

Draft der C++-Vorlesung vom 31.März 2023

0. Übungsblätter 2-4

0. Zur Erinnerung an Tafel: Header- und Source-Files (Deklaration vs. Definition)

1. Parameterübergabe an Funktionen: Sie können Variablen auf drei Arten in der Parameterliste einer Funktion anführen [Haase23, §7.2]:

- per Value (= Kopie): `vector<double>`
- per Referenz (= Original mit anderem Bezeichner): `vector<double>&`
- per Pointer: `vector<double>*`

Verboten in dieser LV!

Damit können Sie **Input**-Parameter

- als Kopie (`vector<double>`) oder
- als konstante Referenz (`vector<double> const&`)

übergeben. Reine **Output**-Parameter und InOut-Parameter werden

- als veränderbare Referenz (`vector<double>&`)

gehandhabt.

2. Signatur einer Funktion: Siehe [Haase23, §7.1] und Beispiel `v_3_signatur`¹.

Damit eine Funktion beim Linken (der bereits kompilierten Programmteile) ihren Aufrufen in anderen Programmteilen zugeordnet werden kann, ist eine eindeutige Kennung (= *Signatur*) nötig.

- **Signatur** := Funktionsname + Datentypen der Parameterliste, z.B.

```
double square(double x) // main:6
int square(int x) // main:13
void print_vek(vector<float> const & v) // v_3c/main:59
```
- Beachte beim letzten Bsp., daß

```
void print_vek(vector<float> & v)
```

 eine neue Signatur darstellt, da das `const` in den Datentyp eingeht.

3. IO-System und File-IO: Die **Ein**- und **Ausgabe** von Daten in Programmen erfolgt über Datenströme (*stream*). Sie kennen bereits die Datenströme `cout` und `cin` mit ihren zugehörigen Operatoren `<<` und `>>`. Zusätzlich haben wir die Fehlerausgabe über `cerr` welche ebenfalls mit `#include <iostream>` inkludiert wird.

Diese Datenströme lassen sich aus/in Files umlenken [Haase23, §8], was bei größeren Datenmengen natürlich einen enormen Vorteil darstellt.

- Statt mit `cout << d` geben wir eine double-Variable `d` in das File `out.txt` aus, siehe dazu Beispiel `v_5a`²:

```
#include <fstream> // ofstream
#include <iostream>
using namespace std;
...
ofstream out_file("out.txt"); // ASCII-File als Ausgabefile
out_file << d;
```
- Wir haben ein Text/ASCII³-file benutzt, Binärfiles sind ebenfalls möglich⁴.

¹http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS23/v_3_signatur.zip

²http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS23/v_5a.zip

³<https://de.wikipedia.org/wiki/Textdatei>

⁴<http://www.cplusplus.com/forum/general/21018/>

- Statt zwei String-Variablen `c1`, `c2` mit `cin>>c1>>c2` über die Tastatur einzugeben, lesen wir diese vom ASCII-File `my_in.txt` ein.

```
#include <fstream> // ofstream
#include <iostream>
using namespace std;
...
ifstream in_file("my_in.txt"); // ASCII-File als Eingabefile
in_file >> c1 >> c2;
```

- Siehe auch das ausführliche Beispiel `Simple_FileIO`⁵.
- Wenn die Operatoren `<<` und `>>` für einen Datentyp definiert sind, so sind obige IO-Operation mit `cin/cout` oder via Files möglich. Anderfalls müssen diese beiden Operatoren für den neuen Datentyp definiert werden [Haase23, §9.8].

4. Einlesen eines Vektors vom ASCII-File: Das Beispiel `file_io`⁶ (html⁷) demonstriert das Lesen/Schreiben eines `double`-Vektors via ASCII-Files.

- Das kurze Hauptprogramm demonstriert die Anwendung der Leserfunktion (`main.cpp:16`) und der Schreibfunktion (`main.cpp:24`).
- Die Lesefunktion `read_vector_from_file` (`file_io.cpp:30`) öffnet das File und bricht mit einer Fehlermeldung ab, falls das Inputfile nicht gefunden wird.
- In obigem Erfolgsfalle wird der Vektor in der Funktion `fill_vector` solange mit Daten gefüllt (und dabei dynamisch verlängert, `file_io.cpp:14`) bis das Fileende erreicht ist. Zeilen `file_io.cpp:15-24` dienen nur der möglichen Fehlerbehandlung [Stroustrup10, p.364] und `file_io.cpp:25` verkürzt den Vektor auf die nötige Länge.
- Die Schreibfunktion `write_vector_to_file` ist selbserklärend.

5. Funktion `main`: Siehe [Haase23, §7.6] und Bsp. `v_5b`⁸.

Das Hauptprogramm ist eine Funktion mit dem vorgeschriebenem Namen `main` und dem Interface

```
int main( int const argc, char const* argv[] )
```

- Der Wert des Rückgabeparameters vom Typ `int` kann in der aufrufenden Programmumgebung ausgewertet werden. Ein Wert 0 steht hierbei für fehlerfreie Programmabarbeitung. Daher auch die standardmäßige Zeile `return 0;` am Ende der Funktion `main`.
- Umgekehrt kann die Programmumgebung der Funktion `main` eine beliebige Anzahl von *Kommandozeilen*parametern übergeben, allerdings nur als klassische C-Strings (`'\0'` terminated). Folglich sind
 - `argc` die Anzahl dieser C-Strings und
 - `argv[]` is ein (C-)Array mit solchen C-Strings.
 - Es gilt stets $argc \geq 1$, da `argv[0]` stets den Namen des laufenden Programmes enthält.
 - Welchen Nutzen habe ich von Kommandozeilenparametern?
Sie können Eingabeparameter an Ihr Programm ohne lästiges (mehrfaches) Eintippen oder neues Compilieren übergeben (Erhöhung der Flexibilität).

⁵http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS23/Simple_FileIO.zip

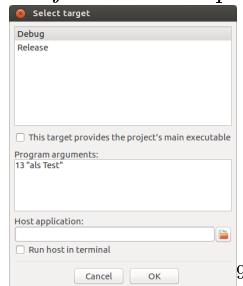
⁶http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS23/file_io.zip

⁷http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS23/file_io/html/file_io_8cpp.html

⁸http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS23/v_5b.zip

- Wie kann ich solche Kommandozeilenparameter an mein Programm übergeben?
 - Aus der Kommandozeile im Terminal mit Leerzeichen getrennt, z.B.
`v_5b.exe 13 "als Test"`
 - In CodeBlocks über die Einstellung

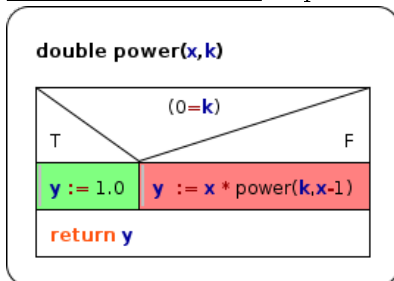
Project → *Set program's arguments* → *P. arguments* → 13 "als Test"



- Wie nutze ich diese Kommandozeilenparameter in meinem Programm?
 - Wurden Kommandozeilenparameter übergeben? → `main.cpp:24`
 - Falls nicht, dann Eingabe über Tastatur → `main.cpp:32-37`
 - Falls ja, dann Konvertierung in benötigten Datentyp, → `main.cpp:26`
 in diesem Fall wird der C-String mittels der Funktion `atoi(argv[1])` (sprich `AsciiToInteger`) auf eine Variable des Datentyps `int` umgewandelt.
 - Eine Auswahl von Konvertierungsfunktionen:

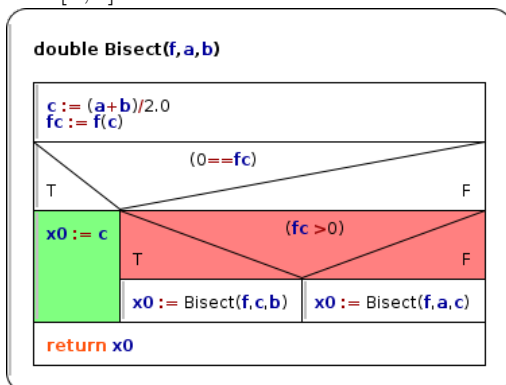
<code>atoi, atol, atof, strtol, strtod, ...</code>	<code>#include <cstdlib></code>
<code>stoi, stol, stof, stod</code>	<code>#include <string></code>
- Die Umwandlung von numerischen Werten in C++-Strings erfolgt mittels der Funktion `to_string`, siehe Bsp. `to_string`¹⁰.

6. Rekursive Funktion: Bsp. Power, siehe [Haase23, §7.7] und Bsp. `Ex770.cpp`¹¹.



Jede Rekursion hat einen (mehrfache) Test(s) auf Abbruch woraufhin **Standardwerte im Abbruchzweig** zurückgegeben werden. Ansonsten wird die Funktion mit geänderten Parametern im **Rekursionsteil** aufgerufen.

7. Rekursive Funktion: Bsp. Bisektion, Bestimme eine Nullstelle der Funktion $f(x)$ mit $x \in [a, b]$.



Den Anfang von [Haase23, §7.8] aufschlagen und lesen.

$$x_0 := \text{Bisect}(a, b, \epsilon)$$

$$:= \begin{cases} c := (a + b)/2 & \text{falls } f(c) == 0 \\ \text{Bisect}(c, b, \epsilon) & \text{sonst, falls } f(c) > 0 \\ \text{Bisect}(a, c, \epsilon) & \text{sonst, falls } f(c) < 0 \end{cases}$$

⁹http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS23/v_5b_arguments.png

¹⁰http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS23/to_string.zip

¹¹<http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS23/./Beispiele/Ex770.cpp>

- Wegen numerischer Rundungsfehler der Gleitkommazahlen kann es passieren, daß der Test des Funktionswertes auf die exakte 0, $0 == fc$, niemals erfüllt wird. Um diese *Endlosrekursion* zu vermeiden und um nur bis zu einer *sinnvollen Genauigkeit* zu rechnen, wird dieser Test durch $|fc| < \varepsilon$ ersetzt. Das ε kann/sollte als Inputparameter an die Funktion übergeben werden, wie in `Bisect1.cpp:15`¹².

Die math. Funktion $\sin(x) - x/2$ ist in `Bisect1.cpp:49` direkt im Funktionskörper implementiert, dies geht flexibler.

- **C++-Funktionen als Inputparameter** von Funktionen:

Die in `Bisect` benötigte Funktion $f(x)$ besitzt die Signatur `double f(double)`, siehe `Bisect3.cpp:18`¹³. Eine Funktion dieser Signatur mit spezifiziertem **Rückgabetypen** kann via

```
const std::function<double(double)>& func
```

als Inputparameter einer Funktion deklariert werden. Damit ergibt sich die Funktionsdeklaration (`Bisect3.cpp:49`)

```
double Bisect3(const std::function<double(double)>& func,
               const double a, const double b, const double eps=1e-6);
```

welche in `Bisect3.cpp:67` aus dem Hauptprogramm aufgerufen wird.

Benötigt `#include <functional>` und die Compileroption `'-std=c++11'`.

- In obiger Deklaration ist der Parameter `double eps=1e-6` ein **optionaler** Parameter mit dem **Standardwert** `1e-6` (default). Dies erlaubt den Aufruf der Funktion als

– `Bisect3(g, anfang, ende, tol*tol)` , aber auch

– `Bisect3(g, anfang, ende)` wobei dann `eps=1e-6` gilt.

Bislang galt stets die Voraussetzung $f(a) > 0 > f(b)$ welche in der abschließenden Implementierung `v_5d`¹⁴ nicht mehr nötig ist.

- Vollständige Fallunterscheidung bzgl. der Relationen von $f(a)$, 0 und $f(b)$, siehe dazu [Haase23, p.67f] und `v_5d/main.cpp:113-137`.

- Flexible Auswahl aus mehreren, vordefinierten Funktionen für $f(x)$, siehe dazu die Zuweisungen der Variablen

```
std::function<double(double)> ff;
```

über die Fallauswahl in `v_5d/main.cpp:78-100`.

Literatur

[Haase23] Gundolf Haase: Einführung in die Programmierung mit C++ (2023), *www*¹⁵.

[Stroustrup10] Bjarne Stroustrup: Einführung in die Programmierung mit C++. Pearson Studium, München (2010).

¹²<http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS23/./Beispiele/Bisect1.cpp>

¹³<http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS23/./Beispiele/Bisect3.cpp>

¹⁴http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/SS23/v_5d.zip

¹⁵http://imsc.uni-graz.at/haasegu/Lectures/Kurs-C/Script/html/script_programmieren.pdf