Portable Zeichenverarbeitung in C++

Portables String-Management

Die Komplexität der portablen Zeichenverabeitung beginnt bereits mit dem Einlesen der Zeichen von der Tastatur. Hat man die Zeichen dann im Speicher, gilt es den passenden String-Container zu finden und die Verarbeitung zu implementieren. **Peter Thömmes, April 2005**

Zeichenketten

In der Programmierung spielte die Speicherung und Verwaltung von Zeichenketten (Strings) schon immer eine besondere Rolle, da sie mehrere Besonderheiten mit sich bringt. Eine solche ist, dass egal, wie man es auch macht, der Speicherbedarf eines Strings auf 32-Bit-Systemen irgendwo zwischen 1 Byte und 2 GB (`int` Länge) liegt oder sich gar bis 4 GB (`unsigned int` Länge) erstreckt. Der String kann sich also von seinen Eckdaten her über den gesamten virtuellen Adressbereich von 32 Bit erstrecken. Eine weitere Besonderheit ist, dass es viele Methoden gibt, die einzelnen Zeichen zu kodieren, wobei man von der 7- oder 8-Bit-ASCII-Kodierung bis zur 16-Bit-UNICODE-Kodierung eine Vielfalt von Varianten vorfindet. Last but not least hat man da die Aufnahme von Zeichen über eine Tastatur, die bereits einiges vom Entwickler abverlangt, wenn der entsprechende C++-Code portabel sein soll. Dies soll dann auch das erste Thema in dieser Abhandlung sein.

Die Zeicheneingabe

Wenn man auf die Tastatur eines PCs drückt, dann liefert der Tastatur-Controller, der die Tasten andauernd scanned, einen sogenannten **Scan-Code** (= ein Byte) an das Basic IO-System (BIOS) des PCs. Dies geschieht indem der Controller den Interrupt `0x09` auslöst und das BIOS im entsprechenden Interrupt-Handler den Scan-Code vom Tastatur-Controller einliest. Der Scan-Code zeigt in den unteren 7 Bit die **Nummer (0...127) der Taste** an, so z.B. bedeuted die 1, das die erste Taste von links oben bedient wurde, was bei MF-II-Tastaturen der `ESC`-Taste entspricht. Durch Setzen des **höchstwertigen Bits (0x80)** im Scan-Code zeigt der Tastatur-Controller an, das die Taste losgelassen wurde (**Release-Code**). Durch Löschen dieses Bits meldet er, dass die Taste gedrückt wurde (**Make-Code**). Tastaturen, die nach diesem Prinzip arbeiten, können also maximal 127 Tasten haben, tatsächlich reichen aber 102 Tasten (MF-II-Tastatur) im Normalfall aus. Das BIOS macht nun eine erste Weiterverarbeitung der Tastatureingabe unter der Annahme, dass bestimmte Scan-Codes (also Tasten) mit bestimmten Beschriftungen (wie z.B. '`A`', '`S`', '`D`', ...) versehen sind (=Tastatur-Layout) und wandelt die Scan-Codes in spezielle **BIOS-ASCII-Codes** um, die dem 7-Bit-Standard-ASCII-Code entsprechen. So wird z.B. das Drücken der Taste, die mit dem Buchstaben '`E`' beschriftet ist, in Kombination mit dem Drücken der Taste, die als `SHIFT`-Taste gekennzeichnet ist, zum Ergebnis '`E`' (also zum ASCII-Code `0x45 = 69`) verarbeitet. Ohne gleichzeitiges Drücken von `SHIFT` wird '`e`' daraus (`0x65 = 101`). Um die Kombination von Tasten mit Sondertasten erkennen zu können, merkt sich das BIOS den aktuellen Zustand der Sondertasten im Tastatur-Status-Byte:

Bit	*Gedrückte Steuertaste*
7	`INS (Einfg)`
6	`CapsLock (Groß)`
5	`NumLock (Num)`
4	`ScrollLock (Rollen)`
3	`Alt`
2	`Ctrl (Strg)`
1	`SHIFT left`
0	`SHIFT right`

Tab. 1: Tastatur-Status-Byte

Tastenkombination	*ASCII-Code*
`Extended Flag`	0
`Ctrl+C (copy)`	3
`Backspace ←`	8
`TAB`	9
`Ctrl+CR (LF)`	10
`ENTER (CR)`	13
`S`	21
`Ctrl+V(paste)`	22
`Ctrl+X (cut)`	24
`ESC`	27
`SPACE`	32
`!`	33
`"`	34
`#`	35
`$`	36
`%`	37
`&`	38
`'`	39
`(`	40
`)`	41
`*`	42
`+`	43
`,`	44
`-`	45
`.`	46
`/`	47
`0..9`	48..57
`:`	58
`;`	59
`<`	60
`=`	61
`>`	62
`?`	63
`@`	64
`A..Z`	65..90
`[`	91
`\`	92
`]`	93
`^`	94
`_`	95
`` ` ``	96
`a..z`	97..122
`|`	124
`Extended Flag`	224

Tab. 2: BIOS-ASCII-Codes

Da ca. 80 Tasten der Tastatur in Kombination mit anderen Tasten (Ctrl, Alt, SHIFT) gedrückt werden können, reicht ein einzelnes Byte, also 256 Werte, nicht für die Kodierung aus. Das BIOS macht deshalb folgendes: es führt die normale Umsetzung in **BIOS-ASCII-Codes** nur für die gebräuchlichsten Tasten/Tastenkombinationen durch. Dann definiert es, dass die Codes 0 (0x00, **NULL**) und 224 (0xE0, α) einen **Erweiterten BIOS-Code** signalisieren, d.h. wenn 0 oder 224 als BIOS-ASCII-Code zurückgeliefert wird, dann muss aus einer anderen Speicherstelle der Erweiterte BIOS-Code gelesen werden.

Tastenkombination	*Code*
F1..F10	0 59..68
SHIFT + F1..F10	0 84..93
Ctrl+SHIFT + F1..F10	0 94..103
Alt + F1..F10	0 104..113
F11..F12	224 133..134
SHIFT + F11..F12	224 135..136
Ctrl+SHIFT + F11..F12	224 137..138
Alt + F11..F12	224 139..140
INS	224 (0) 82
DEL	224 (0) 83
Home	224 71
End	224 79
PgUp	224 (0) 73
PgDn	224 (0) 81
CursUp	224 (0) 73
CursDn	224 (0) 81
CursLt	224 (0) 75
CursRt	224 (0) 77

Tab. 3: Erweiterte-BIOS-Codes

Die Cursor-Bewegungen (PgUp, PgDn, CursUp, CursDn, CursLt, CursRt) sowie INS und DEL stehen hinter 224, es sei denn sie kommen vom Nummern-Block (bei NumLock = off), dann stehen sie hinter 0.

Neben den Erweiterten BIOS-Codes dienen traditionell die normalen Codes unterhalb 32 (0x20) zur Steuerung von Datenfluss und -ausgabe (Meta-Daten). So findet man hier z.B. TAB, CR, LF und ESC.

Der Mechnismus der Übergabe der Codes (Zeichen) durch das BIOS an den nächsten großen Spieler, das **Betriebssystem** (Windows oder UNIX), sieht folgendermaßen aus: Das **BIOS** speichert den aktuellen Zustand der Tastatur intern ab (Tastatur-Puffer inclusive Start- und End-Zeiger, nächstes Zeichen, letztes Zeichen, Tastatur-Status, Erweiterter Tastatur-Status, etc.) und bietet **per Software-Interrupt 0x16 eine standardisierte Schnittstelle zur Tastatur** an. Dabei gibt es mehrere Funktionen, die aufgerufen werden können, d.h. vor dem Auslösen des Interrupts wird im AH-Register der CPU spezifiziert, welche Funktion ausgeführt werden soll. Hierzu gehört die Funktion **AH=0x00**, die das BIOS veranlasst auf eine Zeicheneingabe von der Tastatur zu warten und den BIOS-ASCII-Code im AL-Register zuückzuliefern. Falls dieser Code 0 bzw. 224 (0xE0) ist, dann wird im AH-Register der Erweiterte BIOS-Code zurückgeliefert, andernfalls ist dort der Scan-Code zu finden. Daneben gibt es die Funktion **AH=0x01**, die abfragt, ob überhaupt eine Taste gedrückt wurde und das Ergebnis im AL-Register zurückliefert. Mit der Funktion **AH=0x02** fragt man das Tastatur-Status-Byte ab, welches im AL-Register zurückgeliefert wird.

Durch die Evolution in der Tastatur-Geschichte spielen die Funktionstasten F11 und F12 eine besondere Rolle. Sie kamen erst mit den MF-II-Tastaturen Anfang der 90er ins Spiel. Zuvor nutzte man beim Ur-PC die PC/XT-Tastatur mit 83 Tasten und 10 Funktionstasten in einem Block auf der linken Seite. Die damals neueren PC/AT-Rechner hatten entweder eine MF-I-Tastatur mit 84 Tasten und etwas verbessertem Layout oder eine MF-II-Tastatur mit 101/102 Tasten und komplett überarbeitetem Layout (bei der europäischen Version mit 102 Tasten wurde die breite SHIFT-Taste aufgeteilt in eine SHIFT-Taste und eine Taste mit <, > und |). Um die zusätzlichen Tasten der MF-II-Tastaturen (F11, F12, ...) erfassen zu können, wurde der Software-Interrupt 0x16 um die Funktionen **AH=0x10** (arbeitet wie AH=0x00) und **AH=0x11** (arbeitet wie AH=0x01) erweitert, welche den Scan-Code nicht auf das Layout der MF-I-Tastatur mit nur 84 Tasten abbilden, sondern das Layout der MF-II-Tastatur mit 101/102 Tasten verwenden.

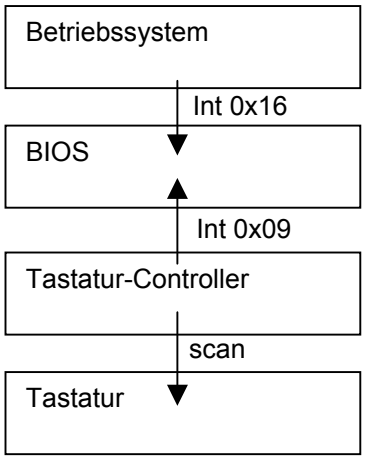

Bild 1: PC-Tastatur einlesen

Das Betriebssystem des PCs kann also die Tasatatur verwenden, indem es das BIOS per Software-Interrupt 0x16 kontaktiert. So kann es feststellen, ob eine Taste gedrückt wurde (AH=0x01), welche Taste gedrückt wurde (AH=0x00) und wie der Status der Sondertasten momentan ist (AH=0x02). Das SDK für den Programmierer einer Konsolen-Applikation unter **Windows** stellt hierzu die Funktionen **kbhit()** (wirkt wie AH=0x01) und **getch()** (wirkt wie AH=0x00) bereit. Nachdem man mit kbhit() festgestellt hat, das eine Eingabe per Tastatur stattgefunden hat, liest man mit getch() den BIOS-ASCII-Code aus. Ist der Code 0 oder 224, dann muss ein weiteres Byte mit getch() gelesen werden, welches dann den Erweiterten BIOS-Code enthält. getch() arbeitet also ganz entsprechend dem Verfahren, wie der BIOS-Interrupt 0x16 funktioniert.

Wenn eine Taste über längere Zeit gedrückt wurde, dann wird das Zeichen vom BIOS mehrfach geliefert, also auch von `getch()`. Hier nun ein einfaches Beispiel für das Lesen des Tastatur-Puffers unter Windows:

```
bool HandleKeyboard(){
 if(kbhit()){
  int i = getch();
   if((!i)||(i == 224)){
    if(!HandleEx(i))
     return false;
   }
   else{
    HandleNormal(i);
   }
 }
 return true;
}

bool HandleEx(int i){
 int j = getch();
 switch(j){
  default:
   break;
  case 73: //PgUp
   break;
  case 81: //PgDn
   break;
 }
 if(j == 68) //F10
  return false;
 return true;
}

void HandleNormal(int i){
 switch(i){
  default:
   break;
  case 8: //BS
   break;
  }
}
```

Unter **UNIX** sieht das aber nicht so einfach aus. Hier wurde zwar versucht mit der library `curses` (bzw. `ncurses` unter linux) einiges nachzubilden. `getch()` verhält sich aber nicht so, wie die Windows Funktion. Möchte man genauso komfortabel mit der Tastatureingabe umgehen, wie unter Windows, dann empfiehlt es sich die Funktionen `kbhit()` und `getch()` nachzubilden. Dabei ist folgendes zu beachten:
a) Man muss die Datei **`stdin`** lesen, wobei das Verfahren zur Kapselung des Auslesens der Tastatur eines Intel-Rechners mit BIOS-Interrupt 0x16 genauso implementiert ist, wie das Lesen der Eingabe von einer Konsole, die über einen asynchronen seriellen Port, wie RS232, angeschlossen ist.
b) `stdin` muss in **`raw`** mode gelesen werden, da es sich sonst im `canonical` mode befindet, d.h. es liefert die eingegebenen Zeichen erst nach Eingabe von `RETURN` (`CR`) ab.
c) Die automatische Bildschirmausgabe (**echo**) muss abgeschaltet werden.
d) Alle Erweiterten BIOS-Codes werden als **`ESC-Sequenzen`** geliefert. `ESC` dient also als Token und nicht `0` oder `224`, wie das beim BIOS und bei Windows der Fall ist. Beispiel:

←`[[A` = `F1`

Dies muss aber nicht so sein, denn je nach UNIX Variante kann es auch so aussehen:

←`[11~` = `F1`

oder auch so:

←`[224z` = `F1`

Die `ESC`-Sequenzen sind also nicht einheitlich für jede UNIX Distribution und können zudem auch noch unterschiedliche Länge haben (siehe **Tab. A**). Wie eine Implementierung von `kbhit()` und `getch()` unter UNIX aussehen kann, ist **Listing 1/2** zu entnehmen. Die sicherste Methode die `ESC`-Sequenzen für die eigene UNIX Umgebung herauszufinden, ist das vorherige Auslesen der Tastatur mit `kbhit()` und `getch()`, wobei der von `getch()` gelieferte Code als Hex-Code auf dem Bildschirm ausgegeben wird. Danach kann man sich einen Dekoder schreiben.

Das letzte Thema in dieser Rubrik ist das **sprach- und landesspezifische Tastatur-Layout**. Beim Installieren eines Betriebssystems wird man nach der gewünschten Sprache und dem Land, in dem man sich befindet, gefragt. Mit den Angaben stellt das System die sogenannte **Locale** ein. Nun ist es aber so, dass man für verschiedene Sprach/Land-Kombinationen auch verschiedene Tastatur-Layouts (**Input-Locales**) entwickelt hat, d.h. es steht ggf. etwas anderes auf der Taste, als das BIOS annimmt. So findet man bspw. auf einer deutschen Tastatur unter SHIFT+'Ö' den Doppelpunkt '`:`', da das BIOS eine amerikanische Tastatur annimmt.

Amerik.Tastatur	*Dt. Tastatur*
\	#
:	SHIFT + Ö
@	"
"	SHIFT + Ä
.	.
/	-
-	ß
_	?
|	'
?	_
=	/

Tab. 5: Amerik. und dt. Tastatur

Man sieht also folgendes: Das **Betriebssystem** muss auf das sprach- und landesspezifische Tastatur-Layout eingestellt werden, damit es durch einen **Lookup der BIOS-ASCII-Codes**, welche eine amerikanische Tastatur voraussetzen, eine Abbildung auf das eigentliche Layout vornehmen kann. Die Abbildung auf andere Layouts als das der amerikanischen Tastatur erfordert aber auch die Umwandlung in Codes oberhalb 127, da Zeichen wie ö, ä, ü, ß usw. nicht zum 7-Bit-Standard-ASCII-Code gehören. Daher ist die Angabe der Code-Seite (Codepage) hier unumgänglich. Die Codepage spezifiziert die Kodierung oberhalb 127.

Unter **Windows** werden Locales nach folgendem Schema definiert (`CP` = CodePage):

```
<Sprache>_<Land>.<CP>
```

Beispiele:

```
English_USA.1252
English_UK.1252
German_Germany.1252
```

Die Kombination `English_USA` wird durch **`1033`** kodiert, was man manchmal auch als Verzeichnisname sieht. Für die 8-Bit Kodierung benutzt Windows die **ANSI-Codeseiten** (Standard = **`1252`**). Neben dem 8-Bit-Verfahren unterstützt Windows aber auch die 16-Bit-**UNICODE**-Kodierung, wobei statt der globalen Definition einer Codepage jedes Zeichen mittels

eindeutigem 16-Bit-Code abgespeichert wird. Die unteren Codes von `0x0000` bis `0x007F` (**UNICODE-Seite `C0`**) entsprechen dabei dem 7-Bit-Standard-ASCII-Code. Die Codes von `0x0080` bis `0x00FF` (**UNICODE-Seite `C1`**) sind äquivalent zu denen der der ANSI Code-Seite 1252.

Unter UNIX sieht das Ganze etwas anders aus, z.B. findet man hier Definitionen nach dem Schema:

```
en_US
en_UK
de_DE@euro
```

oder auch

```
en_US.iso88591
en_UK.iso88591
de_DE.iso885915
```

Normalerweise wird dort der **ISO/IEC8859**-Standard verwendet, wobei die Codes `0x00`...`0xFF` von **ISO/IEC8859-1** eins-zu-eins auf die UNICODE-Codes `0x0000` bis `0x00FF` (**UNICODE Seite `C0` und `C1`**) abgebildet sind, was zugleich der **ANSI-Code-Seite `1252`** entspricht.

Solange man also bei der Sprache Englisch, dem Land USA und der amerikanischen Tastatur bleibt, kann nichts schief gehen :-).

GUI und Tastatur

Um eine portable Tastatur-Abfrage für Applikationen mit Graphischem User Interface (GUI) zu implementieren, muss man auf der Windows-Seite das GDI (Graphics Device Interface) und auf der UNIX-Seite einen X-Server kontaktieren. Das Windows-GDI ist sehr schnell, da es ein Sub-System des Betriebssystems ist. Der X-Server unter UNIX hingegen ist zwar portabel (es gibt auch X-Server für Windows, wie z.B. `Exceed` von `Hummingbird`) aber langsam. Deshalb würde ich persönlich nicht einen X-Server unter meine Anwendung tun, um darauf meine X11-Sourcen wiederzuverwenden, sondern würde intern, in den `.cpp`-Implementierung-Dateien, zwei unterschiedliche Implementierungen für die systemnahen Calls pflegen.

Unter dem Windows-GDI werden Tastatur-Ereignisse in die normale Nachrichten-Schlange der Anwendung abgelegt und wie folgt abgefragt:

```
MSG msg;
while(GetMessage(
          &msg,NULL,0,0))
{
 switch(msg.message)
 {
  case WM_KEYDOWN:
  case WM_KEYUP:
  {
   unsigned short wKey =
        (unsigned short)
              msg.wParam;
   ...
  }
 }
}
```

Der X-Server arbeit ganz analog und man könnte ein Abfrage wie folgt gestalten:

```
XEvent Event;
while(XNextEvent(
            pDisplay,
            &Event)){
 switch(Event.type)
 {
  case KeyPress:
  case KeyRelease:
  {
   char szBuffer[11];
   KeySym keysym;
   XComposeStatus status;
   int nCnt
      = XLookupString(
         &Event.xkey,
         szBuffer,
         10,
         &keysym,
         &status);
   szBuffer[nCnt] = 0x00;
   Character c(
       keysym,
       szBuffer,
       Event.xkey.state);
   ...
   break;
  }
 }
}
```

Innerhalb der Klasse `Character` erfolgt dann die weitere Auswertung, z.B. kann der erste Parameter des Konstruktors (`keysym`) auf Gleichheit mit `XK_BackSpace` oder `XK_Delete` überprüft werden. Details hierzu entnimmt man einer X11-Dokumentation oder man wirft einen Blick in die Datei `X11/keysymdef.h`.

UNICODE bei Strings

Wenn man innerhalb eines Strings, der 8-Bit-Codes enthält, Zeichen aus verschiedenen Code-Seiten einfügen will (bspw. Sonderzeichen etc.), dann muss man Meta-Daten in den String einbauen, um an der jeweiligen Stelle die Code-Seite umzuschalten. Derjenige, der den String darstellen will muss diese zusätzlichen Meta-Daten dekodieren können (Text-Prozessor). Verwendet man generell eine 16-Bit-UNICODE-Kodierung, dann vereinfacht man sich das Leben in diesem Fall erheblich, falls derjenige, der die Daten ausgeben soll, die UNICODE-Dekodierung implementiert hat. Windows kann dies und stellt im Win32-API für jede String-Funktion neben der 8-Bit-ANSI-Variante (**`A`**) eine 16-Bit-UNICODE-Variante (**`W`**) bereit, z.B.

```
TextOutA()
TextOutW()
```

String-Container in C++

Zur Speicherung bzw. Bearbeitung von Zeichenketten etablierten sich in der Welt von C++ folgende String-Container:

- **`char-array`**

Die klassische Art Strings zu speichern geschieht unter Verwendung eines Arrays von Bytes, die den 7- oder 8-Bit-ASCII-Code der Zeichen aufnehmen und die Zeichenkette mit `0x00` abschliessen:

```
char szHello[128] = "";
strcpy(szHello,"Hello");
```

Obwohl die ASCII-Codes nur als positive Werte 0...255 bekannt sind, wird hier in der Regel `char`, also `signed char` und nicht `unsigned char` als Datentyp benutzt. Um den richtigen Code-Wert sichtbar zu machen, muss man deshalb nach (`unsigned char`) casten.

▪ **unsigned short-array**
UNICODE benötigt 2 Byte zur Speicherung eines Zeichens. Zur Aufnahme des 16-Bit-Codes der Zeichen kreierte man den sogenannten wide-character-string, ein array von `unsigned short` Werten, also 16-Bit-Codes:

```
unsigned short
     wszWorld[128] = L"";
wcscpy(
   wszWorld,
   L"World");
```

▪ **STL - std::string**
Mit `string` stellt die STL (Standard Template Library) im `namespace std` ein leistungsstarkes String-Objekt zur Verfügung. `string` kapselt ein `char`-array (7- oder 8-Bit-ASCII-Code) und ist portabel, da die STL portabel ist.

```
string strHello;
strHello = "Hello";
```

▪ **STL - std::wstring**
`wstring` ist der 16-Bit-Code-Text-Container STL, kapselt also ein `unsigned short`-array.

```
wstring wstrWorld;
wstrWorld = L"World";
```

▪ **CORBA string (char-array)**
Ein wenig verwunderlich ist es schon, dass die CORBA-Gemeinde ihren String genauso tauft, wie die STL, da sich keineswegs ein neues Objekt oder gar ein STL-`string` dahinter verbirgt. Vielmehr bildet CORBA `string` einfach auf **`char*`** ab. Dies ist nicht nur unschön, sondern hat auch andere Konsequenzen, denn man kann die Einbindung von `namespace std` der STL, wegen der Namensgleichheit vergessen:

```
using namesoace std;
```

▪ **MFC - CString**
Wie so viele Bibliotheken, hat auch die MFC, als Klassen-Bibliothek des Visual-C++ 6.0 Compilers ihren eigenen String definiert. Leider geschah dies nicht durch Kapselung bzw. Erweiterung von `string` bzw. `wstring` der STL, so dass Code, welchen CString einsetzt nicht portabel ist.

```
CString cstrMFC;
cstrMFC = "MFC";
```

Durch Verwendung der `_T()`-MACROS kann man den Code wahlweise für den 8-Bit- oder den 16-Bit-Code bauen:

```
cstrMFC = _T("MFC");
```

Die MACROS setzen dann, ne nach Compiler-Schalter `_UNICODE`, den jeweils richtigen Term ein, wie z.B. für die Initialisierung:

8-Bit-Code (ANSI):
`_T("MFC")` → `"MFC"`

16-Bit-Code (UNICODE):
`_T("MFC")` → `L"MFC"`

`CString` wird oft auch genutzt, wenn sonst im Code gar keine MFC - Klassen eingestetzt werden, weil das Problem der Konvertierung von und zu `BSTR` (siehe unten) zu aufwendig erscheint. Das Problem habe ich jedoch für den Leser gelöst (siehe weiter unten).

▪ **DCOM - BSTR**
Der Gedanke bei der Erfindung von `BSTR` war sicherlich nicht falsch: man wollte einen String schaffen, der programmiersprachenunabhängig genutzt werden kann. Da manche Sprachen (z.B. Pascal) einen String definieren, indem sie seine Zeichen in ein array speichern, dessen erstes Element (also das mit dem Index 0) die Länge angibt, andere Sprachen, wie C und C++, jedoch keine Längenangabe benutzen und stattdessen `0x00` als String-Ende an den String anhängen, wollte man beide Ideen bedienen. Zugleich sollte der 16-Bit-Code genutzt werden können. So bildete man den `BSTR` aus 16-Bit-Speicherzellen, packte an den Anfang des `BSTR` eine Längenangabe und an das Ende `0x0000`. Da DCOM (Distributed COM) im Prinzip das auf TCP/IP aufsetzende DCE-RPC kapselt (= ORPC) muss die Allokierung des Speichers durch das Netzwerkbetriebssystem geschehen. Im Falle von DCOM heisst dieses Netzwerkbetriebssystem Windows (bzw. COM+ Runtime Environment). Zur `BSTR`-Speicherverwaltung stellt Windows hauptsächlich 2 Funktionen bereit:

```
SysAllocString()
SysFreeString()
```

Beispiel:

```
BSTR bstrMsg =
   SysAllocString(L"Hi");
...
SysFreeString(bstrMsg);
```

Umwandeln von Strings

▪ **char*/string**
Die elementarste Umwandlung ist natürlich die zwischen `char`-array und `string`.
Vom `char`-array zu `string`:

```
char szHello[] = "Hello";
char szWorld[] = "World";
string strText(szHello);
strText += " ";
strText += szWorld;
```

Für den umgekehrten Weg gibt es die Methode `c_str()` in Kombination mit den konventionellen Zeichenverarbeitungsmethoden, die man von C her kennt:

```
string strHello("Hello");
char szText[1024] = "";
strcpy(
     szText,
     strHello.c_str());
```

oder direkt die `copy()` Methode:

```
strHello.copy(
     szText,
     0
     strHello.length());
szText[strHello.length()]
                = 0x00;
```

wobei die abschliessende Null noch manuell angehängt werden muss.

▪ **unsigned short*/wstring**
Natürlich geschehen diese Umwandlungen analog zu denen in der 8-Bit-String-Welt. Hierbei ist zu beachten, dass ein 'L' vor die String-Konstanten geschrieben werden muss, damit der Compiler erkennt, dass es sich um ein `unsigned short`-array handelt:

```
char wszHello[]
          = L"Hello";
char wszWorld[]
          = L"World";
wstring
     wstrText(wszHello);
wstrText += L" ";
wstrText += wszWorld;
```

und umgekehrt:

```
wstring
     wstrHello(L"Hello");
char wszText[1024] = L"";
wcscpy(
     wszText,
     wstrHello.c_str());
```

▪ **BSTR/CString**

Da `CString` von den Machern von `BSTR` ist, ist natürlich klar, dass es dort vordefinierte Wege der Umwandlung gibt. Deshalb wird oft `CString` selbst dann eingesetzt, wenn sonst gar keine MFC benutzt wird. Wie man dennoch auf `CString` verzichten kann wird weiter unten gezeigt. Hier zunächst der Microsoft-Weg:

```
CString cstrHi(_T("Hi"));
BSTR bstrHi =
cstrHi.AllocSysString();
...
SysFreeString(bstrHi);
```

Um sich den allokierten `BSTR` anschauen zu können, muss neben der MFC noch ein wenig die Active Template Library ATL beansprucht werden:

```
#include <afxwin.h>//MFC
#include <ATLBase.h>//ATL
CString SpyBSTR(
          BSTR bstrStr){
 CComBSTR combstrTemp;
 combstrTemp.Attach(
             bstrStr);
 CString strString(
          combstrTemp);
 combstrTemp.Detach();
 return strString;
}
```

Die Konvertierungen über `CString` kann man für reine Win32-Konsole-Anwendungen nicht verwenden ohne die MFC und die ATL einzubinden. Ein Weg, der hingegen immer funktioniert, ist die unten gezeigte Methode der Umwandlung in `string` bzw. `wstring`.

▪ **BSTR/char***

Die Zusammenarbeit von `char`-arrays und `BSTR` findet man in der Regel bei DCOM (siehe oben), denn dort ist man gezwungen `BSTR` als String-Container für den Netwzerk-Transport zu benutzten. Obwohl man das Verpacken in den `BSTR` mit `SysAllocStringByteLen()` durchführen kann, wobei ein echtes `char`-array in das `unsigned short`-array des `BSTR` gepackt wird, ist diese Vorgehensweise nicht ratsam, da die Gegenseite zunächst nicht sieht, dass es sich um ein gepacktes `char`-array handelt. Am sichersten ist es, wenn man alles, was man in einen BSTR packt zunächst in ein `unsigned short`-array umwandelt, wie das Beispiel zeigt:

```
BSTR AllocBSTR(
      const char* szStr){
 unsigned short* wszStr
                  = NULL;
 unsigned long dwSize
   = (strlen(szStr)+1)*2;
 wszStr =
    (unsigned short*)
       CoTaskMemAlloc(
                 dwSize);
 mbstowcs(
    wszStr,szStr,dwSize);
 BSTR bstrStr =
  SysAllocString(wszStr);
 CoTaskMemFree(wszStr);
 return bstrStr;
}
```

***Bemerkung**:*

Im Watch-Fenster von Visual C++ 6.0 kann man einen `BSTR` mit gepacktem `char`-array mittels 's' sichtbar machen:

```
   bstrStr,s
```

und mit normalem `unsigned short`-array mittles 'su'

```
   bstrStr,su
```

Hat das Netzwerk nun den `BSTR` zu einem anderen Rechner übertragen (DCOM), dann kann die Anwendung dort sich die Daten wie folgt anschauen:

```
bool SpyBSTR(
   BSTR bstrStr,
   char* szStr,
   unsigned short dwLen){
 unsigned long dwNumChars
     = wcslen(bstrStr);
 if(dwLen <
        (dwNumChars + 1))
     return false;
 wcstombs(
        szStr,
        bstrStr,
        dwNumChars);
 szStr[dwNumChars]
               = 0x00;
 return true;
}
```

Man darf auf keinen Fall vergessen den System-Speicher des `BSTR` wieder freizugeben, wenn der Funktionsaufruf ans Netzwerk getan ist:

```
 SysFreeString(bstrStr);
```

Ein Call an das DCOM-Netzwerk würde dann etwa so aussehen:

```
BSTR bstrStr =
      AllocBSTR("Test");
MyCall(bstrStr);
SysFreeString(bstrStr);
```

Die andere Seite implementiert dann im Stub-Objekt:

```
void MyCall(){
 char szSpy[1024] = "";
 SpyBSTR(
       bstrStr,
       szSpy,
       sizeof(szSpy));
 ...
}
```

DCOM auf diese Art zu nutzen kommt nun sehr nah an CORBA heran. Sollte man also planen Teile des Netzwerk-Software-Codes in einer CORBA-Implementierung wieder zu verwenden, dann ist dies sicherlich der richtige Weg.

BSTR/unsigned short*

Die 16-Bit-Variante arbeitet analog. Da das `unsigned short`-array jedoch näher an `BSTR` ist als das `char`-array, vereinfachen sich die Konvertierungen lediglich etwas:

```
BSTR AllocBSTR(const
 unsigned short* wszStr){
 return
  SysAllocString(wszStr);
}


bool SpyBSTR(
  BSTR bstrStr,
  unsigned short* wszStr,
  unsigned short dwLen){
 unsigned long dwNumChars
       = wcslen(bstrStr);
 if(dwLen<(dwNumChars+1))
     return false;
 wcscpy(wszStr,bstrStr);
 wszStr[dwNumChars]
                = 0x00;
 return true;
}
```

- **BSTR / string**

Wenn man Strings in die DCOM-Netwerk-Umgebung schickt, dann gibt es aus meiner Sicht nur 2 Strings, die man verwenden sollte: `string` und `BSTR`. Einzige Ausnahme sind UNICODE-Implementierungen - diese sollten dann die Paarung `wstring` und `BSTR` verwenden. Begründung: Die Verwendung von `BSTR` ist zwingend bei DCOM und die von `string` (wie auch `wstring`) bietet ein Höchtmaß an Kompatibilität und Effizienz.

Hier also nun der ultimative Weg von `string` hin zu `BSTR`:

```
BSTR AllocBSTR(
   const string& strStr){
 unsigned short* wszStr
                 = NULL;
 unsigned long dwSize
 = (strStr.length()+1)*2;
 wszStr=(unsigned short*)
  CoTaskMemAlloc(dwSize);
 mbstowcs(
       wszStr,
       strStr.c_str(),
       dwSize);
 BSTR bstrStr =
  SysAllocString(wszStr);
 CoTaskMemFree(wszStr);
 return bstrStr;
}
```

Und für den Empfänger auf der anderen Seite des Netwerkes:

```
string SpyBSTR(
           BSTR bstrStr){
 unsigned long
    dwNumChars
       =wcslen(bstrStr);
 char* szStr = new
      char[dwNumChars+1];


 wcstombs(
       szStr,
       bstrStr,
       dwNumChars);
 szStr[dwNumChars]
                = 0x00;
 string strStr(szStr);
 delete[] szStr;
 return strStr;
}
```

Auch wenn die Gefahr besteht, dass ich mich wiederhole: auf keinen Fall darf der Aufruf von

```
SysFreeString(bstrStr);
```

fehlen! Speicherlöcher sind eines der Hauptübel in komplexer Software.

- **BSTR / wstring**

Natürlich ist die 16-Bit-Version in ihrem Verhalten ähnlich. Da jedoch `BSTR` und `wstring` hinsichtlich der Kodierungsbreite (16 Bit) Artgenossen sind, verienfachen sich die Dinge hier sehr:

```
BSTR AllocBSTR(
 const wstring& wstrStr){
 return SysAllocString(
        wstrStr.c_str());
}

wstring SpyBSTRW(
          BSTR bstrStr){
 unsigned long dwL
       = wcslen(bstrStr);


 unsigned short*
  wszStr = new
   unsigned short[dwL+1];
 wcscpy(wszStr,bstrStr);
 wszStr[dwL] = 0x0000;
 wstring wstrStr(wszStr);
 delete[] wszStr;
 return wstrStr;
}
```

Portable String-Algorihmen

Natürlich ist klar, dass man Bücher mit String-Algorithmen füllen kann und deshalb kann das Thema an dieser Stelle nur angeschnitten werden. Wichtig zu wissen ist nur folgendes: Sollen die Algorithmen portabel sein, dann gibt es genau 2 Möglichkeiten:

a) Man benutzt `string` oder `wstring` aus der Standard Template Library (STL) und passt sie den eigenen Zwecken an.

b) Man schreibt eine eigene String-Klasse und erfindet alles, was die STL bereits bietet nochmal. Für den nur mittelmäßig erfahrenen Programmierer ist dies nicht zu empfehlen.

Hier eine kleine Kostprobe zur Variante a:

- **Großbuchstaben**

Unter Berücksichtigung der sprach- und landesspezifischen Einstellung des Systems kann man wie folgt eine Umwandlung in Großbuchstaben vornehmen:

```
string ToUpper(
 const string& strIN,
 const char* const
           szLoc = "C"){
 string strRET(strIN);
 locale loc(szLocale);
 const ctype<char>& ct =
 use_facet<ctype<char> >(
                     loc);
 ct.toupper(
 const_cast<char*>(
    strRET.c_str()),
    strRET.c_str()
      + strRET.length());
 return strRET;
}
```

Die Nutzung würde dann etwa so ausehen:

```
string strName("Peter");
string strNAME
     = ToUpper(strName);
```

- **Zeilen eines Textes**

Um die Zeilen eines Textes zu ermitteln, kann man bspw. folgende Funktion benutzen:

```
void GetLines(
 const string& strText,
 list<string>& listLines)
{
 listLines.clear();
 int iPos = 0;
 int iStartPos = iPos;
 do
 {
  iPos =
     strText.find(
               "\n",
               iStartPos);
  if(iPos!=string::npos){
   unsigned long dwLen =
        iPos - iStartPos;
   char* szLine = new
         char[dwLen + 1];
   strText.copy(
           szLine,
           dwLen,
           iStartPos);
   szLine[dwLen] = 0x00;
   listLines.push_back(
               szLine);
   delete[] szLine;
   ++iPos;
   iStartPos = iPos;
  }
  else if(iStartPos){
   unsigned long dwLen =
     strText.length() -
                iStartPos;
   char* szLine = new
         char[dwLen + 1];
   strText.copy(
            szLine,
            dwLen,
            iStartPos);
   szLine[dwLen] = 0x00;
   listLines.push_back(
                szLine);
   delete[] szLine;
  }
  else{
   listLines.push_back(
               strText);
  }
 }
 while(
     iPos!=string::npos);
}
```

Die Nutzung der Funktion könnte dann so ausehen:

```
list<string> listLines;
GetLines(
      strText,
      listLines);
list<string>::iterator it
     = listLines.begin();
for(;
   it != listLines.end();
    ++it){
 printf("%s\r\n",*it);
}
```

Das sind nur 2 kleine Beispiele. Fest steht jedoch: Die Zeit, die man in die Programmierung und das Testen von String-Algorithmen investiert, ist meist enorm und manche Fehler findet man erst sehr spät, wenn es z.B. um komplexe Dinge wie effektive Speicherverwaltung und copy-on-write geht. Am effektivsten ist die Verwendung von `string` und `wstring` als Basis der eigenen String-Programmierung in C++.

Fazit

Um das String-Management einer Software **portabel** und damit wiederverwendbar zu machen, geht man folgenden Weg:

1.) Eingabe
Für reine Konsole-Anwendungen implementiert man das Einlesen der Tastatur in einer Klasse, die sowohl `kbhit()` als auch `getch()` platformübergreifend anbietet. Dabei arbeitet man effizient, wenn man die ESC-Sequenzen von UNIX in die BIOS-Codes von Windows (ganz im Sinne von `getch()` unter Windows) abbildet. Für Anwendungen mit Graphischem User Interface (GUI) kapselt man in gemeinsamen Klassen/Methoden die Nachrichtenbehandlung des Windows-GDI und die Event-Behandlung des X-Servers, wobei man intern, wenn nötig, mittels `#ifdef _WIN32` den Compiler zu dem ein oder anderen Code lenkt.

2.) Verarbeitung/Netzwerk
Man arbeit grundsätzlich nur mit `string` und `wstring` der STL, da diese nicht nur sehr schnell arbeiten, sondern in jedem C++-Compiler auf jeder Platform verfügbar sind, da die STL seit 1994 Teil des C++ Standards ist. Die Verwendung in Zusammenarbeit mit der DCOM-Netzwerk-Umgebung habe ich oben gezeigt. Für verteilte Anwendungen jedoch ist DCOM nicht die Grundlage für Portabilität, den DCOM gibt es normalerweise nur unter Windows (zugegeben - es gibt auch Nachbauten unter linux). Für CORBA war das theoretisch anders gedacht, jedoch erweist sich die Portierung oft als mühselig und kompliziert, wobei man sich von dem Hersteller des ORB (= Netzwerk-Laufzeitumgebung bzw. Middleware - analog zu DCOM) abhängig macht, was früher oder später schlimme Konsequenzen haben kann. Also ist die einzig sichere portable Implementierung das Arbeiten mit reinen Socket-Verbindungen auf der Basis von TCP/IP. Dann schiebt man nur reine Byte-Ströme hin und her und gibt den einzelnen Bytes eine Bedeutung. Man kann mit der STL auch sehr schnell einen einfachen Umsetzer von und zu Bytetströmen schreiben, so dass man generische Helper-Funktionen aufruft um auf das Netz zuzugreifen.

3.) Ausgabe
Die portable Bildschirmausgabe wurde hier nicht behandelt. Soll die Ausgabe jedoch auf einen Text-Bilschirm mit 80x25 Zeichen gehen (Konsole-Anwendung), dann gibt es durchaus die Möglichkeit mit einer einzigen, nicht zu großen Klasse die Win32- und die UNIX-Funktionalität der Konsole relativ schnell unter einen Hut zu bringen. Bei einer graphischen Ausgabe (GUI) hingegen muss man schon wesentlich mehr investieren, da man auf der einen Seite das GDI von Windows und auf der anderen einen X-Server unter UNIX hat.

```
bool Keyboard::set_c_lflag__(
     int iFileDescriptor,int iFlag,int iState,int& iOldState)
{
   iOldState = 0;

   //Get the old state of the flag:
   termios tio;
   if(tcgetattr(iFileDescriptor,&tio))
     return false;
   iOldState = (tio.c_lflag & iFlag);

   //Set new attributes:
   //(TCSADRAIN -> apply changes after all output was transmitted)
   if(iState)
     tio.c_lflag |= iFlag;
   else
     tio.c_lflag &= ~iFlag;
   if(tcsetattr(iFileDescriptor,TCSADRAIN,&tio))
     return false;
   return true;
}

bool Keyboard::kbhit()
{
   //Switch off canonical mode (canonical mode delivers line by line
   //only and decodes special characters like EOF (0x1A)...):
   bool bNoKeyboard = false;
   int iOldStateICANON = 0;
   if(!set_c_lflag__(STDIN_FILENO,ICANON,0,iOldStateICANON))
   {
     if((errno == EINVAL) || //STDIN_FILENO is not a controlling terminal
        (errno == ENOTTY))   //there is no controlling terminal connected
     {
       bNoKeyboard = true;
     }
     else
     {
       return false;
     }
   }

   //Array of file-descriptors (sockets) for readability-check:
   fd_set fdsetRead;
   FD_ZERO(&fdsetRead);
   FD_SET(STDIN_FILENO,&fdsetRead); //STDIN_FILENO = fileno(stdin)

   //Timeout to select socket:
   timeval timevalNONE;
   timevalNONE.tv_usec = 0;
   timevalNONE.tv_sec = 0;

   //Waiting for the socket to get ready:
   int iNumSockets = select(
               STDIN_FILENO + 1,&fdsetRead,NULL,NULL,&timevalNONE);

   //Switch back the ICANON flag:
   set_c_lflag__(STDIN_FILENO,ICANON,iOldStateICANON);

   if(iNumSockets <= 0) //if not at least one socket ready
     return false;
   return true;
}
```

Listing 1: kbhit() für UNIX

```
long Keyboard::getch()
{
  //Switch off canonical mode (canonical mode delivers line by line
  //only and decodes special characters like EOF (0x1A)...):
  int iOldStateICANON = 0;
  if(!set_c_lflag__(STDIN_FILENO,ICANON,0,iOldStateICANON))
    return -1;

  //Switch off echo of input characters:
  int iOldStateECHO = 0;
  if(!set_c_lflag__(STDIN_FILENO,ECHO,0,iOldStateECHO))
  {
    set_c_lflag__(STDIN_FILENO,ICANON,iOldStateICANON);
    return -1;
  }

  //Get the next input character:
  long lChar = -1;
  char cBuf[1] = "";
  for(;;)
  {
    int iNumRead = read(STDIN_FILENO,cBuf,1);
    if(iNumRead == 1) //1 = ok
    {
      lChar = (long) cBuf[0]; //ok
      break;
    }
    if(!iNumRead) //0 = no character there
      break;
    if(errno == EINTR) //if call was interrupted by a signal
      continue; //try again
    break; //error
  }

  //Switch back the ICANON flag:
  set_c_lflag__(STDIN_FILENO,ICANON,iOldStateICANON);

  //Switch back the ECHO flag:
  set_c_lflag__(STDIN_FILENO,ECHO,iOldStateECHO);

  return lChar;
}
```

Listing 2: getch() für UNIX

<127>	**DEL**

<ESC> [1 ~	HOME
<ESC> [2 ~	INS
<ESC> [3 ~	DEL
<ESC> [4 ~	END
<ESC> [5 ~	PG_UP
<ESC> [6 ~	PG_DOWN
<ESC> [1 1 ~	F1
<ESC> [1 2 ~	F2
<ESC> [1 3 ~	F3
<ESC> [1 4 ~	F4
<ESC> [1 5 ~	F5
<ESC> [1 7 ~	F6
<ESC> [1 8 ~	F7
<ESC> [1 9 ~	F8
<ESC> [2 0 ~	F9
<ESC> [2 1 ~	F10
<ESC> [2 3 ~	F11
<ESC> [2 4 ~	F12
<ESC> [3 7 ~	SHIFT + F1
<ESC> [3 8 ~	SHIFT + F2
<ESC> [3 9 ~	SHIFT + F3
<ESC> [4 0 ~	SHIFT + F4
<ESC> [4 1 ~	SHIFT + F5
<ESC> [4 2 ~	SHIFT + F6
<ESC> [4 3 ~	SHIFT + F7
<ESC> [4 4 ~	SHIFT + F8
<ESC> [4 5 ~	SHIFT + F9
<ESC> [4 6 ~	SHIFT + F10
<ESC> [4 7 ~	SHIFT + F11
<ESC> [4 8 ~	SHIFT + F12
<ESC> [4 9 ~	CTRL + F1
<ESC> [5 0 ~	CTRL + F2
<ESC> [5 1 ~	CTRL + F3
<ESC> [5 2 ~	CTRL + F4
<ESC> [5 3 ~	CTRL + F5
<ESC> [5 4 ~	CTRL + F6
<ESC> [5 5 ~	CTRL + F7
<ESC> [5 6 ~	CTRL + F8
<ESC> [5 7 ~	CTRL + F9
<ESC> [5 8 ~	CTRL + F10
<ESC> [5 9 ~	CTRL + F11
<ESC> [6 0 ~	CTRL + F12

Tab. A: ESC Sequenzen unter linux