C-Zeichenketten

Übersicht

- Umgang mit Zeichenketten
  - Eine spezielle Anwendung für Arrays
  - Ein Erbe aus C-Zeiten
- Typische Fallen (siehe auch Arrays)
  - Wie vergleiche ich "Strings"
  - Welche typischen Bibliotheksfunktionen werden gebraucht?
- Wie kann ich den Programm Aufruf auswerten
  - DOS:> Programm –i test.in –o test.out

Varianten von Zeichenketten

- Zeichenketten als `char*` sind eine Erbschaft aus C
  - effizient, umständlich zu handhaben, fehleranfällig
  - in Klassenbibliotheken gibt es Besseres:
    `class string` <string> Standard C++ Library
    `class CString` <afxwin.h> Microsoft Foundation Classes MFC

leider auch in C++ weitverbreitet, deshalb muß man sie kennen

- Zeichenketten in C sind 1-dim. Arrays von Zeichen mit '\0' als Endemarke (*zero terminated string*)
  - die Größe des Arrays muß der Programmierer überwachen
  - in der Praxis wird mit Zeichenketten meist über Zeiger gearbeitet
- häufigste Fehlerursache ist die fehlende Unterscheidung:
  - Zeichenkettenkonstante (String): `const char*`
  - Zeichenkettenvariable (StringBuffer): `char[27]` bzw. `char*`

Zeichenkettenkonstanten *String*

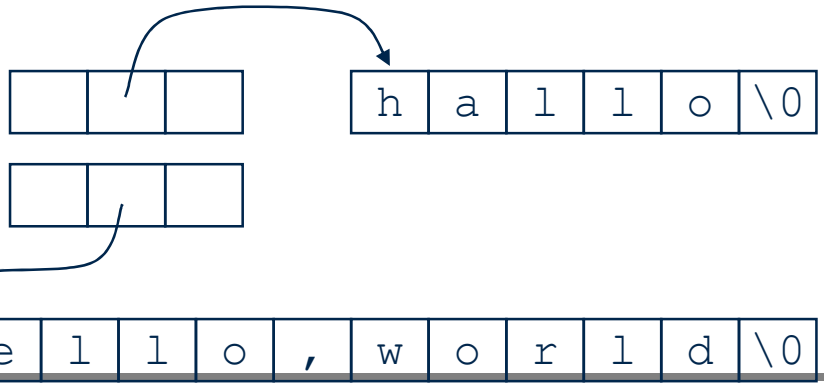
- vom Compiler wird automatisch ein Array passender Größe allokiert
  - immer ein Element mehr als der Text für die Endemarke '\0'
- `const` stellt sicher, daß das Array nicht versehentlich überschrieben wird
  - Funktionen, die Zeichenketten zurückgeben, akzeptieren keine `const`-Param.

als Array:

```
const char txt1[] = "hallo";
const char txt2[] = {'h','a','l','l','o','\0'};
```

als Zeiger:

```
const char* txt3 = "hallo";
const char* txt4 = "hello,"
                   "world";
```



Verkettung zur Compile Zeit

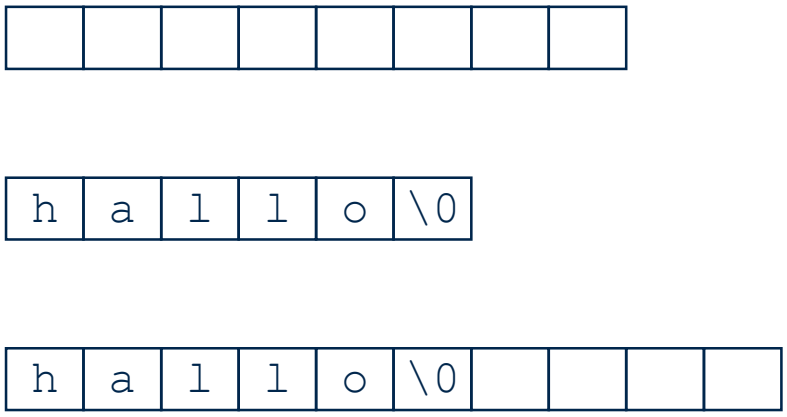
Zeichenkettenvariablen *StringBuffer*

- der Programmierer allokiert ein Array von **wohlüberlegter** Größe
  - zusätzliches Element für die Endemarke '\0' nicht vergessen !
- Funktionen, die Zeichenketten zurückgeben, wollen meist die Größe des verfügbaren Arrays wissen
  - stellen damit intern sicher, daß nicht über das Ende des Arrays hinaus geschrieben wird

```
// nicht initialisiert:
char Buffer1 [8];

// Länge automatisch:
char Buffer2 []    = "hallo";

// nur teilweise belegt:
char Buffer3 [10] = "hallo";
```



Problematische Operationen

mit Vorsicht einzusetzen

- Vergleich: == und !=
  - vergleicht nicht Strings, sondern Zeiger auf char
  - stattdessen: `i=strcmp(cs,ct)`
- Zuweisung: =
  - kopiert keinen String, sondern einen Zeiger auf char
  - stattdessen: `strcpy(s,ct)`
  - Initialisierung und Zuweisung von Strings nicht verwechseln
- verwirrend: char* kann sein:
  - Zeichenkettenkonstante
  - Zeichenkettenvariable
  - Zeiger auf char

```
const char hallo[] = "Hallo";
const char* Text1 = &hallo[0];
const char* Text2 = "Hallo";

if (Text1==Text2) {
   // wird nie ausgeführt
}
if (strcmp(Text1,Text2)==0) {
   // so ist es richtig
}

Text2 = Text1;
Text2[1] = 'e';// ändert auch
Text1 !
if (Text1==Text2) {
   // ist jetzt true
}
```

Beispiel zum Kopieren

ganz wichtig:
1. Parameter ist Variable,
2. Parameter ist Konstante

- keine eigenen Operationen für Zeichenketten, nur die allgemeinen Array- und Zeiger-Operationen
- jede Zeichenkettenverarbeitung muß zeichenweise erfolgen

```
void StrCopy (char* z,
              const char* q)
{ int i = 0;
  while (q[i] != '\0') {
    z[i] = q[i];
    i++;
  }
  z[i] = q[i];  // Endemarke
kopieren
}
```

Indizierung von Zeigern ☹

```
const char* pHello = "Hello";
char message [20];    // groß genug ???
StrCopy (message, pHello);
```

```
void StrCopy (char* z,
              const char* q)
{
  while (*q != '\0')
    *z++ = *q++;    // Zeigerarithmetik
  *z = *q;          // Endemarke kopieren
}
```

```
void StrCopy (char* z,
              const char* q)

{
  while (*z++ = *q++);  //
}
```

faszinierend kurz, aber
kein Strukturiertes Programm

Bibliotheksfunktionen

Wo eine Konstante verlangt ist, darf man auch eine Variable einsetzen ("automatische Dereferenzierung"). Nicht aber umgekehrt !

eine Auswahl vordefinierter Zeichenkettenfunktionen:

```
#include <string.h>
#include <stdlib.h>
char s[len]; char t[len];     Zeichenkettenvariablen
const char *cs, *ct;          Zeichenkettenkonstanten
char c; int i; double d;
bool b;

s=strcpy(s,ct)          kopiert ct nach s
s=strcat(s,ct)          hängt ct an das Ende von s
i=strcmp(cs,ct)         Vergleich von cs und ct; i >,=,< 0 entspr. cs >,=,< ct
t=strstr(cs,ct)         t zeigt auf erstes Auftreten von ct in cs
i=strlen(cs)            liefert die Nutzlänge von cs (ohne abschließende \0)

d=atof(cs)              konvertiert cs in eine Gleitpunktzahl
i=atoi(cs)              konvertiert cs in eine Ganzzahl

b=isalpha(s[0])         wahr falls  s[0]  A – Z oder a – z.
...                     viele weitere
```

Vorsicht !
ist s groß genug ???

Ein-/Ausgabe mit `cin` und `cout`

```
Hello, World!
Hello, World!
Eine lange Zeichenkette
***feste Breite
```

- `cout` schon vielfach genutzt:
  - Zeichenkettenkonstanten, auch mit Sonderzeichen (z.B. `\n` für Zeilenvorschub)
  - Zeichenkettenvariable werden automatisch dereferenziert
  - Verkettung von Strings
  - Formatierung: `setfill`, `setw`
- `cin` liest Zeichenkette in Array
  - liest bis zum nächsten *whitespace*
  - erzeugt abschließende `\0`
  - Zeiger muß auf hinreichend großes char-Array zeigen !!!

```
const char* pHallo = "Hello";
cout << pHallo << ", World!\n";
cout << "Hello" << ", World!\n";
cout << "Eine lange "
        "Zeichenkette" << endl;
cout << setfill('*') << setw(15)
     << "feste Breite" << endl;

char Eingabepuffer [5];
char* pPuffer = &Eingabepuffer[0];
char* pNirwana;    // besser = NULL
cin >> Eingabepuffer;      // Array
cin >> pPuffer;    // Zeiger auf Array
cin >> pNirwana;   //
```

Behandlung von Eingabefehlern mit `cin`

- eingebaute Fehlerbehandlung:
  - `cin >>` liest nur Zeichenfolge, die zum Parametertyp paßt
- Diagnosemöglichkeit:
  - `(cin >> ...)` liefert 0 bei fehlerhafter Zeichenfolge
- verbleibendes Problem:
  - fehlerhafte Zeichenfolge steht noch im Eingabepuffer
  - weitere `cin` Anweisungen werden sofort ausgeführt
- Notwendige Zusatzbehandlung:
  - Zustands-Flags löschen
  - Eingabepuffer leeren

```
if (cin >> Wert) {
  // korrekt eingegeben;
  // Wert weiterverarbeiten
}
else {
  // Eingabefehler; cin wieder
  // in Grundstellung bringen:

  cin.clear(); // Flags löschen

  char Muell = '\0';
  do {           // Puffer leeren
    cin.get(Muell);
  } while (Muell !='\n' );
}                // bis Zeilenende
```

Flexible und robuste Eingabe mit `cin`

- Problem:
  - Einzulesender Datentyp ist im Programm festgelegt
  - was tun, wenn wahlweise eine Zahl oder ein Text eingebbar sein soll ?
- Lösung:
  - Einlesen als Zeichenkette
  - Analysieren und Konvertieren der Zeichenkette im Programm
- Vorsicht !
  - Pufferüberlauf abfangen, falls der Benutzer eine sehr lange Zeichenkette eingibt

damit schreibt getline nicht über das Pufferende hinaus

```
const int Laenge = 25;
char Puffer [Laenge];
cin.getline (Puffer, Laenge);
   // getline liest bis Zeilenende,
   // jedoch max. Laenge Zeichen

if (EingabeKorrekt (Puffer)){
   Konvertiere (Puffer);
}
else {
   while (cin.getline(Puffer,
                      Laenge))
      ; // Eingabe leeren
}
```

Einige Hilfsfunktionen für Eingabe mit `cin`

- Flexible Behandlung mit weiteren Bibl.-Funktionen
  - Vorrausschauendes Lesen ohne den Puffer zu löschen
  - Zurückschreiben eines bereits gelesenen Zeichens
  - Lesen bis ein bestimmter Buchstabe eingegeben wird
- Ziel:
  - Vermeidung (und Behebung) von Eingabefehlern

```
const int MaxLine = 128;
char CharIn[MAxLine];
char CharPeak;

cin.eatwhite()
CharPeak = cin.peak();

cin.getline(CharIn,MaxLine);
            // hängt '\0' an den String
cin.getline(CharIn,MaxLine,';');
            // ';' nicht in String und aus
            // Puffer gelöscht


CharPeak = cin.get() // ein Zeichen
cin.putback(CharPeak)

cin.get(CharIn,MaxLine,';');
            // es wird keine '\0' angehängt
            // ';' bleibt im Puffer
```

Bildschirmausgabe mit `printf`

C-Stil (statt `cout`)
eingebaute Typen teilweise präziser zu formatieren;
jedoch nicht erweiterbar

- variable Parameterliste; Ergebnis: Anzahl ausgegebener Zeichen
  `int printf (const char* format, ...)`
- 1. Parameter ist Formatstring mit Formatbezeichnern für Folgeparameter
  - Formatbezeichner und aktueller Parametertyp müssen zusammenpassen
  - Formatbezeichner: % [flags] [width] [.prec] [length_mod] type_char

auch in CString::Format

```
int i=1234; long j=5678;
float f=93.243; double d=34567.1;
char c='a'; char* s="Hallo";

printf ("-i=%6d; -i=%u\n",-i,-i);
printf ("j=%06lX -j=%lo\n",j,-j);
printf ("c=\'%c\' c=%xhex\n",c,c);
printf ("f=%f=%.4e\n",f,f);
printf ("d=%6.2f, d=%9.2e\n",d,d);
printf ("%s%6s%.2s%p\n",s,s,s,s);
```

liefert am Bildschirm:

```
-i= -1234; -i=64302
j=00162E -j=37777764722
c='a' c=61hex
f=93.242996=9.3243e+01
d=34567.10, d= 3.46e+04
Hallo HalloHa0062
```

Tastatureingabe mit `scanf`

C-Stil (statt `cin`)
keine Vorteile

- variable Parameterliste; Ergebnis: Anzahl eingelesener Werte
  `int scanf (const char* format, ...)`
- Übergabe mit Return -Taste
  - überliest Leerstellen, Tabulator und Zeilenvorschub
    - Eingabefelder und sonstige Zeichen durch Leerstellen trennen
    - sonstige Zeichen werden exakt so erwartet
  - konvertiert Eingabefelder gemäß Formatstring und weist Folgeparametern zu
    Formatbezeichner: % [suppress] [width] [length_mod] type_char

`int i; long j; float f;` `double d; char s1[9],s2[9];`	***Eingabe***	***Ergebnis***
`scanf ("%d %li",&i,&j);`	`1234  0562`	*ok*
`scanf ("%d %li",&i,&j);`	`1234  0xF3A9`	*ok*
`scanf ("%f ; %lg",&f,&d);`	`3.5e3;0.44`	*ok*
`scanf ("%f;%lg",&f,&d);`		*Fehler*
`i=scanf ("%s %s",&s1,&s2);`	`hugo fred`	*ok,i=2*
`i=scanf ("%s , %s",&s1,&s2);`	`hugo ;fred`	*Fehler,i=1*

Kommandozeilenparameter

- Das Betriebssystem übergibt beim Aufruf eine variable Anzahl von Parametern aus der Kommandozeile an `main`
  - `argc` liefert die Anzahl
  - `argv` ist ein Array von Zeigern auf Zeichenketten (liegen irgendwo)
  - `argv[0]` zeigt auf den Programmnamen aus der Kommandozeile

```
#include <iostream.h>

int main (int argc, char* argv[])
{
  assert (argc >= 1);   // Programmnamen gibt's immer
  cout << "Programmname: " << argv[0];
  for (int i=1; i<argc; i++) {
    cout << i << ". Parameter: " << argv[i] << endl;
  }
  return 0;                    // Fehlercode ans Betriebssystem: 0=ok
}
```

`string` aus Standard C++ Library

- **allokiert automatisch den benötigten Speicher**
- viele nützliche Funktionen:
  - Verkettung +
  - Vergleich ==, !=, <, <=, >, >=
- Typkonversion
  - const char* ⇨ string automatisch
  - string ⇨ const char* explizit mit string::c_str()
- Ein-/Ausgabe mit streams
  Operatoren >> und <<
  istream& getline (istream&,string&);

```
#include <string>
#include <iostream>
using namespace std;

void main ()
{
  string Str = "Hello";
  const char* Cs = Str.c_str();

  Str = Str + ", World";
  cout << Str << endl;
  cin >> Str;

  int PosXY = Str.find ("xy");
}
```

CString aus Microsoft Foundation Classes

- **allokiert automatisch den benötigten Speicher**
- viele nützliche Funktionen:
  - Verkettung +
  - Vergleich ==, !=, <, <=, >, >=
  - mächtige Formatierung à la printf mit CString::Format (...)
- automatische Typkonversion
  - const char* ⇨ string
  - string ⇨ const char*
- Ein-/Ausgabe mit streams durch Konversion in C-Zeichenkette

```
#include <afxwin.h>
#include <iostream.h>

void main ()
{
  CString Str = "Hello";
  const char* Cs = Str;

  Str = Str + ", World";
  cout << Str << endl;
  cin >> Str;

  int PosXY = Str.Find ("xy");
}
```

In der Anwendung weitgehend gleichwertig mit string.
MFC gibt es schon länger als Standard C++ Library.