Praxis der Programmierung

Konstanten, Strings, Programmparameter

Institut für Informatik und Computational Science
Universität Potsdam

Henning Bordihn

Einige Folien gehen auf A. Terzibaschian zurück.

Fehlerarten

- **Compilerfehler**: Fehler, die der Compiler erkennt, z.B.
  - Semikolon vergessen
  - Variable benutzt aber nicht definiert etc.

  $\leadsto$ meist kein ausführbares Programm

- **Laufzeifehler**: Fehler, die bei der Abarbeitung des Programms auftreten, z.B.
  - Division durch 0
  - fehlende Werte
  - Zugriff auf Datei, die nicht geöffnet werden kann
  - kein Speicherplatz mehr vorhanden etc.

- **logische Fehler**: Fehler im Programmentwurf

Konstanten

Konstanten

- Literale (vordefinierte Konstanten elementarer Typen)
- Variablen, die mit Typattribut `const` definiert sind
  - `const` *Datentyp Bezeichner*;
  - Variable nach Initialisierung schreibgeschützt
  - Beispiel: `const double PI = 3.1415927`
  - Konvention: Bezeichner aus Großbuchstaben
  - für Variablen, Pointer, Parameter

$\leadsto$ Schutz vor unbeabsichtigten Änderungen

$\leadsto$ "Dingen einen namen geben"
(z.B. PI, ROT, GELB, MONTAG, DIENSTAG, ...)
$\leadsto$ vereinfacht Lesbarkeit und Wartung des Quellcodes

Konstanten (2)

- Aufzählungstypen
- Definition mit Schlüsselwort `enum` und "Mengenschreibweise"
  ⇝ Definition eines neuen Datentyps mit endlich vielen konstanten Werten
- Deklaration von Variablen dieses Typs mit Schlüsselwort `enum`

```
enum ausbildung {
       HAUPTSCHULE,
       REALSCHULE,
       ABITUR,
       BERUFSAUSBILDUNG,
       BACHELOR,
       MASTER
};
```

```
int akademiker (enum ausbildung a) {
     if (a == BACHELOR
      || a == MASTER)
               return 1;
     else
               return 0;
}
```

Konstanten (3)

- symbolische Konstanten
  - `#define` *Name Wert*
  - Präprozessor ersetzt *textuell* alle Vorkommen von *Name* durch *Wert*
  - Beispiel: `#define PI 3.1415927`
  - Konvention: *Name* aus Großbuchstaben

⇝ keine Typprüfung durch den Compiler

Strings

Zeichenketten (Strings)

- String = Zeichenkette
- statische (unveränderliche) Strings in Anführungszeichen definiert
  - bisher als Parameter von `printf` und `scanf`
  - `printf("Ich bin ein String.");`
- Zeichenkette ist Folge von Character-Werten

$\rightsquigarrow$ Array vom Typ `char`

Strings (2)

- sind `char`-Arrays mit **Nullzeichen** `'\0'` als Markierung des Stringendes
- Viele String-Funktionen benötigen das Nullzeichen.
  ⇝ bei Definition des Arrays einplanen!

  ```
  char vorname [6] = {'N', 'a', 'd', 'j', 'a', '\0'};
  ```

- Initialisierung mit konstanter Zeichenkette:

  ```
  char vorname [6] = "Nadja";
  ```

  ⇝ automatisches Anfügen des Nullzeichens

- *Erinnerung:* Arrays sind Pointer auf das erste Element

  ```
  char *vorname = "Nadja";
  ```

  (ist Pointer auf den ersten Buchstaben)

Statische *versus* dynamische Strings

- statische Strings
  - Definition mit String-Syntax: `char *str = "Hallo";`
  - Pointer auf statischen Speicherblock direkt im Binärcode
    - ⇝ String darf nur gelesen werden
    - ⇝ Zugriff z.B. auf `str[500]` kann zu schweren Fehlern führen u.U. Änderung am Maschinencode

- dynamische Strings
  - Definition als Array mit Größenangabe: `char str[1024];`
  - Normale Zugriffsmöglichkeiten wie bei allen Arrays
  - ⇝ Überschreiben einzelner Buchstaben im String möglich

Statische Strings benutzen

- Zugriff auf einzelne Zeichen, z.B.:

```
char *str = "Hallo";
char c1 = str[0];   //  == 'H'
char c2 = str[1];   //  == 'a'
char c3 = str[5];   //  == '\0'
```

- Ausgabe des gesamten Strings mit `printf`:
  `printf(str);` oder `printf("... %s ..., str);`

⇝ Formatelement `%s` zum Integrieren in formatierte Ausgaben, Übergabe eines `char`-Pointers (`str`)

Dynamische Strings benutzen

- wie statische Strings
- außerdem schreibender Zugriff auf einzelne Buchstaben
- Einlesen des Strings, z.B. mit `fgets`:

```
char str2[1024];
printf("Geben Sie Ihren Namen ein!");
fgets(str2, 1024, stdin);   // max. 1023 Zeichen (warum?)
                            // von stdin lesen
                            // und ab Adresse str2 speichern
```

Standardfunktionen zur Ein- und Ausgabe

- definiert in `<stdio.h>`
- `int printf (const char * format, ...);`
- `int puts (const char * s);`
  - schreibt übergebenen String `s` nach `stdout`
  - kopiert das Nullzeichen *nicht* mit
  - fügt ein `'\n'` an
- geben die Länge der ausgegebenen Strings zurück
- `const char *` $Name \rightsquigarrow$ Array-Elemente konstant
  `char * const` $Name \rightsquigarrow$ Pointer konstant

- `char * gets (char * s);`
  - liest von `stdin` bis `'\n'` oder **EOF** in `char`-Array `s`
  - ersetzt `'\n'` bzw. **EOF** durch das Nullzeichen
  - übergibt Pointer `s`

- `char * fgets (char * s, int size, FILE * stream);`
  - liest `size-1` Zeichen aus `stream` bis `'\n'` oder **EOF** und spreichert sie in `s`
  - `'\n'` wird mit gespeichert und `'\0'` angehängt
  - übergibt Pointer `s`

- *Warum ist* `gets` *im Vergleich zu* `fgets` *gefährlich?*

- `int scanf (const char * format, ...);`
  - Argumente nach dem Formatstring sind **Adressen von Variablen**,
  - Speichern der Werte aus `stdin` auf diesen Adressen
  - Anzahl und Typen der Formatelemente müssen zu den adressierten Variablen passen (sonst unbestimmtes Verhalten)
  - *keine* Steuerzeichen (wie `'\n'`) im Formatstring!

- Beispiel:

```
int zahl;

printf ("\nEingabe: ");
scanf("%d", &zahl);

printf ("\nDer Wert %d wurde eingelesen.\n", zahl);
```

- andere Zeichen als Formatelemente im Formatstring möglich:
  - `scanf()` liest diese Zeichen und ignoriert sie für den Rückgabe-String
  - stellt sie in `stdin` zurück
  - verhält sich so, bis `'\n'` gelesen wird

```
float t;
printf("Temperatur im Format xx C: ");
scanf("%f C", &t);
t = (9. * t) / 5. + 32.;
printf("\nTemperatur in Fahrenheit: %f F", t);
```

⇝ Kein Newline-Zeichen `'\n'` im Formatstring von `scanf()`!

Übergabe von Strings als Parameter

- **Übergabe** eindimensionaler Arrays an Funktionen:
  formale Parameter als
  - offenes Array oder
  - Pointer auf den Komponententyp
- Anwendung bei Übergabe von Zeichenketten (`char`-Array)

```
int lengthOfString(char * ar) {
   int i = 0, l = 0;
   while (ar[i] != '\0') {
      l++;
      i++;
   }
   return l;
}
```

```
int main() {
   char *s = "Hallo Du!";
   int n = lengthOfString(s);
   ...
}
```

Standardfunktionen zur Stringverarbeitung

- in Header-Datei `<string.h>` definiert

`size_t strlen (const char * s);`	Länge
`char * strcpy (char * dest, const char * src);`	Kopieren
`char * strcat (char * dest, const char * src);`	Anhängen
`int strcmp (const char * s1, const char * s2);`	Vergleichen

  (0 bei Gleichheit, d.h. wenn `*s1 == *s2`)

- Arbeiten bis zum Nullzeichen `'\0'`

- `size_t` vordefinierter Datentyp als Rückgabetyp des `sizeof`-Operator (ist meist `unsigned int` oder `unsigned long`)

Warum Stringfunktionen wie `strcpy` ?

- Aufgabe: Kopieren von String `src` in String `dest`

- naives Herangehen: `dest = src;`

  ⇝ Was passiert?

- Übergabe des Pointers

  ⇝ Jede Änderung an `dest` auch in `src` und umgekehrt

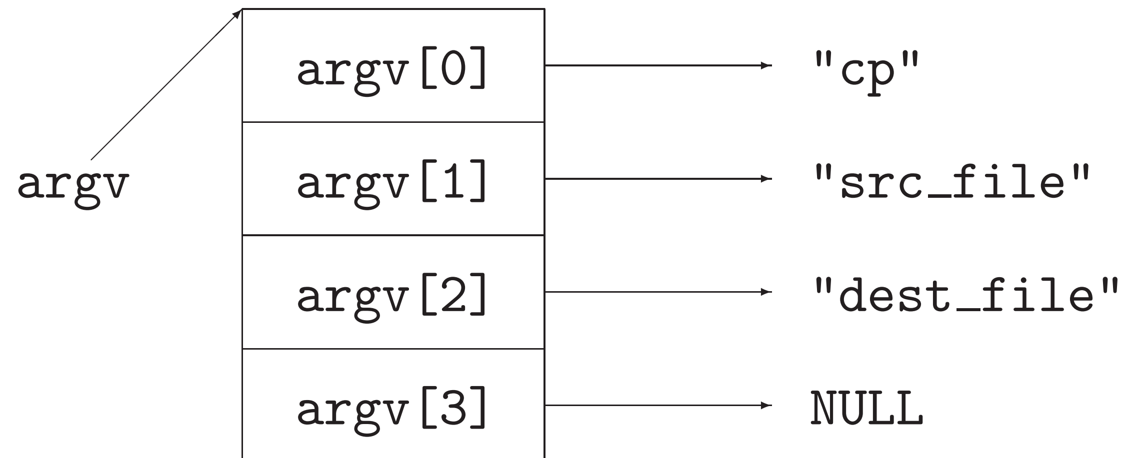
- `strcpy` ändert keinen Pointer,
  sondern kopiert den Inhalt von `src` an die Stelle `dest`

  ⇝ Verdopplung des Strings im Speicher

Parameterübergabe beim Programmaufruf

- Beispiel: `cp src_file dest_file`
- zwei Varianten der `main`-Funktion:
  - `int main()` — parameterlos
  - `int main(int argc, char * argv[])` — zwei Parameter
- `argc` (**arg**ument **c**ounter): Anzahl der Argumente
- `argv` (**arg**ument **v**ector): Vektor (Array) der Argumente

  Argumente sind Strings ⇝ Array von `char`-Arrays
  ⇝ Array von Pointern auf `char`
- erstes Element von `argv` (`argv[0]`): Programmname



- Übergabe von Zahlen: Typumwandlung String ⟶ Zahltyp erforderlich
- Standardfunktionen in `<stdlib.h>`

`double atof(const char * nptr);`	ascii to `float`
`int atoi(const char * nptr);`	ascii to `int`
`int atol(const char * nptr);`	ascii to `long`