

Beispiel: Funktion für Matrixvektormultiplikation (STL-Vektoren)

```
#include <iostream>
#include <iomanip>
#include <vector>

using namespace std;

vector<double> matvekmult(vector<vector<double>> a, vector<double> x)
// Liefert A*x als Funktionswert
{
    int m=a.size(),n=x.size(); // keine Ueberpruefung auf Rechtecksgestalt
    vector<double> b(m);

    for (int i=0; i<m; ++i) {
        b[i] = 0;
        for (int j=0; j<n; ++j) b[i] += a[i][j]*x[j];
    }
    return b;
}

int main()
{
    int m,n;
    cout << "m n: "; cin >> m >> n;

    vector<double> x(n),b;

    // Matixvereinbarung
    vector<vector<double>> a(m,vector<double>(n));

    // Matriceingabe (a)
    cout << endl << "a:" << endl;
    for (int i=0; i<m; ++i)
        for (int j=0; j<n; ++j)
            cin >> a[i][j];

    // Vektoreingabe (x)
    cout << endl << "x:" << endl;
    for (int j=0; j<n; ++j) cin >> x[j];

    // Matrixvektormultiplikation (b=a*x)
    b = matvekmult(a,x);

    // Vektorausgabe (b)
    cout << endl << "b:" << endl;
    for (int i=0; i<m; ++i)
        cout << fixed << setprecision(2) << setw(6) << b[i] << endl;
    return 0;
}
```

Zeichen

Datentypen für Zeichen

`char`, `unsigned char`, `signed char` (Größe 1 Byte=`CHAR_BIT` Bits, definiert in `climits`)

Sie zählen (wie `bool`) zu den ganzzahligen Datentypen und unterliegen in arithmetischen Ausdrücken der Integererweiterung. Die Regeln für implizite Typumwandlungen gelten analog.

Es ist implementierungsabhängig, ob `char` dieselbe Darstellung wie `signed char` oder `unsigned char` besitzt. (g++-9.4, Ubuntu Linux 20.04: `char`≐`signed char`).

Bsp. für Zeichenlitterale: `'A'`, `'5'`, `'*'`, `'\n'`, `' '`, `'\0'`, `'\40'`, `'\x20'`, `'\''`, `'\\'`.

Der arithmetische Wert eines Zeichens ist positiv für die druckbaren Zeichen des Minimalzeichensatzes.

Ersatzdarstellung für einzelne Zeichen

Zeichen *Bedeutung*

<code>\a</code>	akustisches Signal
<code>\b</code>	Rückwärtspositionierung um 1 Zeichen
<code>\f</code>	Seitenvorschub
<code>\n</code>	Zeilenvorschub
<code>\r</code>	Wagenrücklauf
<code>\t</code>	horizontaler Tabulator
<code>\v</code>	vertikaler Tabulator
<code>\\</code>	rückwärts geneigter Schrägstrich
<code>\?</code>	Fragezeichen
<code>\'</code>	Apostrophzeichen
<code>\"</code>	Doppelapostrophzeichen
<code>\ooo</code>	Oktaldarstellung (ein bis drei Oktalziffern)
<code>\xhh</code>	Hexadezimaldarstellung (ein oder zwei Hexadezimalziffern)

Testen und Umwandeln von Zeichen (`cctype`)

Funktion *Bedeutung*

<code>int islower(int c)</code>	Kleinbuchstabe
<code>int isupper(int c)</code>	Großbuchstabe
<code>int isalpha(int c)</code>	Buchstabe
<code>int isdigit(int c)</code>	Dezimalziffer
<code>int isxdigit(int c)</code>	Hexadezimalziffer
<code>int isalnum(int c)</code>	Buchstabe oder Ziffer
<code>int isgraph(int c)</code>	druckbar ohne Leerzeichen
<code>int isprint(int c)</code>	druckbar einschließlich Leerzeichen
<code>int ispunct(int c)</code>	druckbar mit Ausnahme von Leerzeichen, Buchstaben und Ziffern
<code>int isspace(int c)</code>	Zwischenraumzeichen
<code>int iscntrl(int c)</code>	Steuerzeichen
<code>int tolower(int c)</code>	Umwandlung in Kleinbuchstaben
<code>int toupper(int c)</code>	Umwandlung in Großbuchstaben

c muss eine ganze Zahl vom Datentyp `int` aus dem Wertebereich von `unsigned char` und EOF sein. Ebenso liefern `toupper` und `tolower` ganze Zahlen aus diesem Wertebereich.

C-Zeichenketten

C-Zeichenketten sind C-Vektoren aus `char`, die mit `'\0'` enden.

C-Zeichenkettenkonstante: $"z_0 \dots z_{k-1}" \hat{=} \{'z_0', \dots, 'z_{k-1}', '\0'\}$

Bsp.: Initialisierung von C-Zeichenkettenvariablen

```
char name[5] = "text"; /* {'t','e','x','t','\0'} */
char name[4] = "text"; /* {'t','e','x','t'} keine C-Zeichenkette! */
char name[] = "text"; /* {'t','e','x','t','\0'} */
char name[6] = "text"; /* {'t','e','x','t','\0','\0'} */
char name[] = ""; /* {'\0'} */
```

Funktion (<cstring>)

Bedeutung

`size_t strlen(const char s[])` Länge der Zeichenkette *s* (ohne `'\0'`)

STL-Zeichenketten in C++ (string)

Im Unterschied zu C-Zeichenketten werden C++-Zeichenketten nicht mit `\0` abgeschlossen, sondern es wird die aktuelle Länge mit abgespeichert.

Im folgenden sind einige wichtige Operatoren und Funktionen für den Datentyp `string` aufgeführt. Dabei stehen *s* und *t* für C++-Zeichenketten, *ct* für eine `\0`-terminierte C-Zeichenkette, *c* für ein Zeichen vom Typ `char` und *i* und *n* für ganze Zahlen vom Typ `string::size_type` (g++-9.4 unter Ubuntu Linux 20.04 amd64: `unsigned long`).

Operation

Wirkung

<code>string s</code>	vereinbart leere Zeichenkette
<code>string s(t)</code>	vereinbart Zeichenkette, kopiert Zeichenkette <i>t</i> auf <i>s</i>
<code>string s(t,i)</code>	vereinbart Zeichenkette, kopiert <i>t</i> auf <i>s</i> ab Index <i>i</i>
<code>string s(t,i,n)</code>	wie <code>string s(t,i)</code> , höchstens <i>n</i> Zeichen werden kopiert
<code>string s(ct)</code>	vereinbart Zeichenkette, kopiert C-Zeichenkette <i>ct</i> auf <i>s</i>
<code>string s(n,c)</code>	vereinbart Zeichenkette und initialisiert sie mit <i>n</i> Zeichen <i>c</i>
<code>s[i]</code>	Komponentenwert zum Index <i>i</i>
<code>s.at(i)</code>	Komponentenwert zum Index <i>i</i> mit Bereichsüberwachung
<code>s=t s=ct s=c</code>	weist Zeichenkette, C-Zeichenkette bzw. Zeichen zu
<code>s+t s+ct ct+s</code>	liefert Verkettung zweier Zeichenketten als neue Zeichenkette
<code>s+=t s+=ct s+=c</code>	hängt Zeichenkette, C-Zeichenkette bzw. Zeichen an <i>s</i> an
<code>s==t s!=t s>t s<t s>=t s<=t</code>	vergleicht <i>s</i> und <i>t</i> lexikographisch
<code>s==ct s!=ct s>ct s<ct s>=ct s<=ct</code>	vergleicht <i>s</i> und <i>ct</i> lexikographisch
<code>cs==t cs!=t cs>t cs<t cs>=t cs<=t</code>	vergleicht <i>cs</i> und <i>t</i> lexikographisch
<code>s.size() s.length()</code>	liefert Zeichenzahl (als <code>string::size_type</code>)
<code>s.max_size()</code>	liefert Maximalzahl von Zeichen in <i>s</i>
<code>s.resize(n)</code>	ändert Komponentenzahl, ggf. Erzeugung und Initialisierung neuer Komponenten bzw. Löschen vorhandener Komponenten
<code>s.substr(i) s.substr(i,n)</code>	liefert Teilzeichenkette ab Index <i>i</i> bzw. mit max. Länge <i>n</i> als neue Zeichenkette (falls $i \leq s.size()$)
<code>s.c_str()</code>	liefert <i>s</i> als konstante C-Zeichenkette (mit angehängtem <code>\0</code>)
<code>s.data()</code>	liefert <i>s</i> als konstanten C-Vektor aus <code>char</code> (ohne angehängtes <code>\0</code>)
<code>s.find(t) s.find(ct) s.find(c)</code>	sucht erstes <i>t,ct,c</i> in <i>s</i> , liefert Index oder <code>string::npos</code>
<code>s.find(t,i) s.find(ct,i)</code>	sucht erstes <i>t,ct,c</i> in <i>s</i> ab Index <i>i</i> , liefert Index
<code>s.find(c,i)</code>	oder <code>string::npos</code>

<code>s.rfind(t)</code>	<code>s.rfind(ct)</code>	<code>s.rfind(c)</code>	wie <code>s.find</code> , jedoch Suche nach letztem Auftreten
<code>s.rfind(t,i)</code>	<code>s.rfind(ct,i)</code>		”
<code>s.rfind(c,i)</code>			”
<code>s.insert(i,t)</code>	<code>s.insert(i,ct)</code>		fügt <code>t</code> bzw. <code>ct</code> in <code>s</code> ab Index <code>i</code> ein (falls $i \leq s.size()$)
<code>s.erase(i)</code>	<code>s.erase(i,n)</code>		entfernt alle bzw. max. <code>n</code> Zeichen von <code>s</code> ab Index <code>i</code> (falls $i \leq s.size()$)
<code>s.replace(i,n,t)</code>	<code>s.replace(i,n,ct)</code>		ersetzt max. <code>n</code> Zeichen in <code>s</code> ab Index <code>i</code> durch <code>t</code> bzw. <code>ct</code> (falls $i \leq s.size()$)
<code>s.copy(ct,n,i)</code>			kopiert max. <code>n</code> Zeichen ab Index <code>i</code> (Voreinst.: 0) von <code>s</code> nach <code>ct</code> (falls $i \leq s.size()$), <code>\0</code> wird <i>nicht</i> angehängt
<code>cout << s</code>	<code>cin >> s</code>		Ausgabe, Eingabe (Länge der Zeichenkette <code>s</code> wird angepasst)
<code>getline(cin,s)</code>			liest Zeile von Eingabe und schreibt diese nach Entfernen von <code>\n</code> in <code>s</code> (Längen Anpassung), liefert (Zustand von) <code>cin</code>

Beispiel: Zahldarstellung zur Basis B mit $2 \leq B \leq 36$

```
#include <iostream>
#include <string>

using namespace std;

string dec2base(unsigned int n, unsigned int base)
{
    string s; char c;
    int digit;

    // Abdividieren (liefert Ziffern von rechts nach links)
    do {
        digit = n%base;
        s += digit<10 ? '0'+digit : 'a'+digit-10;
        n /= base;
    } while (n!=0);

    // Ziffernreihenfolge umkehren
    for (string::size_type i=0; i<s.size()/2; ++i) {
        c = s[i];
        s[i] = s[s.size()-1-i];
        s[s.size()-1-i] = c;
    }

    return s;
}

int main()
{
    unsigned int n;
    cout << "n: "; cin >> n;

    cout << "Binaerdarstellung: " << dec2base(n,2) << endl;
    cout << "Oktaldarstellung: " << dec2base(n,8) << endl;
    cout << "Hex.darstellung: " << dec2base(n,16) << endl;

    return 0;
}
```