weitere Funktion `examine_path ()` auf.
  Diese ist das eigentliche Kernstück des Beispiel-Programms.
- `examine_path()` macht im wesentlichen dasselbe wie das `flood_fill()` aus der Übungsaufgabe: Es prüft, ob ein Punkt ein Randpunkt ist oder bereits markiert wurde. Falls beides nicht zutrifft, markiert `examine_path()` den Punkt und ruft sich selbst für die vier benachbarten Punkte auf. Als Erweiterung gegenüber `flood_fill()` markiert `examine_path()` die Punkte nicht einfach nur als „erledigt“, sondern notiert für jeden Punkt ein Maß für die „Kosten“ („cost“), die nötig sind, ihn zu erreichen. Diese Kosten werden bei jedem Aufruf von `examine_path()` um 1 erhöht. Wenn sich der Roboter drehen muß, um den Punkt zu erreichen, werden die Kosten um eine weitere Einheit erhöht.
- Auf diese Weise enthält `path_buffer[]` nach dem Durchlauf von
  `aim_at_closest_reachable_pixel()` bzw. aller von ihm aufgerufenen `examine_path()`s
  eine „Weltkarte“, auf der für jeden Punkt die Kosten notiert sind, um ihn zu erreichen.
  (Für unerreichbare Punkte sind die Kosten `PATH_UNKNOWN`, also eine möglichst große Zahl.)
- Der `keyboard_handler()` wurde um die Taste „p“ ergänzt, mit der man sich im laufenden Programm den Inhalt des `path_buffer[]`s auf den Bildschirm ausgeben lassen kann.

Der Roboter wird weiterhin von Hand gesteuert. Das Einbauen eines Automatismus', mit dessen Hilfe er autark den nächstgelegenen – „billigsten“ – erreichbaren unbekannten Punkt ansteuert, wird Gegenstand des nächsten Beispielprogramms robosim-parcour-3.c sein – siehe unten.

4.9 Ermittlung des kürzesten Weges: Minimum berechnen

Beispiel 3: robosim-parcour-2a.c

Dieses Beispiel-Programm ist eine Vorübung zu robosim-parcour-3.c. Während robosim-parcour-2.c für jeden erreichbaren Punkt die Kosten ermittelt, ermittelt robosim-parcour-2a.c die Koordinaten des jeweils „kostengünstigsten“ erreichbaren unbekannten Punktes.

- Es werden zusätzliche Variablen `bx`, `by` und `best_cost` eingeführt. `bx` und `by` sollen die Koordinaten des „kostengünstigsten“ („besten“) erreichbaren unbekannten Punktes speichern; `best_cost` speichert die zugehörigen Kosten.

- In `aim_at_closest_reachable_pixel()` wird vor dem Aufruf von `examine_path()` die `best_cost` auf eine große Zahl `PATH_UNKNOWN` initialisiert.
- Innerhalb von `examine_path()` wird geprüft, ob der aktuelle Punkt ein unbekannter erreichbarer Punkt ist (`cost != 0 && get_mark_buffer (x, y) == MARK_REACHABLE`) und ob er kostengünstiger ist als die bisherigen `best_cost`.
- Falls diese Bedingungen zutreffen, wird der aktuelle Punkt als neuer „bester“ Punkt in `bx, by` gespeichert und die neuen „besten“ Kosten in `best_cost`. Auf diese Weise berechnen wir das Minimum der Kosten über alle unbekannten erreichbaren Punkte.
- Der `keyboard_handler()` wurde um die Taste „c“ ergänzt, mit der man sich im laufenden Programm die Koordinaten des so ermittelten „kostengünstigsten“ Punktes einschließlich Kosten ausgeben lassen kann.

Eine weitere Vorübung zu robosim-parcour-3.c ist der Aufbau der Datenstruktur „Stack".

4.10 Speichern des kürzesten Weges: Stack

Aufgabe: Schreiben Sie einen Software-Stack für Koordinaten.

Koordinatenpaare sollen „übereinander gestapelt“ werden.

- Eine Funktion `push()` legt sie oben auf dem Stack ab.
- Eine Funktion `pop()` entfernt sie wieder vom Stack.
- Eine Funktion `dump()` gibt den Inhalt des Stacks auf dem Bildschirm aus.

Vereinfachung: Nur 1 Koordinate. Eindimensional. Nur `x`, kein `y`.

Musterlösung 1 (mit Vereinfachung): robosim-uebung-3a.c

- Als „Aufbewahrungsort“ für den Stack-Inhalt definieren wir ein Array `the_stack[]` von ganzen Zahlen.
- Wir definieren eine globale Variable `stack_pointer`, die als Index für das Array `the_stack[]` dienen soll. Diese wird zu Programmbeginn auf 0 initialisiert und zeigt damit auf einen freien Platz innerhalb von `the_stack[]`.
- Die Funktion `push_number()` schreibt die zu speichernde Zahl in den freien Platz `the_stack[stack_pointer]` und erhöht anschließend den `stack_pointer` um 1, so daß er auf den nächsten freien Platz in `the_stack[]` verweist.
- Die Funktion `pop_number()` entfernt eine Zahl aus dem Stack, indem sie den `stack_pointer` um 1 herabsetzt. (Es ist auch üblich, dies mit einem Auslesen der Zahl zu kombinieren. In diesem Fall wäre `pop_number()` nicht vom Typ `void`, sondern vom Typ `int` und würde eine Zeile `return the_stack[stack_pointer]` enthalten.)
- `push_number()` enthält eine Sicherheitsabfrage, damit sich das Programm im Falle eines Überlaufs von `the_stack[]` mit einer Fehlermeldung kontrolliert beendet, anstatt in den freien Speicher außerhalb von `the_stack[]` zu schreiben.
- `pop_number()` enthält eine Sicherheitsabfrage, die verhindert, daß beim Löschen einer Zahl aus dem bereits leeren Stack der `stack_pointer` auf -1 gesetzt wird, was beim nächsten `push_number()` ebenfalls ein Schreiben in den freien Speicher außerhalb von `the_stack[]` zur Folge hätte.
- Beide Funktionen enthalten Bildschirmausgaben, die anzeigen, was jeweils geschieht.
- Die Funktion `dump_stack()` gibt den aktuellen Inhalt von `the_stack[]` auf dem Bildschirm aus. Zur besseren Übersicht wird im Falle eines leeren Stacks nicht „nichts“ ausgegeben, sondern der Text „`dump_stack: stack is empty`“. Aus demselben Grund wird der Zeile mit den Werten der Text `dump_stack:` vorangestellt, und die Zahlen werden durch Leerzeichen getrennt. (Der `printf()`-Aufruf enthält `" %d"` mit Leerzeichen anstelle von `"%d"`\verb ohne Leerzeichen.)

- Das Hauptprogramm testet den Stack, indem es mehrere Zahlen darin speichert, wieder löscht, andere Zahlen speichert und den Stack wieder leert. Nach jeder Aktion wird `dump_stack()` aufgerufen, um den aktuellen Stack-Inhalt anzuzeigen.

Die Übungsaufgabe gilt auch ohne die Sicherheitsabfragen und ohne die zur besseren Übersicht ausgegebenen Texte als gelöst.

Musterlösung 2 (ohne Vereinfachung): robosim-uebung-3b.c

- Anstelle von Zahlen enthält der Stack jetzt Paare von Zahlen. Hierfür wird ein selbstdefinierter Datentyp eingeführt: ein `struct`.
- Mit `typedef struct { int x, y; } pair;` wird ein neuer Datentyp `pair` eingeführt. Wenn wir eine Variable `a` dieses Typs einführen würden, wären `a.x` und `a.y` ganze Zahlen, die über einen gemeinsamen Namen `a` und Komponentennamen `.x` und `.y` angesprochen würden.
- Gegenüber robosim-uebung-3a.c ersetzen wir nun jedes `x` durch das Paar `x` und `y`. Ansonsten ändert sich nichts.

Alternative Lösungen zum Speichern von Koordinatenpaaren wären:

- 2 Stacks benutzen (einen für `x`, einen für `y`). Dieser Weg ist machbar, aber umständlich.
- Anstelle eines `struct` ein Array benutzen. Vorteil: man kann beide Koordinaten über ihren Array-Index in einer Schleife ansprechen. Nachteil: Bei einem Array ist es zwingend erforderlich, daß beide Variablen im Paar den gleichen Typ haben.
- Den Stack doppelt so groß machen und abwechselnd für x und y benutzen. Wenn man daran denkt, in `push_pair()` immer zuerst x und dann y zu `push`en und nicht vergißt, in `pop_pair()` die Werte in umgekehrter Reihenfolge wieder vom Stack zu holen, ist dieser Weg gangbar. Auch hier ist Typengleichheit zwingend erforderlich.

Wir wollen nun einen Stack verwenden, um den „besten“ Punkt nicht nur zu ermitteln, sondern auch den Weg dorthin zu speichern.

Beispiel 4: robosim-parcour-3.c

Dieses Beispielprogramm läßt den (mit der Taste „a“) einmal angestoßenen Roboter selbständig das Gebiet erkunden. Immer wenn er auf eine Wand oder auf einen bereits bekannten Punkt stößt, sucht er sich selbständig ein neues Ziel.

- Die rekursive Funktion `examine_path()` merkt sich auf einem Stack, auf welchem Weg der aktuell zu untersuchende Punkt erreicht wurde.
  - Am Anfang ist der Stack leer.
  - Beim ersten rekursiven Aufruf wird der zu untersuchende Punkt auf den Stack gelegt. Dieser ist ein Nachbarpunkt des Roboters. Der Stack enthält also am Anfang den korrekten Weg zu den Nachbarpunkten.
  - In jedem weiteren Aufruf findet `examine_path()` auf dem Stack einen korrekten Weg vor und ergänzt ihn um den aktuellen Punkt.
  - Am Ende der Untersuchung eines Punktes, also bevor der nächste Punkt untersucht wird, wird der aktuelle Punkt wieder vom Stack genommen.

  Der Stack ändert sich also ständig und enthält zu jedem Zeitpunkt den Weg zu demjenigen Punkt, der gerade untersucht wird.
- Wenn der aktuelle Punkt die bisher günstigsten Kosten hat, speichert `examine_path()` den Weg zu diesem Punkt – also den aktuellen Inhalt des Stacks – auf der Array-Variablen `best_path[]` ab.

  (Da der Weg den Punkt selbst – als Endpunkt – enthält, braucht dieser nicht noch einmal separat gespeichert zu werden. Wohl hingegen muß die Länge des Weges, also die Anzahl der Punkte abgespeichert werden.)

- Nachdem `aim_at_closest_reachable_pixel()` durchgelaufen ist, ist der kostengünstigste Weg zu einem als „erreichbar“ markierten Punkt bekannt und liegt auf der Array-Variablen `best_path[]`.
- Der `keyboard_handler()` wurde um zwei Tasten ergänzt:
  - Mit „P“ kann man sich im laufenden Programm die „Kosten“ der erreichbaren unbekannten Punkte sowie den „kostengünstigsten“ Weg dorthin anzeigen lassen.
  - Mit „a“ schaltet man in einen Auto-Modus, in dem man die Bewegung nicht für jeden Schritt mit der Leertaste neu anstoßen muß, sondern der Roboter nach einer kurzen Wartezeit den nächsten Schritt selbständig ausführt.
- Wenn die Funktion `aim_at_closest_reachable_pixel()` keinen neuen Weg mehr findet, schaltet sie den Auto-Modus aus, und der Roboter hält an.

Nachdem nun der kostengünstigste Weg bekannt ist, muß „nur noch“ der Roboter diesen Weg entlanggesteuert werden.

4.11 Einem Weg folgen: Richtungsvektoren

Die Funktion `aim_at_closest_reachable_pixel()` bekommt nun eine weitere Aufgabe: Wenn ein kostengünstigster Weg bekannt ist (`best_path_length > 0`), soll sie den Roboter so drehen, daß er diesem Weg folgt.

- Die aktuelle Position innerhalb des Weges wird durch einen Index `best_path_pointer` markiert. Die Koordinaten des nächsten anzusteuernden Punktes lauten also `best_path[best_path_pointer].x` und `best_path[best_path_pointer].y`.
- Die Richtung, in der der nächste anzusteuernde Punkt liegt, erhalten wir, in dem wir von dessen Koordinaten die aktuellen Koordinaten des Roboters `rx, ry` subtrahieren. Wir erhalten einen Richtungsvektor `bdx, bdy`.
- Da der Weg aus zueinander benachbarten Punkten besteht, gibt es überhaupt nur vier Möglichkeiten, wie dieser Richtungsvektor aussehen kann, nämlich: nach oben $(0, 1)$, nach unten $(0, -1)$, nach links $(-1, 0)$ oder nach rechts $(1, 0)$.
- Wir vergleichen den Richtungsvektor `bdx, bdy` mit der aktuellen Bewegungsrichtung `dx, dy` des Roboters, die ja ebenfalls nur dieselben vier Werte annehmen kann. Durch diesen Vergleich ermitteln wir, wie wir den Roboter drehen müssen, damit er sich anschließend in Richtung `bdx, bdy` bewegt.
- Nachdem wir auf diese Weise einen Punkt des Weges abgearbeitet haben, inkrementieren wir den Index `best_path_pointer`, d. h., wir wenden wir uns dem nächsten Punkt zu.
- Wenn wir am Ende des Weges angekommen sind, also `best_path_pointer` durch das Inkrementieren `best_path_length` erreicht hat, löschen wir den Weg, indem wir `best_path_length` auf 0 setzen. Dies bewirkt, daß beim nächsten Aufruf von `aim_at_closest_reachable_pixel()` wieder ein neuer Weg gesucht wird (Aufruf von `examine_path()`).