§4 Zeichen und Zeichenketten

Leitideen: *Jedes Zeichen aus dem Basiszeichensatz wird mittels einer kleinen ganzen Zahl (Länge 1 Byte = 8 Bit) gespeichert, die seiner Position im Zeichensatz entspricht.*

C-Zeichenketten sind nullterminierte C-Zeichenvektoren (d.h. letztes Zeichen enthält nur Nullbits). Dieses dient zur Längenbestimmung.

Die C++-Zeichenketten aus der STL (string) verhalten sich ähnlich wie STL-Vektoren aus Zeichen, zusätzlich steht eine Reihe von Funktionen und Operatoren zur Stringverarbeitung bereit.

- Datentypen und Literale I,II für Zeichen
- C-Zeichenketten
- Zeichenketten in C++, Fortsetzung I-IV

Datentypen und Literale für Zeichen I

- Interne Darstellung von Zeichen durch ganze Zahlen.
  Datentyp `char` ist ganzzahlig (in der Regel 8 Bit-Datentyp)
- Implementierung von `char` entweder durch
  `signed char` Zahlbereich z.B. $-128 \ldots 127$ *oder*
  `unsigned char` Zahlbereich z.B. $0 \ldots 255$
  g++-7.5 unter Ubuntu: `char` $\hat{=}$ `signed char`
- Druckbare Zeichen des Minimalzeichensatzes haben immer positive Werte.
  (ASCII: Druckbare Zeichen im Bereich $32 \ldots 126$)
- Literale für Zeichen: Schreibweise mit einfachen Apostrophen, z.B. `'A'` `'5'` `'*'` `' '`
- Ersatzdarstellungen vor allem für nicht druckbare Zeichen
  z.B. `'\n'` `'\0'` `'\10'` `'\xA'`
  `cout << 'A' << '\101' << '\x41'` gibt `AAA` aus.
- Ersatzdarstellungen für die Zeichen mit Sonderfunktionen
  `'\''` (Apostroph), `'\"'` (Doppelapostroph),
  `'\\'` (Backslash)

Datentypen und Literale für Zeichen II

Caveat

- Zeichenkonstanten werden mit einfachen Apostrophen, Zeichenkettenkonstanten mit Doppelapostrophen geschrieben.
- Die Funktionen für die Zeichenverarbeitung aus `cctype` haben als Erbe aus C `int`-Argumente und liefern `int`-Rückgabewerte.
  (*Grund:* In C `EOF` ("End of file") als Rückgabewert bei Dateiende und Fehlern bei einer Reihe von E/A-Funkt.)
  Deshalb: Bei Verwendung von `toupper` bzw. `tolower` ggf. Cast auf `char` erforderlich.
  Anmerkung 1: Es gibt lokalisierte (d.h. von der Sprachumgebung abhängige) Varianten dieser Funktionen mit den richtigen Datentypen.
  Anmerkung 2: Die aus C geerbten Funktionen sind zum Teil auf elementarer Ebene auch in den Klassen der STL vorhanden und tragen oft denselben Namen wie einfacher zu gebrauchende C++-Funktionen für den gleichen Zweck.

C-Zeichenketten

- Zeichenkettenkonstanten (Literale):
  "$z_0 \ldots z_{k-1}$" $\hat{=}$ {'z_0',...,'z_{k-1}','\0'}
- Zeichenketten in C : C-Vektoren aus `char`, enden mit `'\0'` (ASCII: NUL) .
  Deshalb: In Zeichenvektoren Platz für `'\0'` nicht vergessen!
  Bsp.: `char a[3]="**",b[16]="Zu langer String";`
  b a: 

  (*g++*−4.9 *amd*64)
- Länge der Zeichenkette mittels `'\0'` bestimmbar.
  (Wird in vielen Funktionen verwendet, z.B. `strlen`)

Caveat

- `strlen` liefert den vorzeichenlosen Ganzzahltyp `size_t` (oft: `unsigned int` oder `unsigned long`).
  Problematisch wegen vorzeichenloser Arithmetik.
- C-Zeichenkettenvergleiche mittels Funktionen (`strcmp`) und *nicht* mit Vergleichsoperatoren!
- Zuweisungen und direkte Kopien von C-Zeichenketten *nicht* möglich.

Zeichenketten in C++

Allgemeines

- Header-Datei: `string`
- C++-Zeichenketten (“Strings“) sind *nicht* nullterminiert
  aktuelle Länge wird im Datenobjekt abgespeichert und kann dynamisch verändert werden
  `s.size()` Anzahl der Zeichen von *s* (kein `\0` am Ende!)
- `\0` ist ein normales Zeichen in Strings – daher Vorsicht bei Umwandlung in C-Zeichenkette
- Diverse Konstruktoren für Strings vorhanden, insb. einer mit C-Zeichenkette als Parameter
  Bsp.: `string s("text"); // Vereinb. okay`
- *Kein* Konstruktor mit `char`-Parameter

  Bsp.:
  ```
  string s('c');    // Vereinb. nicht erlaubt
  string s(1,'c'); // Vereinb. okay
  ```

Anmerkung: Konstruktoren ermöglichen Vereinbarungen und bewirken ggf. implizite Typumwandlungen!

Zeichenketten in C++ – Fortsetzung

Operatoren

- Komponentenzugriff wie bei Vektoren
  `s[0]` erstes Zeichen von *s*
  `s[s.size()-1]` letztes Zeichen von *s*
  Indextyp: `string::size_type` (vorzeichenlos)
- Zuweisungsoperator
  Bsp.:
```
string s; char ct[64]; char c;
s  = ct;          // okay
ct = s;           // nicht erlaubt
s  = c;           // okay
c  = s;           // nicht erlaubt
```
- Additionsoperator (Verkettung)
  Bsp.:
```
string s,t; char cs[64],ct[64]; char c;
s+t; s+ct; ct+s; s+c; c+s; // erlaubt
cs+ct;   // nicht erlaubt
```

Zeichenketten in C++ – Fortsetzung II

Weitere Operatoren

- Operator `+=` (Anhängen)
  Bsp.:
```
s+=t; s+=ct; s+=c; // okay
cs+=t; c+=t;       // nicht erlaubt
```
- Vergleichsoperatoren (lexikalisch) `== != < > >= <=`
  Bsp.:
```
s==t    s>t    // okay
s==ct   s>ct   // okay
ct==s   cs>t   // okay
cs==ct  cs>ct  // nicht erlaubt
```

Substrings und C-Zeichenkettenzugriff

- `substr` liefert Substring als Temporärobjekt, z.B.
  `s.substr(i)` Substring von *s* ab Index *i*
- Exception (ggf. Programmabbruch), falls *i* `>s.size()`
  i `=s.size()` zulässig (leerer Substring)
- `c_str` liefert konstante C-Zeichenkette *mit* `\0` am Ende – darf *nicht* modifiziert werden
- `data` liefert konstante C-Zeichenkette *ohne* `\0` am Ende – darf ebenfalls *nicht* modifiziert werden

Zeichenketten in C++ – Fortsetzung III

Suchen, Einfügen, Löschen und Ersetzen (Komponentenfunktionen)

- `find` – sucht in String nach Substring oder einzelnem Zeichen
- Liefert Index *i* oder vorzeichenlose Zahl `string::npos` [größte vorzeichenlose Zahl `(string::size_type)(-1)`]
- Viele weitere Varianten: z.B. Suche erst ab angegebenem Index oder vom Ende her (`rfind`) – allerdings keine Suchmuster (reguläre Ausdrücke)
- Einfüge-, Lösch- oder Ersetzungsindex *i* ist bei `insert`, `erase` bzw. `replace` das *erste* Argument.
- Exception (ggf. Programmabbruch), falls *i* `>s.size()`. Zulässig: *i* `=s.size()`
- `copy` – kopiert auf C-Zeichenkette, allerdings *ohne* `\0`

Zeichenketten in C++ – Fortsetzung IV

Ein/Ausgabe

- Stringeingabe etwa mittels `cin >> s` viel bequemer und sicherer als in C wegen automatischer Größenanpassung (bei Platzproblemen Exception, kann abgefangen werden)
- `getline` Zwei unterschiedliche Ausprägungen: beide lesen von einer Zeile, schreiben auf eine Zeichenkette und entfernen `\n`
- Unterschiede:

  `getline(stream,s)` Funktion, hängt Zeile an String an und entfernt `\n` *(bequem)*

  `stream.getline(cs,n)` Komponentenfunktion, liest höchstens $n-1$ Zeichen ein und schreibt sie auf C-Zeichenkette fester Länge *(unbequem)*