Programmentwicklung II

Bachelor of Science

Prof. Dr. Rethmann / Prof. Dr. Brandt

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

Sommersemester 2021

Include-Dateien von C nutzen

Alte Bibliotheken aus C können in C++ weiterhin verwendet werden, allerdings wird beim Einbinden die Endung `.h` weggelassen und stattdessen vor den Namen der Bibliothek der Buchstabe `c` voran gestellt:

- aus `#include <math.h>` wird `#include <cmath>`
- aus `#include <stdlib.h>` wird `#include <cstdlib>`
- aus `#include <string.h>` wird `#include <cstring>`
- aus `#include <time.h>` wird `#include <ctime>`

Funktionen wie `sin`, `pow`, `sqrt`, `rand`, `strstr` oder `localtime` sind dann im namespace `std` vorhanden.

Oft gibt es alternative oder ergänzende Bibliotheken:

- `numeric` und `numbers` als Ergänzung zu `cmath`
- `chrono` als Ergänzung für `ctime`

Dazu später mehr. Schauen wir uns zunächst einige wichtige Klassen an, die wir zum Programmieren in den Übungen benötigen.

Input-Streams ermöglichen Eingaben über die Tastatur, Output-Streams ermöglichen Ausgaben auf dem Bildschirm.

Sie werden über die Operatoren << zur Ausgabe auf dem Bildschirm und >> zum Einlesen von der Tastatur angesprochen:

```
#include <iostream>      // keine Endung .h !

int main(void) {
    int a;

    std::cout << "Bitte int-Wert eingeben: ";
    std::cin >> a;       // kein Adressoperator !

    char c = 'a';
    std::cout << c << " = " << a << std::endl;
    return 0;
}
```

Damit wir nicht überall std:: schreiben müssen, können wir die using-Direktive using namespace std; benutzen:

```
#include <iostream>
using namespace std;        // damit kein "std::" notwendig

int main(void) {
    int a;

    cout << "Bitte int-Wert eingeben: ";
    cin >> a;

    char c = 'a';
    cout << c << " = " << a << endl;
}
```

Wird die using-Direktive genutzt, können alle Namen eines Namensbereichs wie „normale“ Namen verwendet werden.

Wird stattdessen eine using-Deklaration wie using std::cout benutzt, kann nur der angegebene Name ohne Qualifizierung genutzt werden:

```
#include <iostream>
using std::cout;

int main(void) {
    cout << "Hello, world!" << std::endl;
    return 0;
}
```

Formatierte Ausgabe: `iomanip`

- `setw`: konstante Breite festlegen
- `setfill(char)`: Zeichen zum Auffüllen festlegen
- `setbase`: Basis der Zahl festlegen (oktal, dezimal oder hexadezimal)
- `setprecision`: Anzahl der Nachkommastellen festlegen
- `fixed`: Festkommadarstellung
- `scientific`: Exponentialdarstellung

Formatierte Ausgabe: iomanip

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

int main(void) {
    cout << setbase(8) << endl;
    for (int i = 1; i <= 1000000; i *= 10)
        cout << i << endl;
}
```

Ausgabe:

```
1
12
144
1750
23420
303240
3641100
```

Formatierte Ausgabe: iomanip

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

int main(void) {
    cout << setfill('_') << endl;
    for (int i = 1; i <= 1000000; i *= 10)
        cout << setw(10) << i << endl;
}
```

Ausgabe:

```
_________1
________10
_______100
______1000
_____10000
____100000
___1000000
```

Formatierte Ausgabe: iomanip

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

int main(void) {
    cout << setprecision(5) << endl;
    for (double d= 0.1; d >= 0.0000001; d /= 10)
        cout << d << endl;
}
```

Ausgabe:

```
0.1
0.01
0.001
0.0001
1e-05
1e-06
1e-07
```

Formatierte Ausgabe: iomanip

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

int main(void) {
    cout << setprecision(5) << fixed << endl;
    for (double d= 0.1; d >= 0.0000001; d /= 10)
        cout << d << endl;
}
```

Ausgabe:

```
0.10000
0.01000
0.00100
0.00010
0.00001
0.00000
0.00000
```

Formatierte Ausgabe: iomanip

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

int main(void) {
    cout << scientific << endl;
    for (double d= 0.1; d >= 0.0000001; d /= 10)
        cout << d << endl;
}
```

Ausgabe:

```
1.000000e-01
1.000000e-02
1.000000e-03
1.000000e-04
1.000000e-05
1.000000e-06
1.000000e-07
```

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

int main() {
    string t("Ene mene muh und raus bist du!");

    cout << t << "\nsize: " << t.size() << endl;
    cout << "substr: " << t.substr(13, 8) << endl;
    size_t pos = t.find("muh");
    cout << "substr: " << t.substr(pos) << endl;
}
```

The substring is the portion of the object that starts at character position pos and spans len characters (or until the end of the string, whichever comes first).

- `size_t` is an unsigned integral type
- a value of `string::npos` indicates all characters until the end of the string
- `string substr(size_t pos = 0, size_t len = npos) const;`

Wieso kann die Methode `substr` mal mit einem, mal mit zwei Parametern aufgerufen werden? Wie ist so etwas realisiert?

Wir kennen das bereits aus C von den Funktionen `printf` und `scanf`. Die Standard-Bibliothek `stdarg` bietet die Möglichkeit, eine Liste von Funktionsargumenten abzuarbeiten, deren Länge und Datentypen nicht bekannt sind.

```
int printf(const char *format, ...);    // korrekte Syntax
int scanf(const char *format, ...);  // für Deklaration !
```

Wenn eine Parameterliste mit `...` endet, darf die Funktion mehr Argumente akzeptieren als Parameter explizit beschrieben sind. Die drei Punkte `...` dürfen nur am Ende einer Argumentenliste stehen.

Mit dem Typ `va_list` definiert man eine Variable, die der Reihe nach auf jedes Argument verweist.

```
va_list vl;
```

Das Makro va_start initialisiert vl so, dass die Variable auf das erste unbenannte Argument zeigt. Das Makro muss einmal aufgerufen werden, bevor vl benutzt wird. Es muss mindestens einen Parameter mit Namen geben, da va_start den letzten Parameternamen benutzt, um anzufangen.

```
va_start(vl, format);
```

Jeder Aufruf des Makros va_arg liefert ein Argument und bewegt vl auf das nächste Argument. Das Makro va_arg benutzt einen Typnamen, um zu entscheiden, welcher Datentyp geliefert und wie vl fortgeschrieben wird.

```
int ival = va_arg(vl, int);
char *sval = va_arg(vl, char *);
```

Vorsicht: Der Typ des Arguments wird nicht automatisch erkannt. Um den korrekten Typ angeben zu können, wird ein Format-String benutzt.

Eventuell notwendige Aufräumarbeiten erledigt va_end.

```
va_end(vl);
```

```c
#include <stdio.h>
#include <stdarg.h>

int fkt(const char *fmt, ...);
void getAndPrintNextValue(char c, va_list *l);

int main(void) {
    int n;

    n = fkt("sfdsd", "eins", 2.0, 3, "vier", 5);
    printf("--> %d Argumente gelesen\n", n);

    n = fkt("fdsd", 6.0, 7, "acht", 9);
    printf("--> %d Argumente gelesen\n", n);

    return 0;
}
```

```c
int fkt(const char *fmt, ...) {
    int z;
    va_list l;

    va_start(l, fmt);

    for (z = 0; *fmt; fmt++, z++)
        getAndPrintNextValue(*fmt, &l);

    va_end(l);
    return z;
}
```

```
void getAndPrintNextValue(char c, va_list *l) {
    char *sval;
    int ival;
    double fval;

    if (c == 'd') {
        ival = va_arg(*l, int);
        printf("%d (int)\n", ival);
    } else if (c == 'f') {
        fval = va_arg(*l, double);
        printf("%f (double)\n", fval);
    } else if (c == 's') {
        sval = va_arg(*l, char *);
        printf("%s (char *)\n", sval);
    }
}
```

Ausgabe:

```
eins (char *)
2.000000 (double)
3 (int)
vier (char *)
5 (int)
--> 5 Argumente gelesen
6.000000 (double)
7 (int)
acht (char *)
9 (int)
--> 4 Argumente gelesen
```

Die Implementierung der Methoden `substr(pos, len)` und `substr(pos)` basieren nicht auf variabel langen Parameterlisten, sondern auf Default-Parameter.

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

int main() {
    std::string s("Ene mene muh");
    std::string t(" und raus bist du!");
    std::string u(" noch lange nicht");

    s.append(t);            // oder:  s += t;
    std::cout << s << std::endl;

    size_t pos = s.find("!");
    s.insert(pos, u);
    std::cout << s << std::endl;

    s.replace(pos, u.size(), " jetzt doch");
    std::cout << s << std::endl;
}
```

Wo ist der Fehler im folgenden C-Programm?

```c
#include <stdio.h>
#include <string.h>

int main(void) {
    char *s = "Hallo Welt";
    char *t = strtok(s, " ");

    printf("%s\n", t);
    t = strtok(NULL, " ");
    printf("%s\n", t);

    return 0;
}
```

File I/O

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

int main() {
    fstream f;

    f.open("test1.dat", ios::app | ios::out);

    if (f.is_open()) {
        f.write("Hello, World!\n", 14);
        f.close();
    } else cerr << "failed to open test1.dat\n";

    return 0;
}
```

Beim Öffnen verschiedene Modi mittels ODER verknüpfen:

- `ios::app` append output
- `ios::ate` seek to EOF when opened
- `ios::binary` open the file in binary mode
- `ios::in` open the file for reading
- `ios::out` open the file for writing
- `ios::trunc` overwrite the existing file

Online-Tutorials:

- `http://www.cplusplus.com/`
- `https://en.cppreference.com/w/Main_Page`

Vereinfachung: Im Konstruktor kann die zu öffnende Datei angegeben werden, außerdem sind Operator-Überladungen definiert.

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

int main() {
    fstream f("test1.dat", ios::app | ios::out);

    if (!f) {
        cerr << "failed to open test1.dat\n";
        return 1;
    }

    f << "Hello, World!\n";
    f.close();
}
```

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

int main() {
    fstream f("test1.dat", ios::in);
    char line[256];          // kein string !!!!!

    if (! f.is_open()) {
        cerr << "failed to open test1.dat\n";
        return 1;
    }

    while (! f.eof()) {
        f.getline(line, 256);
        cout << line << endl;
    }
    f.close();
}
```

Dasselbe mit Strings:

```
#include .....

int main() {
    fstream f("test1.dat", ios::in);
    string line;              // string statt char *

    if (! f.is_open()) {
        cerr << "failed to open test1.dat" << endl;
        return 1;
    }

    while (! f.eof()) {
        getline(f, line);       // globale Funktion
        cout << line << endl;
    }
    f.close();
}
```

Auch beim Lesen sind Operatorüberladungen definiert. Positionieren in Dateien ist mittels `tellg` und `seekg` möglich.

```
int main() {
    char word[256];
    long start, end;
    fstream f("test1.dat", ios::in);

    start = f.tellg();
    f.seekg(0, ios::end);       // offset, direction
    end = f.tellg();
    cout << "size is: " << (end-start) << " bytes." << endl;

    f.seekg(0, ios::beg);       // back to beginning
    f >> word;
    cout << word << endl;
}
```

Positionieren beim Schreiben mittels `tellp` und `seekp`.

Seit 2017 gibt es eine zusätzliche Biliothek `filesystem`, die den Umgang mit Dateien und Verzeichnissen vereinfacht.

```
#include <iostream>
#include <fstream>
#include <filesystem>

namespace fs = std::filesystem;
using namespace std;

const int SIZE = 256;

int main(void) {
    int values[SIZE], arr[SIZE];
    fstream file;
    fs::path path = fs::current_path() / "myfile";

    for (int i = 0; i < SIZE; i++)
        values[i] = i;
```

```
    file.open(path, ios::out | ios::binary | ios::trunc);

    cout << "filesize: " << fs::file_size(path) << endl;
    cout << "free in file: " << fs::space(path).free << endl;
    file.write((char *)values, SIZE * sizeof(int));
    file.close();

    cout << "filesize: " << fs::file_size(path) << endl;
    fs::resize_file(path, SIZE / 2 * sizeof(int));

    file.open(path, ios::in | ios::binary);
    cout << "filesize: " << fs::file_size(path) << endl;
    file.read((char *)arr, SIZE * sizeof(int));

    for (int i = 0; i < SIZE; i++)
        if (values[i] != arr[i])
            cout << i << ": " << values[i]
                 << " :: " << arr[i] << endl;
}
```

Die Streams `cin`, `cout` und `cerr` kennen wir schon aus C, dort hießen sie allerdings anders:

```
fprintf(stdout, "Hallo!");  // printf("Hallo!");
fscanf(stdin, "%d", &i);    // scanf("%d", &i);
fprintf(stderr, "Hallo!");  // ????
```

Die Streams `cerr` und `stderr` sind nicht gebuffert, sodass die Nachrichten direkt angezeigt werden, und nicht erst nach einem `endl`.

Soll auch ohne `endl` bereits auf `cout` ausgegeben werden, kann `std::flush` genutzt werden:

```
cout << "dieser Teil wird schon ausgegeben ..." << flush;
longComputation();   // oder einfach:  sleep(2)
cout << " bevor jetzt new-line ausgegeben wird" << endl;
```

Was wird bei dem folgenden Programm in die Datei geschrieben?

```
#include <fstream>
.....
class Klasse {
    ofstream file;

public:
    Klasse(char *name = "default.txt") {
        file.open(name);
        if (!file) {
            cout << "could not open file" << endl;
            return;
        } else file << "Eins" << endl;
    }

    void function() {
        file << "Zwei" << endl;
    }
```

```
    ~Klasse() {
        file << "Drei" << endl;
        file.close();
    }
};

int main(void) {
    Klasse *k = new Klasse();

    k->function();
    return 0;
}
```

Was wird in die Datei geschrieben?

```
class Datum {
private:
    int _tag, _monat, _jahr;

public:
    Datum(int t = 1, int m = 1, int j = 2000);
    Datum(string dat);     // Implementierung ?

    bool istSchaltjahr();
    int kalenderwoche();
    int tagImJahr();
    string toString();     // Implementierung ?
};

int main() {
    Datum d(7, 8, 2019);
    cout << d.toString() << endl;
}
```

So funktioniert es leider nicht:

```
string toString() {
    string str;

    str += _tag;
    str += ".";
    str += _monat;
    str += ".";
    str += _jahr;

    return str;
}
```

Die Attribute `_tag`, `_monat` und `_jahr` sind vom Typ `int`, für den weder eine Operatorüberladung noch eine Methode `append` in der Klasse `string` definiert ist!

So funktioniert es:

```
string toString() {
    ostringstream os;

    os << setfill('0') << setw(2) << _tag << ".";
    os << setw(2) << _monat << "." << setw(4) << _jahr;

    return os.str();
}
```

Ein Objekt der Klasse `ostringstream` verhält sich wie ein Objekt der Klasse `ostream`, nur das die Werte nicht auf dem Bildschirm ausgegeben werden sondern in einen `string` geschrieben werden.

Damit obiger Code funktioniert, müssen die Header `sstream` und `iomanip` eingebunden werden.

Wir können auch einen Konstruktor definieren, der die Daten aus einem `string` ausliest:

```
Datum(string dat) {
    istringstream is(dat);
    char t;

    is >> _tag;
    is >> t;
    is >> _monat;
    is >> t;
    is >> _jahr;
}
```

Der Aufruf `Datum d("27.3.2021");` würde jeweils den Punkt in der Variablen `t` speichern und verwerfen, Tag, Monat und Jahr werden in die entsprechenden Klassenattribute eingetragen.

Würde das obige Beispiel auch ohne die Variable `t` funktionieren?

```
Datum(string dat) {
    istringstream is(dat);

    is >> _tag >> _monat >> _jahr;
}
```

Nein! Sonst würde der Monat nicht gelesen werden, da der Punkt „.“ im Datum nicht als Wert vom Typ `int` eingelesen werden kann.