Einführung in C

Patrick Schulz
patrick.schulz@paec-media.de

29.04.2013

Hello World in Java

```
1 public class hello_world
2 {
3     public static void main(String [] args)
4     {
5         System.out.println("Hello World!");
6     }
7 }
```

Hello World in C

```c
#include <stdio.h>

int main()
{
    printf("Hello World\n");
    return 0;
}
```

Präprozessor, Compiler, Assembler und Linker

Wie wird der C-Quellcode übersetzt?

Beispiel

```
gcc main.c
```

- Präprozessor: Makroprozessor, Textersetzung, bedingtes Kompilieren
- Compiler: Erzeugen von Assemblercode
- Assembler: Erzeugen von Maschinencode
- Linken (Binden) von verschiedenen Objekten

→ Ausführbare Datei

Zur Erinnerung

Jedes Programm in Java benötigt eine Main-Methode:

```
1     public static void main(String [] args)
```

Die main-funktion

Der Standard definiert zwei zulässige Deklarationen:

1

- `int main()`
- `int main(void)`

2

- `int main(int argc, char* argv[])`
- `int main(int argc, char** argv)`

Kommentare

- `/* ... */`
- `// ...` als Erweiterung, nicht in C90

Basistypen

- char
- int
- float
- double

Typmodifizierer (type modifiers)

- unsigned
- long
- short

Weitere Modifizierer

- const
- static
- extern
- (auto, volatile, register)

if-else

Bedingter Programmfluss, Fallunterscheidungen:

```
1 if(i < 0)
2 {
3     /* do something */
4 }
5 else
6 {
7     /* do something else */
8 }
```

switch

Auswertung vieler verschiedener Zustände:

```
1  switch(i)
2  {
3      case 0:
4          /* ... */
5          break;
6      case 1:
7          /* ... */
8          break;
9      /* ... */
10     default:
11         /* ... */
12         break;
13 }
```

switch

Hinweise für switch

- `break` nicht vergessen
- einfaches Abfangen von Fehlern mit `default`

while-Schleife

Bedingte Wiederholung:

```
1 while(1)
2 {
3     /* do something */
4 }
```

for-Schleife

N-malige Wiederholung:

```
1 for(unsigned int i = 0; i < N; ++i)
2 {
3     /* do something */
4 }
5
6 /* vor C99: */
7 unsigned int i;
8 for(i = 0; i < N; ++i) { }
```

Syntax

- Rückgabetyp
- Name
- Parameterliste
- Funktionsrumpf

```
1 int square (int number)
2 {
3     return number * number;
4 }
```

Was sind Zeiger?

- Verweise (Referenzen) auf Objekte
- Speichern die Adresse von Objekten
- Definition eines Zeigers mit `Typ*`

Was sind Zeiger?

- Der Adressoperator `&` liefert die Adresse einer Variable
- Der Dereferenzierungsoperator `*` liefert den Inhalt einer Adresse
- `*` und `&` sind komplementär
- Es gilt: `*(&x) == x`

Was sind Zeiger?

Beispiel

```
1 int a = 5;
2 int* ptr;  /* Zeiger auf int */
3 ptr = &a;  /* Zuweisung der Adresse von a (&a) */
4 *ptr = 42; /* Dereferenzierung */
5 printf("%d\n",a); /* 42 */
```

Wozu werden sie benötigt?

Erst durch Zeiger ist call-by-reference in C möglich:

```
1 void change (int* ptr)
2 {
3     *ptr = 42;
4 }
5
6 int i = 0;
7 change(&i); /* uebergabe der Adresse */
8 printf("%d\n",i);   /* 42 */;
```

Der NULL-pointer

NULL: Abwesenheit eines gültigen Wertes/einer gültigen Adresse.

Beispiel

```
1 int* ptr = ...;
2
3 if(ptr == NULL)
4 {
5     /* error */
6 }
```

Mit Zeigern kann man rechnen

```
1 /* ptr: Zeiger auf int */
2
3 /* dereferenzieren eines Elements 2 Einheiten weiter
      */
4 *(ptr + 2) = 42;
5 /* inkrementieren des Zeigers */
6 ++ptr;
```

Mit Zeigern kann man rechnen

Differenzen von Zeigern

```
1 int array[10] = { 1,2,3,4,5,6,7,8,9,10};
2 int* left = &array[3];
3 int* right = &array[7];
4 unsigned int diff = right - left;
```

enum

Aufzählungstyp, Vermeidung von "Magic Numbers"

```
1 enum Wochentag
2 {
3     montag, dienstag, mittwoch, donnerstag,
4     freitag, samstag, sonntag
5 };
6 Wochentag t = montag;
```

arrays

Felder eines Typs

```
1 int array [] = { 1,2,3,4,5,6,7,8,9 };
2 /* indexzugriff */
3 array[3] = 27;
```

strings

strings: arrays von `char`

```
1 /* konstanter Zeiger auf char */
2 const char* str1 = "Hallo Welt";
3 /* Zeiger auf char (schlecht) */
4 char* str2 = "foobar";
5 /* char array; beachte die terminierende 0 ('\0') */
6 char str3 [] = { 's', 't', 'r', 'i', 'n', 'g', '\0' };
```

struct

```
1  struct datei
2  {
3      unsigned int size;
4      char* name;
5  };
6
7  /* Definition einer neuen Variable datei */
8  struct datei d;
9  /* Elementzugriff */
10 d.name = "main.c"
11 d.size=23;
```

Mit Vorzeichen

- `int8_t`
- `int16_t`
- `int32_t`
- weitere (64, 128, ...)

Ohne Vorzeichen

- `uint8_t`
- `uint16_t`
- `uint32_t`
- weitere (64, 128, ...)

Zahlen statt Arrays verwenden um platzsparend Wahrheitswerte zu speichern:

```
1 uint32_t status = 0;
2 /* Setzen des 3. Bits (es gibt ein 0. Bit)  */
3 status = status | (1 << 3);
4 /* Loeschen des 12. Bits */
5 status = status & ~(1 << 12);
```

Mehrdimensionale Arrays

Anlegen eines eindimesionalen Arrays, umrechnen der Indizes:

```
1 const uint8_t outerLength = 5;
2 const uint8_t innerLength = 10;
3 uint8_t array [outerLength * innerLength];
4 /* zugriff auf array [1,2] */
5 uint8_t i = 1;
6 uin8_t j = 2;
7 array[innerLength * i + j];
```

Zwingende Angabe von `struct`:

```
1 struct structure
2 {
3     int a;
4     int b;
5 };
6
7 struct structure name;
```

Besser mit `typedef`:

```
1 typedef struct
2 {
3     int a;
4     int b;
5 } structure;
6
7 structure name;
```

Schlechte Einrückung

Einrückung beachten

```
1 if(i < 0)
2 {
3 i = -i;
4 if(*ptr == 0)
5 {
6 break;
7 }
8 }
```

Gute Einrückung

Einrückung beachten

```
1 if(i < 0)
2 {
3     i = -i;
4     if(*ptr == 0)
5     {
6         break;
7     }
8 }
```

Aussagekräftige Variablennamen

```
1  int16_t* a;
2  uint8_t i,j;
3  uint8_t l1, l2;
4  uint8_t c;
5  for(i = 0; i < l1; ++i)
6  {
7      for( j = 0; j < l2; ++i)
8      {
9          if(array[i][j] == l2)
10         {
11             ++c;
12         }
13     }
14 }
```

Aussagekräftige Variablennamen

```
1  int16_t* array;
2  uint8_t i,j;
3  uint8_t outerLength, innerLength;
4  uint8_t counter;
5  for(i = 0; i < outerLength; ++i)
6  {
7      for( j = 0; j < innerLength; ++i)
8      {
9          if(array[i][j] == innerLength)
10         {
11             ++counter;
12         }
13     }
14 }
```

Kommentare reichlich verwenden

```
1  while(1)
2  {
3      *ptr = complexCalculationFunction(*ptr++);
4      if(*ptr < -27)
5      {
6          error("Array exceeds limits!");
7          exit(1);
8      }
9      int* a = malloc(320);
10 }
```

```
1  while(1)
2  {
3      /* ptr is a pointer to an array of samples */
4      /* update the actual value and increment the
           pointer */
5      *ptr = complexCalculationFunction(*ptr++);
6      /* -27 is the lowest possible value, see
           lib_limits.c */
7      if(*ptr < -27)
8      {
9          /* error just raises an errormessage, we have
               to terminate the program ourself */
10         error("Array exceeds limits!");
11         exit(1);
12     }
13     /* we need exactly 320 bytes, since we need 80
           ints, with sizeof(int) == 4 */
14     int* a = malloc(320);
15 }
```

Quellen

- https://en.wikibooks.org/wiki/C_Programming
- Programmieren in C++ - Ulrich Breymann
- C-Kurs der Freitagsrunde - https://wiki.freitagsrunde.org
- Microcontroller und C Einführung des Projektlabors