FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Ergänzungen zur Vorlesung

Einführung in C++

Prof. Dr. rer. nat. Alexander Voß

Fachbereich Medizintechnik und Technomathematik
Dualer Studiengang Scientific Programming
FH Aachen

Kontakt a.voss@fh-aachen.de
Stand 07.10.2016

Allgemeines zur Vorlesung

Voraussetzungen

- Gute Programmierkenntnisse in Java.
- Eine installierte virtuelle Machine (VM) 'FH Aachen - CPP' in VirtualBox (alternativ VMware Player).

Ziele der Vorlesung

- Eigene C und C++ Programme schreiben zu können.
- Einen Überblick über die Programmiersprache C++ inkl. C++11/14-Sprachfeatures und häufig verwendeter Programmiermuster zu geben.

Eine Übersicht der behandelten und damit prüfungsrelevanten Themen finden Sie am Ende.

Kern der Vorlesung

- Code-Snippets in Form von Beispielprojekten.
- Links und Stichpunkte, um *selbständig* Aspekte von C++ weiter zu verfolgen und zu verstehen.
- Lernstandskontrollen und Übungen.

Quellen

- Begleitliteratur
  - Der C++ Programmierer. Breymann.
  - C++ - kurz & gut. Loudon.
- Weitere Quellen
  - cppreference.com

Generell gilt: Wenn Sie gerne in Büchern lesen, suchen Sie sich welche, die Sie vom Stil her ansprechend finden und die Ihrer Art zu Lernen und Ihrem Niveau entsprechen.

Prüfung

- Es gibt eine schriftliche Klausur über 2h oder eine mündliche Prüfung über 30 min (abhängig von der Anzahl der Anmeldungen).
- Die schriftliche Klausur wird in elektronischer Form geschrieben, also am eigenen Laptop. Es handelt sich um eine sog. Kofferklausur, bei der alle Unterlagen wie Übungen, Aufzeichnungen, Bücher und natürlich die Code-Snippets zulässig sind.
- Nicht zugelassen ist jegliche Kommunikation, insbesondere über das Internet.
- Prüfungsrelevant sind grundsätzlich alle Themen, die in der Vorlesung oder Übung behandelt worden sind. Eine Übersicht finden Sie am Ende.

Beachten Sie die geltenden Anmelderegularien und -zeiträume!

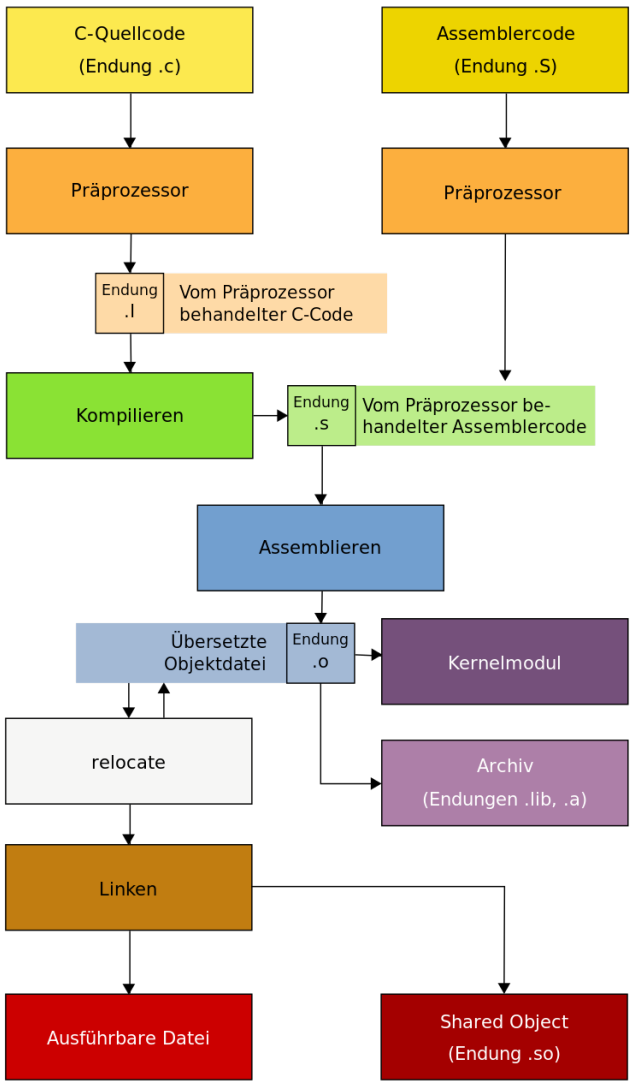
Wissenswertes zu C++

Übersicht

- 1979 von Bjarne Stroustrup bei AT&T als Erweiterung der Programmiersprache C entwickelt.
- Aktuelle Fassung C++14 (Jan. 2015).
- Als ISO Standard registriert (ISO/IEC 14882:2014).
- Zählt zu den objektorientierten Programmiersprachen und ermöglicht eine effiziente und maschinennahe Programmierung, aber auch eine Programmierung auf hohem Abstraktionsniveau.

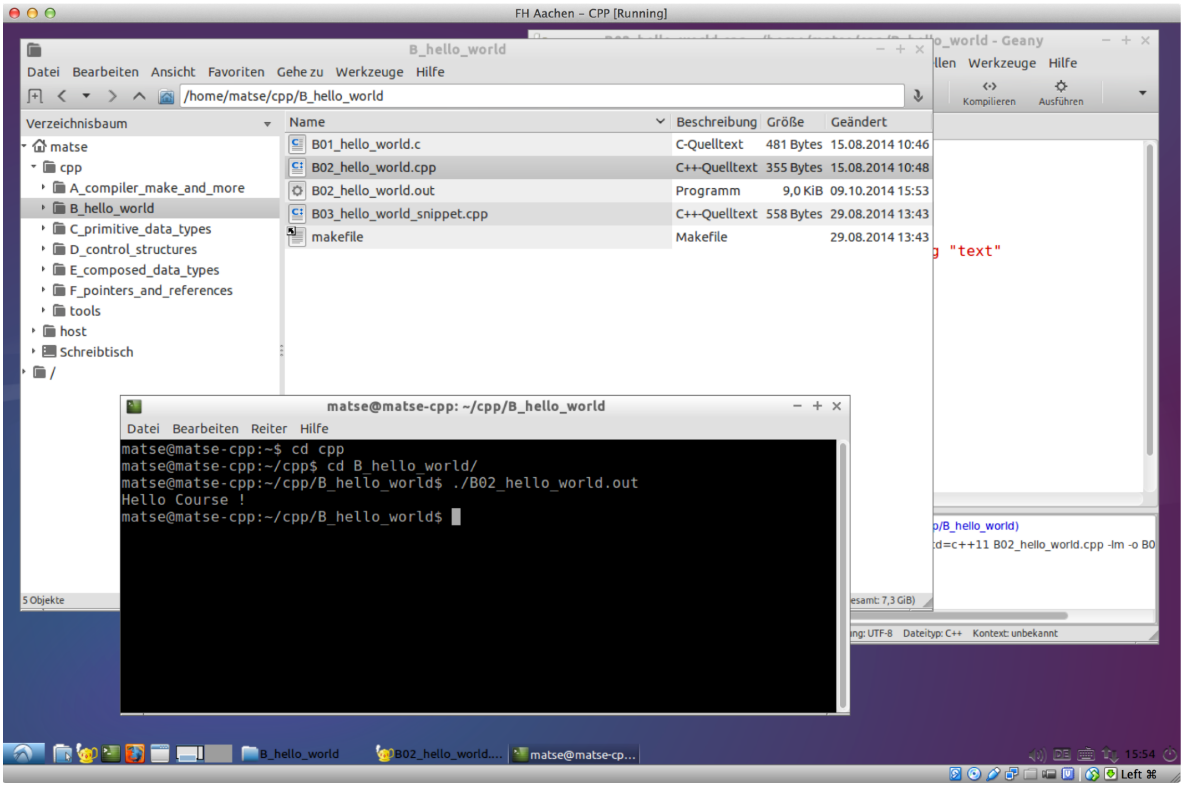
Compiler

- Es gibt keine virtuellen Maschinen oder Zwischensprachen. Stattdessen werden Programme direkt für eine Zielplattform erzeugt und laufen dort 'native'.
- Design Flow GNU Compiler Collection gcc (Wikipedia):
  - Aufruf des Compilers 'gcc'.
  - Compilieren und Erstellen von Assemblercode.
  - Linken und Erstellen einer ausführbaren Datei
- Alternativen zu gcc/g++ (Auswahl)
  - Microsoft Visual Studio C++
  - Intel C++ Studio
  - LLVM/Clang



Ubuntu VM

IDE



- Compiler und IDE sind grundsätzlich frei wählbar, hier: virtuelle Maschine (VM) mit installiertem Ubuntu
- Tools sind:
  - Geany, als einfacher Editor
  - gcc, g++ oder clang, clang++ als Compiler Suit

Code Snippets

Idee

- Anstelle von Folien zum jeweiligen Thema wird die Sprache anhand von Code-Snippets erläutert.
- Somit ist es möglich
  - die Konzepte der Sprache live zu erleben (und idealerweise auch zu verstehen)
  - Antworten auf Fragen und Details live in der Vorlesung herauszufinden – Fragen Sie!
  - unklare Konzepte direkt in der nächsten Übung mit Hilfe der Snippets auszuprobieren und nachzuvollziehen.

Code-Snippets ersetzen nicht das Lernen anhand von Begleitliteratur und das Nachlesen tiefergehender Details!

Ziele bei jedem Thema oder Sprachfeature

- Wissen, was es ist bzw. wovon es handelt.
- Wissen, wieso es existiert.
- Wissen, wie man es verwendet.

Beispiel Lambda-Ausdrücke

- Was? Lambda-Ausdrücke sind anonyme Funktionen, mit denen Typen für Delegaten erstellt und lokale Funktionen geschrieben werden können (Zitat).
- Wieso? Sie erlauben eine kurze und besser lesbare Syntax.
- Wie: `auto lambda x2 = [](double x){return x*x;};`

Genereller Workflow

- Source-Code mit Editor verändern und abspeichern.
- Compileraufruf `gcc` oder `g++` bzw. `make` zum Erstellen des Programms.
- Test durch Aufruf des Programms.

A,B,C,D – Übungen

Aufgabe 'Aldamir' [VM starten]

Führen Sie folgende Schritte aus:

- *VirtualBox* starten (alternativ VMware).
- Bereitgestellte Appliance (VM) in VirtualBox importieren, falls noch nicht geschehen.
- Virtuelle Maschine (VM) starten.
- Dateimanager *PCManFM* in VM starten.
- LXTerminal starten, einen Befehl (z.B. `ls -la`) eingeben und ausführen (Enter/Return).

Aufgabe 'Anborn' [Kommandozeilen-Befehle]

Starten Sie *LXTerminal* in der VM und probieren Sie die folgenden Kommandozeilen-Befehle aus bzw. lesen deren Bedeutung *(z.B. `man ls`):*

- *mkdir, rmdir, cd,*
- *ls, cp, mv, rm,*
- *cat, more, chmod, jobs, kill, bg,*
- *zip, unzip*

Beachten Sie:

- Wenn Sie einen Befehl mit einem `&` abschließen, wird dieser im Hintergrund ausgeführt, ohne auf das Ende zu warten (z.B. `geany x.c &`).
- Wenn Sie eine Datei mit `rm` löschen, ist sie endgültig gelöscht.

Fragen Sie, falls Sie keine Erfahrung mit Kommandozeilen haben!

Aufgabe 'Aratan' [Shared Folder]

Richten Sie einen *Shared Folder* in Ihrer VM ein. Führen Sie diese Schritte durch:

- VM ordnungsgemäß runterfahren
- In den Einstellungen von *VirtualBox* einen Eintrag im Reiter *Shared Folders* unter *Machine Folders* einrichten. Verweisen Sie auf einen Ordner auf Ihrem Rechner und nennen diesen Eintrag z.B. `host`.
- VM neu starten.
- Im Verzeichnis `host` sollte der eingebundene Ordner sichtbar sein.

Fragen Sie, falls *mounten* nicht automatisch funktioniert.

Aufgabe 'Barahir' [`hello_world` compilieren und starten]

Starten Sie die VM und führen Sie folgende Schritte aus:

- Dateimanager *PCManFM* in VM starten.
- In den Ordner `cpp/B_hello_world` navigieren.
- Doppelt auf `B02_hello_world_C.cpp` klicken – der Editor *Geany* öffnet sich.
- Im Terminal das Programm durch Ausführen der Befehle m*ake* erzeugen und durch Aufruf `./B_hello_world` starten (man beachte den .)

Aufgabe 'Baranor' [cout]

Führen Sie folgende Schritte aus und verstehen Sie den Code:

- Legen Sie im Dateimanager *PCManFM* ein neues Verzeichnis an, z.B. namens `lectures`
- Kopieren Sie die Datei `B02_hello_world.cpp` als `Baranor.cpp` in das angelegte Verzeichnis.
- Kopieren Sie die Datei `makefile` aus dem Ordner `cpp/tools` ebenfalls in das angelegte Verzeichnis.
- Starten Sie den Editor *Geany* durch Doppelklick auf `Baranor.cpp.cpp`.
- Ändern Sie den Ausgabetext beliebig, kompilieren Sie das Programm neu und starten Sie es.
- Erweitern Sie nun das Programm und geben Sie die 6'er-Reihe von 0*6 bis 9*6 aus.

Aufgabe 'Cemendur' [Typen und Deklarationen]

Verstehen Sie die Datentypen und Deklarationen in

- `C_primitive_data_types`

Fragen Sie bei Unklarheiten!

Aufgabe 'Derufin' [Datentypen, Schleifen, cin, cout]

Berechnung der Potenz:

- Legen Sie analog zu Aufgabe 'Baranor' ein Programm `Derufin.cpp` an. Dieser Schritt wird im Folgenden nicht mehr erwähnt.
- Berechnen Sie b^n (b hoch n) für feste natürliche Zahlen b und n und geben das Ergebnis aus.
- Lesen Sie b und n von der Konsole ein.
- Schreiben Sie eine Funktion `Pot`, die b und n übergeben bekommt und das Ergebnis zurückgibt und ausgibt.

Aufgabe 'Denethor' [Datentypen, Schleifen, cin, cout]

Berechnung von Primzahlen:

- Berechnen Sie alle Primzahlen bis zu einer eingelesenen natürlichen Zahl n und geben Sie diese Primzahlen aus.
- Hinweis: der Modulo-Operator ist %.

Aufgabe 'Dior' ♙ [cin, Datentypen, Schleifen, Funktionsaufrufe]

Berechnung der Fibonacci-Folge:

- Berechnen Sie die ersten Fibonacci-Zahlen bis zu einem Index n, den Sie von der Konsole einlesen,
  $f_0 = 0, f_1 = 1, f_n = f_{n-2}+f_{n-1}$ für $n>1$
  nicht-rekursiv in einer Schleife und geben Sie die Zahlen aus.
- Formulieren Sie die Schleife als *for, do-while* und *while*-Schleife.
- Berechnen Sie die Folge rekursiv.

Aufgabe 'Duilin' ♙ [cin, Strings, Konvertierung]

- Lesen Sie Text von der Konsole ein und versuchen Sie diesen in `int`, `float` und `double` Zahlen umzuwandeln. Benutzen Sie dazu die Funktionen `stoi`, `stof` und `stod`. Konvertieren Sie das Ergebnis zurück in einen String mittels `to_string`.
- Was passiert bei fehlerhaften Eingaben?

Aufgabe 'Duinhir' [Snippets]

Verstehen Sie alle Programme in

- `D_control_structures`

Fragen Sie bei Unklarheiten!

E – Übungen

Aufgabe 'Earendil' [struct]

Implementieren Sie eine Struktur, die die Farbwerte eines Pixels beschreibt:

- Der `struct` enthält Farbwerte R,G,B sowie einen Alpha-Wert, jeweils aus dem Bereich 0..255.
- Geben Sie die Größe Ihres `struct`s in bytes aus.

Aufgabe 'Earnil' [arrays]

- Legen Sie ein (globales) Array der Länge 3 von doubles an.
- Lesen Sie das erste und zweite Element von der Konsole ein und speichern Sie die Summe in dem dritten Element.
- Geben Sie alle Elemente in einer Funktion aus.
- Legen Sie ein weiteres Array der gleichen Größe an und kopieren Sie das erste in das zweite Array in einer Funktion.

Aufgabe 'Earnur' [struct, arrays]

Implementieren und Testen Sie eine Struktur, die ein Polynom beschreibt. Das Polynom ist durch seine (`double`) Koeffizienten eindeutig beschrieben. Nehmen Sie an, dass es zunächst nicht mehr als Grad 9 hat.
Programmieren Sie Funktionen, um

- ein Polynom auszuwerten (`eval`) und
- zu einem weiteren Polynom zu addieren (`add`).

Aufgabe 'Ecthelion' [Mathematik]

- Schreiben Sie ein Programm, welches die angegebene Berechnungsvorschrift ausführt, bis sich der neue Wert nicht mehr "signifikant" vom vorhergehenden Wert unterscheidet (z.B. Abstand 1e-12). Die Vorschrift lautet:

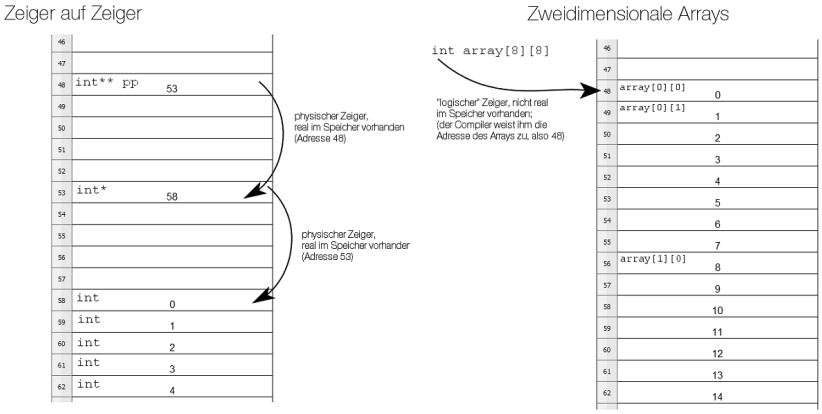
  $x_{n+1} = 1/2 \cos(x_n)$ Startwert $x_0 = 1.0$

- Wie lautet der Wert und wie viele Iterationen werden benötigt?

F - Zeiger

Idee

- Jede Variable besitzt einen Ort im Speicher, wo der Wert der Variablen abgespeichert ist. Dieser Ort ist die *Adresse* der Variablen.



(Bild: Uni Göttingen)

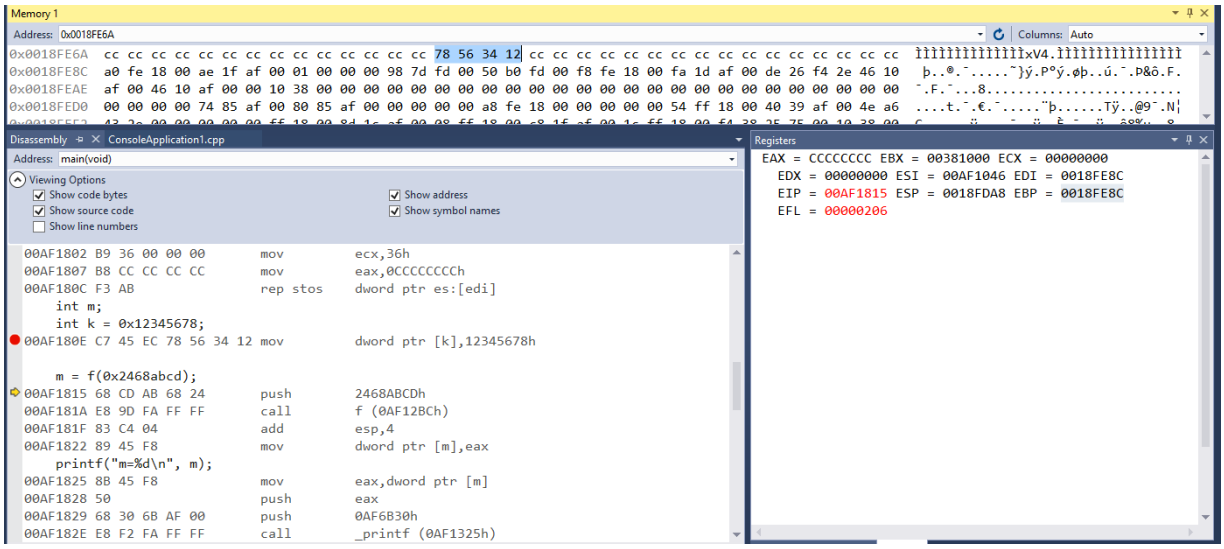
Computer		Programmers		
Address	Content	Name	Type	Value
90000000	00	sum	int (4 bytes)	000000FF(255_{10})
90000001	00			
90000002	00			
90000003	FF			
90000004	FF	age	short (2 bytes)	FFFF(-1_{10})
90000005	FF			
90000006	1F	averge	double (8 bytes)	1FFFFFFFFFFFFFF (4.45015E-308_{10})
90000007	FF			
90000008	FF			
90000009	FF			
9000000A	FF			
9000000B	FF			
9000000C	FF			
9000000D	FF			
9000000E	90	ptrSum	int* (4 bytes)	90000000
9000000F	00			
90000010	00			
90000011	00			

Note: All numbers in hexadecimal

(Bild: https://www3.ntu.edu.sg/home/ehchua/programming/cpp/cp4_-PointerReference.html)

- Die Adresse einer Variablen kann ebenfalls in einer Variablen gespeichert werden, das ist dann ein *Zeiger* bzw. *pointer*. Hier im Beispiel enthält `ptrSum` die Adresse von `sum`.

In nachfolgendem Bild sieht man einen Speicherausschnitt vom Stack (ESP bzw. EBP), auf dem gerade die lokale Variable k abgelegt wurde.



Weiter sieht man, dass der Aufruf einer Funktion nichts anderes bedeutet, als die Parameter und den derzeitigen Ausführungsort (push 0x2468abcd, EIP durch call) auf den Stack zu legen und dorthin zu springen. Beim return wird diese Adresse dann wieder vom Stack geholt, dieser um die Parameter bereinigt und bei der geholten Adresse vom Stack weitergemacht. Das ist dann normalerweise der Befehl nach dem call. Hier ist der Rückgabewert im Register eax und wird in der lokalen Variablen m abgelegt.

F – Übungen

Aufgabe 'Faramir' [Zeiger]

Zeiger definieren und benutzen:

- Legen Sie eine `int` und eine `float` Variable an und initialisieren Sie sie mit beliebigen Werten.
- Legen Sie zwei Zeiger an, die jeweils auf diese Variablen zeigen.
- Verändern Sie die Werte der Variablen mit Hilfe der Zeiger und
- geben Sie jeweils die Variable und den Wert, auf den der zugehörige Zeiger zeigt, aus.

Aufgabe 'Findegil' [Zeiger, Referenzen]

Aufruf einer Funktion mit Zeigern und Referenzen:

- Schreiben Sie eine Funktion `Swap`, die zwei übergebene double-Zahlen tauscht.
- Schreiben Sie eine weitere Funktion `Swap`, die zwei übergebene double-Zeiger tauscht.
- Implementieren Sie die Funktionen einmal über Zeiger und einmal über Referenzen.

Aufgabe 'Finduilas' [Zeiger]

Zeiger auf Zeiger:

- Definieren Sie zwei Integer `n1,n2` mit beliebigen festen Werten, einen nicht initialisierten Zeiger `p` auf `int` und einen nicht initialisierten Zeiger `pp` auf einen `int`-Zeiger.
- Initialisieren Sie die Zeiger so, dass der Integer `n1` über `p` modifiziert werden kann und `n2` über `pp` und `p`.

Aufgabe 'Forlong' [Zeiger]

- Versuchen Sie, durch Verwendung von Zeigern einzelne Bytes von `int`s zu manipulieren.
- Versuchen Sie, über Zeiger-Manipulationen lokale Variablen gezielt auf dem Stack zu verändern, ohne diese direkt zu adressieren.
  Tipp: Ermitteln Sie zunächst die Adresse einer lokalen Variablen. Die anderen liegen "in der Nähe".
- Versuchen Sie, durch "bösartige" Manipulation des Stacks das Programm zum Absturz zu bringen.

Aufgabe 'Fingolfin' [Felder, Zeiger]

Mit Feldern und Zeigern arbeiten:

- Legen Sie einen C-String mit einem beliebigen Text an:
  `char* str = "hallo";`
- Legen Sie einen Zeiger auf `char` an und laufen Sie mit diesem durch das Feld `str` (einschliesslich '`\0`') um den jeweiligen Charakter, auf den der Zeiger zeigt, auszugeben.
- Geben Sie den jeweils aktuellen Charakter einmal als Charakter und einmal als ASCII Wert aus.
- Geben Sie zusätzlich auch den Wert des Zeigers aus (die Adresse).

Aufgabe 'Firiel' [Zeiger]

- Definieren Sie vier Worte *"Dies", "ist", "ein", "Satz"* in einem Feld mit vier Zeigern.
- Übergeben Sie dieses Zeigerfeld einer Funktion, um dort die Worte in umgekehrter Reihenfolge auszugeben. Drehen Sie die dazu die Reihenfolge der Worte in dem Feld um.

Aufgabe 'Folca' [Zeiger]

- Reservieren Sie dynamisch Speicher für die vier Worte aus 'Firiel', einmal mit malloc und free, und einmal mit new und delete.

Aufgabe 'Fastred' [Zeiger, Konstanten]

Zeiger auf Konstanten, konstante Zeiger:

- Definieren Sie einen `float`, einen Zeiger auf diesen `float`, eine Konstante vom Typ `float`, einen Zeiger auf diese Konstante und einen konstanten Zeiger auf `float`.
- Versuchen Sie die `float` Variablen über diese Zeiger zu ändern.
- Versuche Sie, den Zeigern einen anderen Wert zuzuweisen.

Was funktioniert und was nicht?

Aufgabe 'Folcwine' [function pointer]

Implementieren Sie die Tangententrapezformel (oder Mittelpunktsregel) zur numerischen Berechnung eines Integrals. Übergeben Sie der Tangententrapezformel die Funktion, über die integriert wird, sowie die Grenzen. Nutzen Sie Typedefs und wählen Sie ein schönes Beispiel.

Aufgabe 'Freawine' [function pointer]

Schreiben Sie eine Funktion `Approx`, die einen Startwert `x0`, eine zu iterierende Funktion `f`, sowie ein `Eps` als Parameter erhält:

- `Approx` berechnet `x=f(x)`, mit `x=x0` zu Anfang, solange, bis der neue `x`-Wert sich vom vorhergehenden um weniger als `Eps` unterscheidet. Geben Sie den letzten berechneten Wert zurück.
- Nutzen Sie `Approx` für das Heron-Verfahren (Wurzel-Berechnung, siehe Wikipedia; zunächst für feste Werte a).

Aufgabe 'Frealaf' [function pointer]

Geben Sie in Ihrem Programm eine Funktion an, die bei Beendigung aufgerufen wird. Lesen Sie nach unter `atexit`. Lassen Sie Ihr Programm abstürzen/terminieren - wird Ihre Funktion aufgerufen?

Aufgabe 'Fuinur' [Felder, Zeiger, malloc, free, new, delete]

- Legen Sie ein Feld von `int`s der Größe 10 auf dem Stack an.
- Initialisieren Sie dieses Feld mit der 3-er Reihe (1*3, 2*3, ...).
- Legen Sie einen Zeiger auf `int` an und laufen Sie mit diesem durch das Feld um die jeweilige Zahl, auf die der Zeiger zeigt, auszugeben.
- Geben Sie zusätzlich auch den Wert des Zeigers aus.
- Legen Sie das Feld aus Aufgabe nun dynamisch an (`malloc, new`) und
- geben Sie es am Ende auch wieder frei (`free, delete`).

Aufgabe 'Falathar' [Files, Zeiger]

Dateien lesen und schreiben:

- Lesen Sie eine Textdatei Ihrer Wahl zeilenweise in eine geeignete Datenstruktur ein und ermitteln Sie die Häufigkeiten aller vorkommenden Buchstaben/Zeichen.
- Schreiben Sie eine Textdatei mit diesen Häufigkeiten in der Form
  'Zeichen' (ASCII) #Anzahl
  Also beispielsweise
  `'A' (65) #123`
  für: 123 mal das Zeichen '`A`' mit ASCII 65.
- Vergessen Sie nicht, beide Dateien wieder zu schliessen.

Aufgabe ‘Forweg’ [va_arg]

Funktionen mit beliebiger Anzahl von Argumenten:

- Schreiben Sie eine Funktion, die mit einer beliebigen Anzahl von `double`-Werten aufgerufen werden kann. Summieren Sie sie und geben das Ergebnis zurück.

F - Zeigercheck

Check

- Ich kann meinem Nachbarn erklären, was Zeiger (Pointer) sind.
- Ich kann einen Zeiger anlegen, initialisieren oder dereferenzieren - kurz: verwenden.
- Ich kann Funktionen schreiben, die Werte über Zeiger in Variablen der aufrufenden Funktion ablegt.
- Ich verstehe, was Zeiger mit Feldern zu haben.
- Ich weiß, was Zeigerarithmetik ist und was sie insbesondere mit Feldern zu tun hat.
- Ich kenne Funktionen, mit denen man “in C” Speicher vom System zugeteilt bekommt und wieder freigibt. Gleiches in “C++”.
- Ich kann Zeiger in verschiedene Zeigertypen wandeln und weiß, was ein typenloser Zeiger ist.
- Ich kann ebenfalls Zeiger auf Konstanten, aber auch konstante Zeiger auf Nicht-Konstanten, oder aber konstante Zeiger auf konstante Daten verwenden.
- Ich kann Funktionszeiger verwenden und meinem Nachbarn erklären, wozu man sie verwenden kann.
- Ich kann mit Zeigern auf Zeigern umgehen und habe eine Idee, wozu man sie braucht.
- Ich kenne Referenzen und weiß, wie man sie verwendet und was der Unterschied zu Zeigern ist.
- Ich kenne C-Strings.

G - Klassen und Objekte

Idee

- Wie in Java auch gibt es in C++ Klassen und Instanzen dieser Klassen ➜ Objekte.
- Damit sich ein Objekt in einem zulässigen und nicht zufälligen Zustand befindet, wird beim Erzeugen der sog. Konstruktor (ctor) aufgerufen. Diese Funktion ist Teil der Klasse und heisst so wie die Klasse.
- Einer der wohl häufigsten Fehler ist der, ein neues lokales Objekt über *new* anlegen zu wollen. Instanzen beispielsweise der Klasse C werden einfach so angelegt

  ```
  if (/* Bedingung */) {
      C c; // c existiert!
      // …
  }
  ```

  Das Object c wurde auf dem Stack angelegt und der ctor wurde aufgerufen!

  Das Erzeugen über new ist auch möglich, liefert aber einen Zeiger auf ein neu angelegtes Objekt auf dem Heap.
- Da es, im Gegensatz zu Java, in C++ keine Garbage Collection gibt, spielt hier der sog. Destruktor (dtor) eine ebenso wichtige Rolle wie der ctor. Der dtor wird aufgerufen, wenn das Objekt den Gültigkeitsbereich verlässt, also z.B. am Ende einer Funktion oder eines Scopes, wenn es ein lokales Objekt ist (s.o.).
- *Rule of three* [Stroustrup]: If a class defines one of the following it should probably explicitly define all three: destructor, copy constructor, copy assignment operator.

Check I

Aufgabe 1 — 2+2 P.

Bekanntlich gibt es für zwei ganze, positive Zahlen n und m Zahlen a und b, die die Gleichung n=a*m+b erfüllen – nämlich gerade a=n div m und b=n mod m. Schreiben Sie zwei Funktionen myDiv(int n, int m, ? a, ? b), die jeweils a und b zu gegebenen n und m berechnen. Geben Sie dabei a und b mittels

a) Referenzen, bzw.
b) Zeiger

zurück. Kommentieren Sie die entsprechenden Stellen in main ein und rufen Sie jeweils myDiv entsprechend auf.

Aufgabe 2 — 4 P.

Die Funktion

```
int myCount(const char* msg, char c) { …
```

bekommt einen Zeiger auf ein char-Feld (C-String) übergeben und soll die Anzahl enthaltener Characters c ermitteln und zurückgeben.

a) Implementieren Sie den Suchalgorithmus und nutzen Sie **nicht** den Indexoperator msg[i] für den Zugriff auf ein Zeichen, sondern nur Zeigerarithmetik und Dereferenzierungen.

Aufgabe 3 — 2+2+4+4

In A3 ist eine leere Klasse Tuple vorbereitet. Diese Klasse soll ein Tuple von zwei ganzen Zahlen repräsentieren, also z.B. (1,2). Ergänzen Sie sie gemäß folgender Anforderungen und kommentieren Sie die entsprechenden Stellen in main ein.

a) Die Klasse besitzt zwei private Member a,b vom typ int.
b) Sie besitzt ebenfalls einen Konstruktor, der zwei ganze Zahlen übergeben bekommt und die Member a und b initialisiert.
c) Die Klasse verfügt weiter über einen copy-Konstruktor, der allerdings zu den Argumenten 1 bzw. 2 hinzuzählt (siehe Code).
d) Es existieren zwei Getter get_a und get_b, die jeweils den Wert von a bzw. b zurück liefern.

Ihre Punktzahl: Ihre Note:

(Gesamt 20 Punkte. <10:5 10…12:4 13…15:3 16…18:2 19,20:1)

G - Übungen

Aufgabe 'Galador' [class, constructor, operator<<]

Schreiben Sie eine Klasse `Bruch` mit Zähler und Nenner:

- Implementieren Sie einen Default-Constructor.
- Implementieren Sie einen Constructor, der einen Zähler und einen Nenner übergeben bekommt. Benutzen Sie die Form der Initialisierung mit ':'.
- Implementieren Sie einen Copy-Constructor.
- Implementieren Sie einen Destruktor.
  Implementieren Sie einen `friend operator<<` zur Ausgabe.
- Implementieren Sie Getter- und Setter für Zähler und Nenner.

Aufgabe 'Golasgil' [class, constructor, operator<<]

Schreiben Sie eine Klasse `Kontakt`, die einen Namen und ein Alter enthält:

- Implementieren Sie einen Default-Constructor.
- Implementieren Sie einen Constructor, der einen Namen und ein Alter übergeben bekommt. Benutzen Sie die Form der Initialisierung mit ':'.
- Implementieren Sie einen Copy-Constructor.
- Implementieren Sie einen Destruktor.
  Implementieren Sie einen `friend operator<<` zur Ausgabe.
- Implementieren Sie Getter- und Setter für Namen und Alter.

Aufgabe 'Gamling ' [class, constructor, copy constructor, operator=, operator<<]

Schreiben Sie eine Klasse `Punkt` mit zwei ganzzahligen Koordinaten (X, Y):

- Implementieren Sie einen Default-Constructor.
- Implementieren Sie einen Copy-Constructor.
- Implementieren Sie einen Constructor, der zwei Werte (X, Y) zur Initialisierung übergeben bekommt. Benutzen Sie die Init. mit ':'.
- Implementieren Sie einen Constructor, der einen struct zur Initialisierung übergeben bekommt. Benutzen Sie die Member-Init. mit ':'.
- Implementieren Sie einen Destructor.
- Implementieren Sie einen eigenen `operator=`.
- Implementieren Sie einen `friend operator<<` zur Ausgabe.
- Implementieren Sie Getter- und Setter-Methoden für X und Y.
- Implementieren Sie eine Member-Funktion für die Max-Norm.

Aufgabe 'Garulf' [cout, cin, vector, push_back, size]

Speichern Sie eingegebene ganze Zahlen:

- Lesen Sie in Ihrem C++-Programm ganze Zahlen via `cin` ein.
- Geben Sie die Eingabe zur Sicherheit direkt noch einmal über `cout` aus.
- Speichern Sie jede Zahl in einem Vektor der Klasse `vector<int>` mittels `push_back` (`#include <vector>`).
- Wenn -1 eingegeben wurde, beenden Sie die Eingabe und geben alle im Vektor gespeicherten Zahlen aus (size gibt die Größe, [] den Zugriff auf die Elemente).

Aufgabe 'Gram' [vector]

Sehen Sie sich die Klasse `vector` in einer Referenz Ihrer Wahl an.

- Initialisieren Sie einen Vektor obigen Typs mit einer Menge von Wörtern, z.B. direkt mit { … } oder via Einlesen.
- Füllen Sie einen weiteren Vektor rückwärts mit den Wörtern des ersten Vektors (nicht die Buchstaben rumdrehen, nur die Reihenfolge der Wörter).
- Geben Sie alle Wörter des zweiten Vektors in einer Schleife aus.
- Löschen Sie den ersten Vektor.

Aufgabe 'Goldwine' – Hausaufgabe! [vector]

- Implementieren Sie einen Stack für `int`s und für Strings mit den entsprechenden Funktionen `pop` und `push`. Nutzen Sie als interne Speicherstruktur vector.

Aufgabe 'Gandalf' – Hausaufgabe! [Rule-Of-Three, new, delete]

Erweitern Sie Aufgabe 'Goldwine' und verwenden Sie als interne Speicherstruktur ein dynamisches Feld von int (mit new anlegen). Achten Sie auf die Rule-Of-Three.

G - Klassen und Vererbung

Idee (Sehr-Kurz-Version)

- Im Unterschied zu Java sind Methoden einer Klasse erst einmal *nicht* virtuell.
- Wenn eine Methode einmal als virtuell deklariert ist, dann bleibt sie es auch (bei gleicher Signatur!).
- Durch den Zusatz "=0" werden Methoden *pure virtual* und die Klasse *abstrakt*. Das bedeutet, dass keine Instanzen von diesen (und abgeleiteten) Klassen erzeugt werden können, bis nicht alle pure virtual Methoden auch definiert sind (die Klasse demnach nicht mehr abstrakt ist).
- Es gibt keine expliziten Interfaces, da es aufgrund der möglichen Mehrfachvererbung keine Notwendigkeit dafür gibt. Die Idee von Interfaces wird durch abstrakte Klassen mit pure virtual Methoden realisiert. Darüber hinaus kann natürlich eine abstrakte Klasse dennoch Funktionalität enthalten.
- Bei der Mehrfachvererbung kann entschieden werden, ob man ein Diamantmuster mit einer gemeinsamen Basisklasse haben möchte oder mehrfache Implementierungen der Basisklasse. Auch hier wird das Schlüsselwort virtual verwendet. Man beachte, dass im Fall *einer* Basisklasse (Diamant) es auch nur einen Konstruktoraufruf gibt.

G - Übungen Fortsetzung

Aufgabe 'Gálmód' [virtual]

Erklären Sie Ihrem Nachbarn/Ihrer Nachbarin, worin sich ein Objekt einer Klasse mit virtuellen Funktionen von dem einer Klasse ohne unterscheidet.

Aufgabe 'Gríma' [class, virtual, vector, set]

Schreiben Sie eine Kontakt-Verwaltung:

- Schreiben Sie eine Klasse (Geschäfts)Kontakt, die einen Namen (string) und ein Alter (int) enthält.
- Bereiten Sie eine virtuelle Funktion MonatsReport vor.
- Leiten Sie jeweils von Kontakt eine Klasse Kunde ab, die einen Kontostand (int) enthält, sowie eine Klasse Lieferant, die eine Bankverbindung (string) enthält.
- Implementieren Sie jeweils für Kunde und Lieferant die Funktion MonatsReport (mit einer einfachen Ausgabe).
- Erstellen Sie einen vector mit Zeigern auf Kontakte und füllen diesen Vektor mit einigen (zwei) fiktiven Kunden und Lieferanten.
- Rufen Sie nun die Funktion MonatsReport für alle Ihre Kontakte in dem Vektor auf.

Aufgabe 'Guthláf' [abstract class, virtual]

Ändern Sie 'Grima' so ab, dass Kontakt ein "Interface" IReport mit der Funktion MonatsReport erbt und die Klassen Kunde und Lieferant dieses implementieren.

Aufgabe 'Gilraen' [virtual]

Schreiben Sie eine Fahrzeug-Verwaltung:

- Schreiben Sie eine Klasse BefoerderungsMittel, die eine Anzahl Sitzplätze enthält sowie eine Klasse Fahrzeug, die eine Eigenschaft TopSpeed enthält.
- Leiten Sie eine Klasse Auto von BefoerderungsMittel und von Fahrzeug ab, die eine Anzahl Raeder enthält.
- Leiten Sie eine Klasse Boot von BefoerderungsMittel und von Fahrzeug ab, die eine Anzahl Schiffsschrauben enthält.
- Leiten Sie eine Klasse Amphibie von Auto und von Boot ab, die jedoch nur eine Anzahl Sitzplaetze (gemeinsame Basisklasse BefoerderungsMittel) aber zwei TopSpeeds (zwei Basisklassen Fahrzeug) enthält.
- Legen Sie Amphibien-Objekte an und testen Sie Ihren Code.

Check II

Aufgabe 1

Ergänzen Sie im Beispiel 'G15_line_example' die entsprechenden Stellen, so dass folgende Anforderungen erfüllt sind:

1. `Line` soll von einer abstrakten Klasse `IDrawable` mit einer virtuellen Methode `Draw` erben, die keine Parameter besitzt und nichts zurückgibt.
2. Die Variablen x1…y2 sollen direkt mit dem Standardwert für ints initialisiert werden (also nicht im ctor, sondern direkt bei der Deklaration).
3. Es gibt einen Ausgabeoperator `op<<`, so dass die auskommentierte Ausgabe in `Draw` funktioniert. Die Ausgabe erzeugt `"(2,3;5,6)"`, wenn die Werte gesetzt wurden.
4. Es gibt zwei 'Setter' `set_P1` und `set_P2`, die die Punkte P1 (also x1 und y1) und P2 (also x2 und y2) setzen und so eine Linie definieren. Die Parameter sollen weiterhin x1,y1 bzw. x2,y2 heissen.

G - Klassen und Operatoren

Idee

- In C++ können viele, aber nicht alle, Operatoren für eigene Klassen überschrieben werden. Das bedeutet, dass beispielsweise für Objekte eines klassischen Vektortyps Ausdrücke der Form

  v1 = v2+v3*4; // elementweise Addition

  möglich sind, wenn Operatoren, hier der +Operator und *Operator, für diese Objekte deklariert wurde.
- Eine Übersicht aller möglichen Operatoren findet sich in [http://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B]
- Der Compiler macht aus Operatoren im Prinzip Aufrufe spezieller Funktionen, Beispiel

  v1 = v2+v3*4;

  entspricht

  v1.operator=(operator+(v2,operator*(v3,4)));

  Hier kann man sich "operator=" oder "operator*" als Funktionsnamen vorstellen, dann ist es ein klassischer Aufruf einer Funktion mit entsprechenden Argumenten.
- Es gibt ein paar Regeln zum guten Umgang mit eigenen Operatoren, die in [http://stackoverflow.com/questions/4421706/operator-overloading] ganz gut zusammengefasst sind, u.a.:
  - Operatoren für die eigene Klasse sollten intuitiv sein und keine eigene "wilde" Syntax realisieren.
  - Wenn symmetrische Operatoren definiert werden, dann auch alle Möglichkeiten. Beispiel Multiplikation Vektor mit Skalar

    v1 = v2*3; aber auch v1 = 3*v2;
  - Wenn möglich sollte die friend-Variante der Member-Variante vorgezogen werden.
  - Es sollte nur eine Implementierung der eigentlichen Operation geben, d.h. wenn beispielsweise += definiert ist und den Algorithmus für die Addition enthält, dann ist + über += zu realisieren, ebenso wie Symmetrien ausgenutzt werden sollten.
- Die Ausführungsreihenfolge der Operatoren kann nicht geändert werden, d.h. * wird immer vor + ausgeführt.

G - Übungen Fortsetzung

Aufgabe 'Galadriel' [operator]

Schreiben oder erweitern Sie eine Punktklasse (siehe 'Gamling') um

- Operatoren +=, -=, +, - und skalare Multiplikation.

Aufgabe 'Galathil' [operator]

Schreiben oder erweitern Sie eine Klasse Bruch (siehe 'Galador') mit ganzzahligem Nenner und Zähler. Implementieren Sie

- Operatoren + - * /

Aufgabe 'Galdor' [operator]

Schreiben oder erweitern Sie eine Klasse Complex mit

- sinnvollen Operatoren Ihrer Wahl.

Aufgabe 'Gilfanon' [operator]

Erweitern Sie die Polynome (siehe 'Earnur') zu einer Klasse mit

- sinnvollen Operatoren Ihrer Wahl, u.a. einem ()-Operator.

Check III – test_set

Aufgabe 1

In dieser Aufgabe geht es um einen Datencontainer für ganze Zahlen, genauer eine Menge set, die für ein Experiment mit (paarweise verschiedenen) Zufallszahlen benötigt wird. Die interne Repräsentation ist beliebig, nur die Benutzung der std-Klassen map, set, unordered_map und unordered_set ist aus Lizenz-rechtlichen Gründen nicht erlaubt. Die Anzahl maximal benötigter Elemente ist im Vorhinein bekannt, so dass der Datencontainer bei der Erzeugung entsprechend dimensioniert werden kann. Realisieren Sie folgende Anforderungen:

(a) Es gibt eine Klasse set mit einem Konstruktor

```
set(unsigned int max_elements)
```

der ein Objekt dieser Klasse initialisiert. Beispiel: Der Code

```
set s1(5);
```

erzeugt eine leere Menge s1 der Klasse set mit Platz für 5 Zahlen.

(b) Es gibt einen weiteren zu definierenden Konstruktor, dem zusätzlich eine beliebige Menge von Zahlen übergeben werden kann.
Beispiel: Bei folgender Deklaration des Mengenobjekts s2 der Klasse set ist dieses schon mit den Elementen 1 und 2 initialisiert und bietet insgesamt Platz für 7 Zahlen.

```
set s2(7,{1,2});
```

(c) Diese Funktion testet, ob die Zahl n in der Menge enthalten ist:

```
bool contains(unsigned int n)
```

(d) Diese Funktion fügt die Zahl n ein, falls n noch nicht enthalten ist:

```
void set_value(unsigned int n)
```

(e) Werden mehr Zahlen in die Menge eingefügt als ursprünglich reservierter Platz vorhanden ist, wird eine Ausnahme geworfen.

(f) Diese Funktion löscht die Zahl n aus der Menge:

```
void remove_value(unsigned int n)
```

(g) Die Klasse set enthält einen binären +-Operator, der zwei Mengen zu einer vereinigt. Doppelte Zahlen kommen in der Vereinigungsmenge nur einmal vor. Die maximale Anzahl von Elementen ergibt sich aus der Summe der maximalen Anzahlen beider Operatoren.

(h) Die Klasse set enthält einen Ausgabeoperator, so dass die Ausgabe in einen ostream funktioniert. Beispiel: Das Ausgabeformat bei einer Menge mit den Zahlen 1,2,3 sieht wie folgt aus: “{ 1 2 3 }”.

Check III – Fragen

Aufgabe 2

- Ich kann Klassen definieren und Objekte anlegen.
- Ich kenne Konstruktoren, Destruktoren, Kopierkonstruktoren, Operatoren, konstante und nicht konstante Memberfunktionen.
- Die Rule-of-three ist mir ein Begriff.
- Ich kann Objekte auch dynamisch anlegen (Heap) und wieder freigeben.
- Operatoren in Klassen sind kein Problem, zumindest kenne ich die gebräuchlichsten wie +=, +, =, <<, ==.
- Ich weiß, wie man in C++ vererbt, was virtuelle Methoden sind, dass es Mehrfachvererbung gibt und welche Probleme bei gemeinsamen Basisklassen auftauchen können (Diamant).
- Ich kann den Begriff der virtuellen Tabelle erklären und kenne die Kosten für virtuelle Funktionen sowohl hinsichtlich Speicherverbrauch als auch Laufzeit.
- Abstrakte Klassen und Methoden (pure virtual) sind mir bekannt und ich kann ebenfalls erklären, warum es keine Interfaces gibt.

G - Übungen Fortsetzung

Aufgabe 'Gwaihir' [initializer-list]

Schreiben Sie eine Klasse (FIFO)Queue für ganze Zahlen. Entwerfen und implementieren Sie:

- Einen Konstruktor, den Sie mit einer initializer-list aufrufen können, etwa so: Queue Q = { 5, 6, 7 }
- Funktionen zum Hinzufügen und zum Entfernen von Einträgen. Nutzen Sie Operatoren, wenn Sie welche für geeignet halten.
- Eine Ausgabe, die alle Elemente der Queue ausgibt.
- Vergleichen Sie Ihre Klasse mit dem Container std::queue.

Aufgabe 'Grinnah' [iterator]

Ergänzen Sie Ihre Queue-Klasse aus 'Gwaihir' um einen Iterator, so dass Sie mit einer Schleife derart

```
for(auto q:Q)
    cout<<q<<endl;
```

Ihre Queue Q durchlaufen können.

Aufgabe 'Greif' [static, timing]

Fibonaccizahlen, Optimierung und Zeitmessung:

- Programmieren Sie die Berechnung der n'ten Fibonaccizahl und nutzen Sie z.B. statische Variablen in Ihrer Berechnung zur Beschleunigung bzw. zur Verringerung der rekursiven Aufrufe. Versuchen Sie besser zu sein als die Varianten in G27_fib_bench.

Aufgabe 'Gothmog' [static]

Schreiben Sie eine Funktion f zur Berechnung der Fakultät n!:

- Lesen Sie n wiederholt von der Tastatur ein und ermitteln dazu n! (Fakultät). Die Eingabe von n=-1 bricht das Programm ab.
- Ist zuvor schon einmal n! ermittelt worden, so nehmen Sie den Wert z.B. aus einem (statischen) Array oder einer anderen geeigneten Datenstruktur ohne ihn zu berechnen. Ist er noch nicht abgefragt worden, so berechnen Sie ihn und legen ihn natürlich auch in dieser Datenstruktur ab.
- Messen Sie Zeiten und vergleichen Sie verschiedene Lösungsvarianten.

G - Lambda-Expressions

Idee

- Lambda-Expressions/-Ausdrücke sind anonyme Funktionen, die im Code an der Stelle definiert werden, an der sie benötigt werden.
  Gute Beispiele hier sind Funktionen der Algorithmus-Bibliothek oder eine Sort-Funktion, der übergeben werden soll, nach welchem Kriterium sie sortieren soll.
- Lambda-Ausdrücke können z.B. in Funktionszeiger konvertiert werden, wenn die Signatur übereinstimmt.
- Es gibt sie in unterschiedlicher Syntax:

  ```
  [](double x) -> double {return x*x*x;}
  ```

  und

  ```
  [](double x){return x*x*x;},2.0)
  ```

  Dabei orientiert sich die erste Form an der mathematischen Schreibweise einer Funktion $x \rightarrow x^3$
- Die [] bzw. das was in den [] enthalten ist, gibt an, wie z.B. mit Variablen zu verfahren ist, die nicht in dem Lambda-Ausdruck definiert werden sondern ausserhalb existieren. Es gibt zwei Varianten, per Referenz [&] und per Kopie [=], dabei können auch einzelne Variablen angegeben werden [&m] bzw. [m] (siehe Code-Beispiele).

G - Übungen Fortsetzung

Aufgabe 'Gorbag' [lambda expression]

Schreiben Sie jeweils einen Lambda-Ausdruck, der

- drei double-Werte addiert und zurückgibt.
- testet, ob ein int-Wert in einem Intervall [a,b] liegt (dabei sind a und b lokale Variablen).
- keine Argumente hat, aber eine lokale int-Variable z auf -z setzt.

Und dann

- eine Funtion eval, die Sie mit einem Lambda-Ausdruck f und einem double-Wert x aufrufen können, und die f(x) zurückgibt.

Aufgabe 'Gungliont' [lambda expression]

Überlegen Sie sich, wie Sie ein Newtonverfahren (siehe Wikipedia) als

- Funktion implementieren würden, die Sie mit Lambda-Ausdrücken für f und f' aufrufen können.
- selber als Lambda-Ausdruck definieren, den Sie ebenfalls mit Lambda-Ausdrücken für f und f' aufrufen können.

H - Templates

Idee

- Templates sind die generischen Klassen in C++, allerdings häufig weit mächtiger.
- Eine der Hauptideen hinter generischen Klassen und damit auch hinter Templates ist die Vermeidung gleichen Codes nur mit anderen Datentypen. Beispiel: Eine Vektorklasse, die Integer speichert, funktioniert intern genauso und bietet die gleiche Funktionalität wie eine weitere Vektorklasse für Fliesskommazahlen. Folglich schreibt man die eigentliche Vektorfunktionalität für einen beliebigen Datentype – unser Template.
- Generische Parameter in C++ dürfen Datentypen, aber auch ganze Zahlen oder sogar Templates selber sein.
- Manchmal ist es notwendig, neben der allgemeinen Template-Klasse eine Spezialversion für bestimmte Datentypen (genauer Template-Parameter) anzubieten. Im Beispiel oben wäre das etwa ein Vektor für Bools, bei denen die Daten, also die Bits, intern gepackt gespeichert würden. Dafür gibt es sogenannte Template-Spezialisierungen.
- Im Gegensatz zu Java oder für Referenztypen in C# wird für jede verwendete Template-Parameter Kombination auch eine eigene Klasse bzw. dazugehöriger Code generiert. Also, ein vector<int> und ein vector<double> sind zwei verschiedene Klassen mit den jeweiligen Codeanteilen im ausführbaren Code. Das kann zu einer erhöhten Codegröße führen, wenn Templates “exzessiv” verwendet werden (z.B. beim Template-Meta-Programming).
- Funktionen oder auch Operatoren, die in einer Templateklasse definiert und verwendet werden, werden erst beim Gebrauch mit den entsprechenden, dann bekannten Template-Parametern, auch compiliert. Das bedeutet beispielsweise, man kann durchaus einen Ausdruck der Form x=t*t für Variablen x und t vom Typ Template-Parameter T verwenden und erst wenn der konkrete Typ T feststeht und Code etwa für T=int generiert wird, wird überprüft, ob der *-Operator für T definiert ist.

H - Übungen

Aufgabe 'Hador' [templates]

Schreiben Sie Ihre Punktklasse ('Gamling') zu einer Template-Klasse um, so dass sie prinzipiell mit unterschiedlichen Datentypen funktioniert.

Aufgabe 'Hallas' [templates]

Schreiben Sie Ihre Bruchklasse ('Galador') zu einer Template-Klasse um, so dass sie prinzipiell mit unterschiedlichen Datentypen funktioniert.

Aufgabe 'Herion' [templates]

Schreiben Sie Ihre Complexklasse ('Galdor') zu einer Template-Klasse um, so dass sie prinzipiell mit unterschiedlichen Datentypen funktioniert.

Aufgabe 'Hirgon' [templates]

Schreiben Sie Ihre Polynomklasse ('Gilfanon') zu einer Template-Klasse um, so dass sie prinzipiell mit unterschiedlichen Datentypen funktioniert.

Aufgabe 'Húrin' [templates]

Schreiben Sie Ihre Queue ('Grinnah') zu einer Template-Klasse um, so dass sie prinzipiell mit unterschiedlichen Datentypen funktioniert.

Aufgabe 'Hyarmendacil' [templates]

Schreiben Sie eine Template-Klasse, die zur Compile-Zeit B hoch N für zwei natürliche Zahlen B und N berechnet.

H - Übungen Fortsetzung

Aufgabe 'Haleth' [unique-ptr, shared_ptr]

Entwerfen Sie eine beliebige Klasse und füllen Sie (jeweils) einen std::vector mit

- Objekten Ihrer Klasse,
- mit Zeigern auf dynamisch angelegte Objekte Ihrer Klasse, bzw.
- unique- und shared-ptr auf diese.

Werden alle Elemente ordnungsgemäß zerstört oder gibt es Memory-Leaks?

Können Sie alle Elemente in einer Schleife ausgeben?

Aufgabe 'Helm Hammerhand' [my-smart-ptr]

Entwerfen Sie eine eigene smart-ptr Klasse, die sich wie unique- bzw. shared-ptr verhält.

I/J - Threads

Idee

- Threads sind eigene Ausführungseinheiten innerhalb des Prozesses (des Programms), sie laufen grundsätzlich erstmal parallel, wenn sie gestartet wurden. In einem System mit mehreren Kernen können sie, je nach Betriebssystem und Einstellungen, auch tatsächlich parallel laufen (und nicht nur simuliert).
- Es gibt immer einen Main-Thread. In diesem wird dann main ausgeführt.
- Threads werden mit einer Workerfunktion und Parametern gestartet. Sie müssen beendet, bzw. es muss auf sie gewartet werden, bevor der Main-Thread endet.
- Gemeinsame Ressourcen, etwa Variablen, müssen vor gleichzeitigem Zugriff geschützt werden. Das geschieht beispielsweise durch die Verwendung eines Mutex.

I/J - Übungen

Aufgabe 'Isildur' [threads]

Starten Sie Threads, von denen der erste 1 Sek., der zweite 2 Sek. und der dritte 3 Sek. schlafen. Der Hauptthread wartet auf alle drei.

Aufgabe 'Ioreth' [threads]

Summieren Sie die Zahlen 1..N parallel und thread-safe in einer globalen Summenvariablen auf.

Aufgabe 'Imrahil' [threads]

Schreiben Sie eine parallele, sichere Variante der summierten (bzw. zusammengesetzten) Sehnentrapezformel (siehe Code-Snippet bzw. Wikipedia→Trapezregel) bei vorgegebener Anzahl Teilintervalle.

Aufgabe 'Isilmo' [threads, condition variables]

Schreiben Sie eine Producer-Consumer-Variante unter Verwendung von condition variables. Genauer:

- Ein Thread, der Producer, generiert "Aufgaben", hier Zahlen, und schreibt sie thread sicher in eine queue<int>. Das führt der Thread solange aus, bis eine globale "stop" Variable gesetzt wird.
- Ein weiterer Thread, der Consumer, liest aus der queue diese "Aufgaben" aus (und bearbeitet sie normalerweise). Er wird von dem Producer mittels einer condition variable darüber informiert, dass eine neue Zahl verfügbar ist, damit er nicht die ganze Zeit über die queue fragen muss, ob sie noch Werte enthält.
- Der main Thread startet den Producer und den Consumer Thread, wartet 3s und setzt dann die "stop" Variable.

Beachten Sie, dass Sie den Consumer "aufwecken" müssen. Nutzen Sie dafür eine sogenannte "Poison Pill"…

Themen

Stichpunkte

- Compilieren, Linken
- class, ctor, dtor, rule of three
- struct vs class
- RAII
- Zeiger, Zeigerarithmetik, Referenzen, Funktionszeiger, smart ptr
- threads, pattern condition variables, mutex
- Vererbung, abstrakte Klassen, (pure) virtuelle Funktionen, Mehrfachvererbung
- Operatorüberladung
- templates
- initializer_list, Iteratoren
- const, auto, static, friend, typedef
- arrays, vector, map
- exceptions
- new, delete, malloc, free
- file io
- lambda expressions
- va_arg