C Repetitorium

- Für das Praktikum „Betriebssystem“ sind C-Grundlagenkenntnisse im Umfang der Erstsemestervorlesung „Grundlagen der Programmierung“ oder „Informatik 1“ erforderlich.
- Im folgenden wird noch einmal auf einige (wenige) Aspekte von C eingegangen, die mir im Rahmen des Praktikums wichtig erscheinen.
- Weitergehende Literatur: z.B.
  - Kerninghan/Ritchie: Programmieren in C, Hanser Verlag, 1990
  - Gervens et al: Vorlesungsskript Grundlagen der Informatik, FH Osnabrück, 2009
  - ....

C Repetitorium

Grundlegendes zur Programmiersprache C:

- C ist eng mit UNIX verbunden, 90% von UNIX und der Großteil von Windows ist in C geschrieben
- Entstanden aus den ersten UNIX-Sprachen (Sprache B)
- Entwickler: Ken Thompson, Dennis Ritchie (Urväter von UNIX)
- C ist die ideale Sprache zur Systemprogrammierung
- Vorsicht: Die Mächtigkeit von C ist gleichzeitig die Quelle von vielen Fehlern!
  Der Programmierer ist für sein System verantwortlich!
  (Stichworte: Speicherverwaltung, Zeiger ....)

C Repetitorium

Prinzipieller Aufbau eines C Programms:

```
#include <stdio.h>

/* Dies ist ein Kommentar */
int main(int   argc,
         char *argv[],
         char *env[])
{
  int returnCode = 0;
  printf("Hallo Welt");

  return returnCode ;
}
```

Headerdateien einbinden (enthalten Typdefinitionen, Variablen und Funktionsdeklarationen)

Kommentare ausschließlich in `/* */` einschließen.
(`//` ist C++ Stil, wird aber den *meisten* C Compilern verstanden)

Funktionen: Syntax
Deklaration (in Headerdatei oder in C Datei):

```
<Funktionstyp><Funktionsname>(
<Parameterliste>);
```

Definition:

```
<Funktionstyp><Funktionsname>(
<Parameterliste>)
{
  <Vereinbarungen>;
  <Anweisungen>;
  return (<Ausdruck>);
}
```

C Repetitorium

Übersetzen eines C Programms:

Im Rahmen des Betriebssysteme Praktikums wird mit dem GNU C Compiler gearbeitet!

Syntax (nur mit den minimalen Schaltern):
gcc –o <progname> <sourcename> <*optionen*>
Beispiel:

```
>gcc –Wall –o demo demo.c
```

Alle Programme sollten ohne Warnings übersetzt werden!

Werden mehrere C Dateien für ein ausführbares Programm benötigt:

```
# Erzeugt Objektdatei <file1.o>:
gcc –c <file1.c>
# Erzeugt Objektdatei <file2.o>:
gcc –c <file2.c>
# Binde die Objektdateien zu einer ausfuehrbaren Datei:
gcc –o <progname> <file1.o> <file2.o>
```

C Repetitorium

Übersetzen eines C Projektes mit make: makefile

Beispiel für ein einfaches makefiles:

```
PRODUCT=demo

# Tool Definition
CC=gcc
CCFLAGS=-DDEBUG –Wall –g –c
LNKOPT=-lm

# file definition
#
OBJECTS=file1.o file2.o \
   file3.o

# dependencies:
file1.o:  file1.c file1.h \
   file2.h
file2.o: file2.c file1.h
file3.o: file3.c
```

```
# rules
$(PRODUCT): $(OBJECTS)
   $(CC) $(LNKOPT) –o  \
   $(PRODUCT) $(OBJECTS)

# pattern rules:
%.o: %.c
   $(CC) $(CCFLAGS) -o $*.o $<

all: $(PRODUCT)

clean:
   @-rm *.o
   @-rm *~
```

C Repetitorium

Übersetzen eines C Projektes mit make:

- `make all`
  Kompiliert alle C-Dateien (`file1.c, file2.c, file3.c`) und bindet die Objektdateien (`file1.o, file2.o, file3.o`) zu dem ausführbaren Programm (`demo`)
- `make clean`
  Löscht alle .o Dateien und Backupdateien

Stolperfallen:

- Am Anfang einer Zeile müssen Tabs verwendet werden, Leerzeichen führen zu Fehler.
- Als Zeilentrenner muss \ (Backslash) verwendet werden.

C Repetitorium

Beispiel: Aufbau eines C Programms (mit Kodierregeln, die in anderen Umfeldern anders sein können) (1):

```
#include <stdio.h>
#include "myTypes.h"

/**************************************
* demo.c
*
* Purpose: Just demonstrates some C constructs
*
* Author: J. Wuebbelmann
*
* History: 20061113: JW: Created File
*          20090402: JW: Changed types
*
*
**************************************/
```

Standardheader werden in < > eingeschlossen, eigene Header in " "

Kodierregel:
Jede Datei muss einen Kommentarkopf mit Name, Zweck,
Autor und Historie enthalten!

C Repetitorium

Aufbau eines C Programms (2):

```
/* Definitionen:     */
/* Beispiel:  */
#ifdef DEBUG
#define DEBUGOUT printf
#else
#define DEBUGOUT
#endif


/* Typ Definitionen */
/* Beispiele:    */
/* Strukturen  */
typedef struct errorCodes
{
   ErrorEnums  error;
   char*       errorString;
}
ErrorCodes;

/* Einfache Typen */
typedef unsigned int uint32_t;
/* Ende der Typ Definitionen */
```

Alle Definitionen, die für das ganze Programm gelten, stehen am Anfang des Programms!

Sollen Strukturen in mehreren Dateien verwendet werden, dann besser in Header! (z.B. hier: ErrorEnums ist in myTypes.h definiert.)

Definition eigener Datentypen erlaubt Maschinenunabhängigkeit! In Standardheadern und Systemaufrufen wird fast ausschließlich mit eigenen Typen gearbeitet. (Beispiel: `size_t`). Nutzen Sie die Typen aus stdint.h (ab C99 standardisiert)

C Repetitorium

Aufbau eines C Programms (3):

```
/* globale Variablen */
/* Beispiel: */
/* kann in einer anderen C Datei mit
extern ErrorCodes* errorCodes;
deklariert werden */
ErrorCodes  errorCodes[] =
{
 { noError,      "No Error"},
 { stringError, "invalid String"},
 { fileError,    "Error: handling a File"},
 { openError,    "Error: open a file"},
 { maxErrorNo,  "Errornumber exceeds range"}
};

const char version[] = "Version 0.2";
const char author[]  = "J. Wuebbelmann";

/* Externe Funktionen:           */
/* Beispiel: (funktion gibt es nirgends..)*/
extern uint32_t myExternFunction(uint32_t value,
                                 uint32_t* reference);
```

Kodierregel:
Wenn möglich, auf globale Variablen verzichten.
Das Schlüsselwort const wann immer möglich verwenden.

externe Funktionen oder Variablen können auch in Headerdateien deklariert sein.

C Repetitorium

Aufbau eines C Programms (4):

```
/* Funktions Deklaration  */
/* Beispiel fuer call by value*/
ErrorEnums myInternFunction(ErrorEnums xcode);
/* Beispiel fur Call by Reference  */
uint32_t myGetStringLength(char*       string,
                           ErrorEnums* xcode);

/*****************************************
* function: main()
* purpose: starting point
* parameters:
*  argc  (in): number of arguments
*  argv[](in): array of arguments
*  env[] (in): array of environment variables
*
* returns:
*  int: exit status
*
*****************************************/
int main(int argc, char *argv[], char *env[])
{
```

Alle Funktionen müssen vor der ersten Benutzung deklariert werden.

Kodierregel:
Alle Funktionen müssen kommentiert werden.
Inhalt: Zweck, Parameter (Ein- oder Ausgabe?), Rückgabewerte.

C Repetitorium

Aufbau eines C Programms (5):

```
int main(int argc, char *argv[], char *env[])
{
  ErrorEnums xcode  = noError;
  uint32_t   i      = 0;
  uint32_t   length = 0;

  DEBUGOUT("name: %s release: %s by: %s\n",
   argv[0], version, author);

  printf("Print environment: \n");
  while (NULL != env[i])
  {
    printf("  env[%d] = %s \n", i, env[i]);
    i++;
  }

  printf("Print arguments: \n");
  for (i = 0; i < argc; i++)
  {
     printf("  argv[%d] = %s \n", i, argv[i]);
  }
```

Kodierregel:
`main()` muss mit Rückgabewert (`int`) und optional mitParameterliste implementiert werden

Debug Ausgabe hier mit bedingter Übersetzung (s.o.)

Kontrollstrukturen:
`while()` und `for()` Schleifen...

Kodierregel:
Einrückungen bei allen Blöcken ist Pflicht! (am besten mit Leerzeichen, keine Tabulatoren)

C Repetitorium

Aufbau eines C Programms (6):

```
  /* test function:  */
  i = 0;
  do
  {
    xcode = myInternFunction((ErrorEnums)i++);
  }while (noError == xcode);
  printf ("  main: %d: errorstring: %s\n",
      xcode, errorCodes[xcode].errorString );
  /* zweiter Test: */
  xcode = noError;
  i = 0;
  do
  {
    length = myGetStringLength(env[i],&xcode);
    if (noError != xcode)
    {
      printf("Error: env[%d]: %s\n", i,
              errorCodes[xcode].errorString );
    }
  } while (NULL != env[i++]);
  return 0;
}
```

weitere Kontrollstrukturen: `do while();` Schleifen, `if () ;else ;` Strukturen, `switch()` `case:`

Call by value:
Wert der Variable wird der Funktion übergeben.

Call by Reference:
Zeiger auf Variable übergeben. Die Funktion kann Inhalt der Variable ändern.

C Repetitorium

Aufbau eines C Programms (7):

```
/*****************************************
* function: myInternFunction()
* purpose: demo function for call
*          by value
* parameters:
*  xcode (in)  : errorcode to check
*
* returns:
*  ErrorEnums: exit status
*
*****************************************/
ErrorEnums myInternFunction(ErrorEnums xcode)
{
  if (maxErrorNo <  xcode )
  {
    xcode = maxErrorNo;
  }
  else
  {
   printf ("  myInternFunction: %d: errorstring: %s\n",
            xcode, errorCodes[xcode].errorString );
    xcode = noError;
  }
  return xcode;
}
```

Kodierregel:
Pflichtteil
Kommentarblock!

Call by value:
Wert der Variable wird der Funktion übergeben.
Eine Manipulation der Variablen in der Funktion hat keinen Einfluss auf den Wert in der rufenden Funktion.

Kodierregel:
Wertebereichsüberprüfung aller Parameter!

C Repetitorium

Aufbau eines C Programms (8):

```
/*****************************************
* function: myGetStringLength()
* purpose: demo function for call
*          by reference
* parameters:
*  string (in) : test string
*  xcode (out) : exit status
*
* returns:
*  uint32_t: length of test string
*
*****************************************/
uint32_t myGetStringLength(char* string, ErrorEnums* xcode)
{
  uint32_t length = 0;
  if (NULL != string)
  {
    length = strlen(string);
    *xcode = noError;
  }
  else
  {
    *xcode = stringError;
  }
  return length;
}
```

Call by Reference:
Zeiger auf Variable übergeben. Die Funktion kann Inhalt der Variable ändern.
In Kommentar:
als (out) markieren.
Im Code:
Zugriff über Dereferenzierungs-operator (*).
Häufig werden Zeiger auf Strukturen übergeben.

C Repetitorium

Beispiel: Aufbau einer C Header Datei:

```
#ifndef MYTYPES_H
#define MYTYPES_H
/**************************************
* myTypes.h
*
* Purpose: Demonstrate some C constructs
*
* Author: J. Wuebbelmann
*
* History: 20061113: JW: Created File
*
**************************************/
/* einige Definitionen           */
/* Typ Definitionen              */
/* Beispiele:  Enumeration Feld */
typedef enum errorEnums
{
  noError = 0,
  stringError,
  fileError,
  openError,
  maxErrorNo
} ErrorEnums;

#endif  /* MYTYPES_H */
```

Kodierregel:
Bedingte Übersetzung in der Headerdatei verhindert Mehrfacheinbindung und damit Compilerfehler.

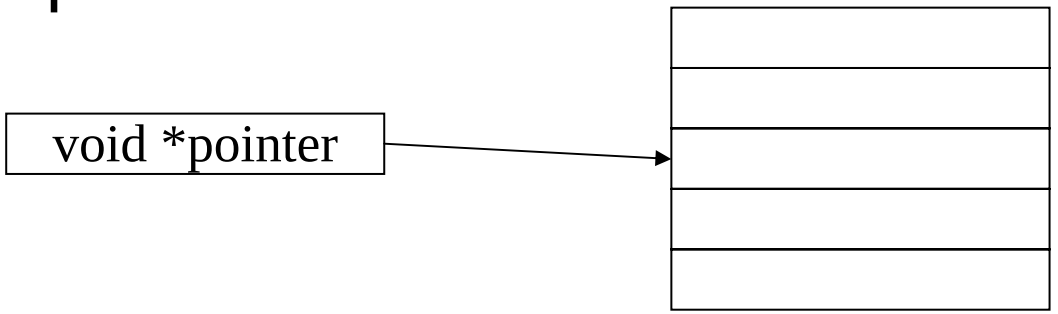
Kodierregel:
Pflichtteil Kommentarblock (Wer, wann, wozu...)

Darf Typdefinitionen, Variablendeklarationen und Funktionsdeklarationen enthalten. (Alles, was keinen Speicher belegt. Code theoretisch auch möglich, bei Einbindung in mehrer C-Dateien wird es aber Linker Fehler geben!)

C Repetitorium

Häufige Probleme: Zeiger / Strings / Arrays

- Zeiger (*Pointer*) enthalten Speicheradressen. Sie ermöglichen das Beschreiben und Lesen von Speicherbereichen.



Beispiel:

```
uint32_t *pointer;  // legt einen pointer
                    // auf uint32_t an
pointer = 100; // der pointer  zeigt auf
               // Adresse 100
*pointer = 15;  // es wird eine 15 in
          // (uint32_t*) 100 geschrieben
```

```
pointer++;  // der Zeiger wird um ein
            // Element vom Typ uint32_t
            // erhoeht. Er zeigt auf
            // Adresse 104
(*pointer)++; // der Inhalt des 32 Bit
              // Worts ab Adresse 104
              // wird inkrementiert
```

C Repetitorium

Häufige Probleme: Zeiger

- ‚Typische‘ Fehler in C sind nicht initialisierte Zeiger.
- Tipps:
  - Existiert bei der Vereinbarung eines Zeigers noch kein gültiger Wert für den Zeiger, sollte er mit NULL initialisiert werden.
  - Vor der Benutzung eines Zeigers sollte er auf NULL überprüft werden (Fehlerbehandlung!).

Beispiel:

```
void getData(void *data, size_t length)
{
  if (NULL != data) read(STDIN_FILNO, data, length);
  else panic("Null pointer detected");
}
```

C Repetitorium

Häufige Probleme: Strings / Arrays

- Ein Array reserviert einen linearen Speicherbereich für Daten.
- Der Compiler überprüft nicht, ob alle Arrayzugriffe sich auf den reservierten Speicherbereich beschränken!
- Der Programmierer ist dafür verantwortlich, dass der Speicher konsistent bleibt!
  (Die Mächtigkeit von C, alles zu können verlangt vom Programmierer, sehr genau zu wissen, was er/sie tut.)

C Repetitorium

Häufige Probleme: Strings / Arrays. Beispiel:

```
uint32_t getData()
{
  char firstName[10]    = {0};
  char lastName[10]     = {0};
  char placeOfBirth[10] = {0};

  printf("Enter first name\n");
  gets(firstName);

  printf("Enter last name\n");
  gets(lastName);

  printf("Born where?\n");
  /* the correct way:  */
  fgets(placeOfBirth,10,stdin );

  /* do a lot more */
  return 0;
}
```

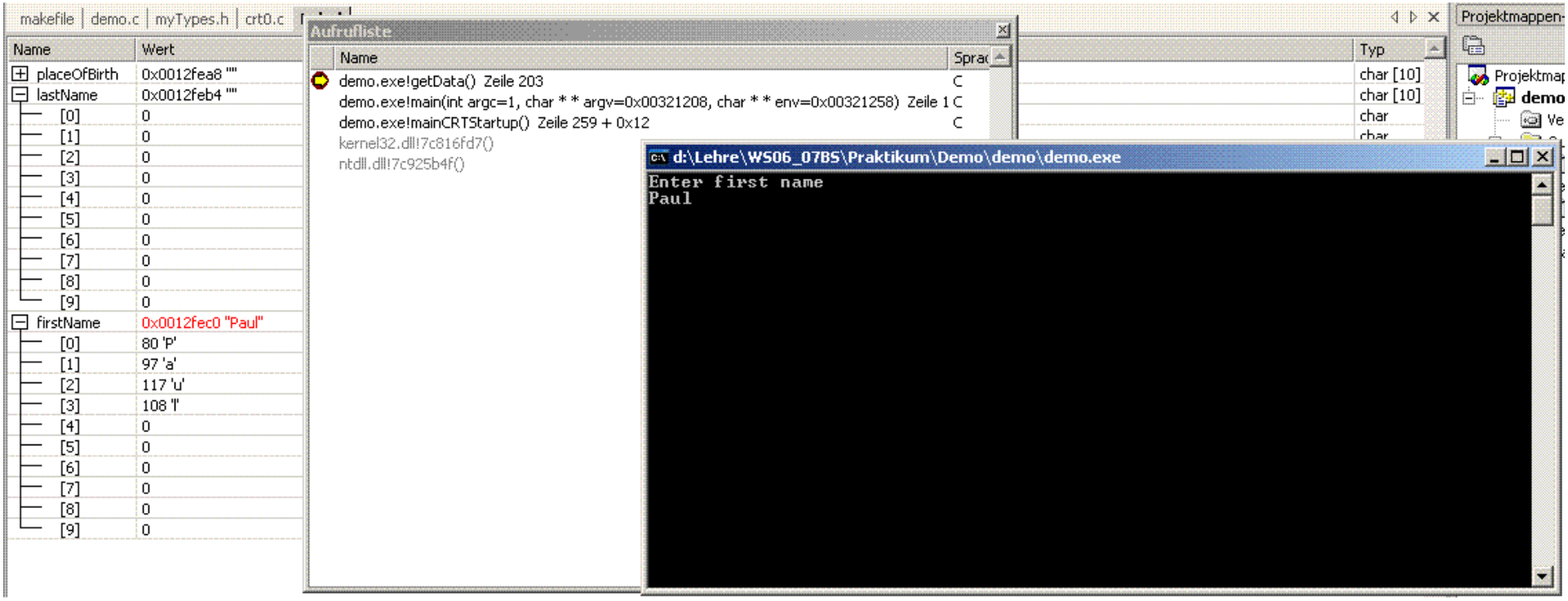
Reserviert 10 Elemente vom Typ `char`.
(Entspricht 10 Byte)
Erlaubte Indizes:
`firstName[0]` .. `firstName[9]`
(`firstName[10]` ist schon verboten!)
`firstName` ist ein Zeiger auf das erste Element des Arrays!
Da bei einem String die \0 am Ende gespeichert wird, stehen 9 Zeichen zur Verfügung!

`gets()` liest einen String von stdin.
Es findet keine Längenüberprüfung statt!
(`gets()` sollte nie verwendet werden, statt dessen kann `fgets()` benutzt werden. `fgets()` beschränkt die Stringlänge.
Auch `scanf()` kann problematisch sein!)

C Repetitorium

Häufige Probleme: Strings / Arrays: Beispiel:

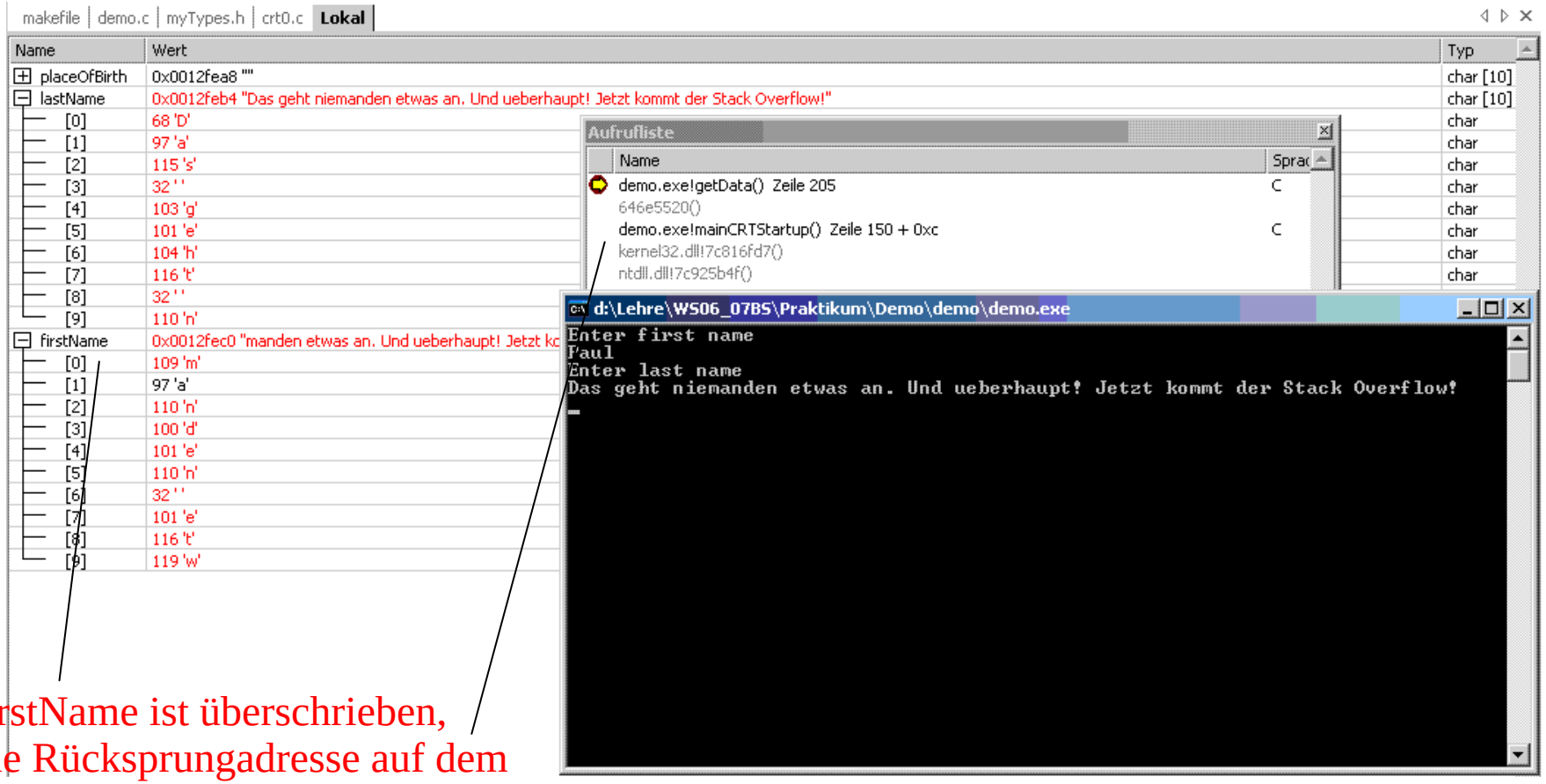
Status nach Eingabe des Vornamens (4 Zeichen)



C Repetitorium

Häufige Probleme: Strings / Arrays: Beispiel:

Status nach Eingabe des Nachnamens (76 Zeichen):



firstName ist überschrieben,
die Rücksprungadresse auf dem
Stack ist korumpiert.

C Repetitorium

Häufige Probleme: dynamischer Speicher:

- Dynamischer Speicher kann mit `malloc(size_t size)` vom Betriebssystem angefordert werden.
  Es ist zu überprüfen, dass der Speicher alloziert werden konnte!
- Wie bei Arrays wird auch bei dynamischem Speicher nicht überprüft, ob die Grenzen des angeforderten Speichers eingehalten werden!
- Der Speicher muss nach Benutzung mit free() wieder freigegeben werden, sonst entstehen Speicherleichen.

C Repetitorium

Häufige Probleme: dynamischer Speicher. Beispiel:

```
void func()
{
  uint32_t* mem = NULL;
  mem = getUint32Mem(100);
  if (NULL == mem)
  {
    panic("Null pointer detected");
  }
  else
  {
    /* do a lot with mem */
    free(mem);
  }
}

uint32_t* getUint32Mem(size_t size)
{
  uint32_t* mem = NULL;
  mem =
   (uint32_t*)malloc(size*sizeof(uint32_t));
  if (NULL == mem)
  {
    panic("Null pointer detected");
  }
  return mem;
}
```

Überall, wo ein Zeiger geholt wird, muss er überprüft werden!

Der Speicher muss wieder freigegeben werden! (`free(NULL)` ist erlaubt.)