5. Übung des Programmierpraktikums

Abgabetermin: Fr, 25. Mai 2007, 23:59 Uhr

Die Übungen sind grundsätzlich allein zu machen. Gruppenarbeit ist nicht erlaubt.
Abzugeben sind jeweils die sinnvoll dokumentierten Programmfiles (Files für Bsp. 5: `bsp_5.cpp`, `bsp_5_fkt.cpp`, `bsp_5_fkt.hpp`) indem Sie diese in Ihr Verzeichnis `math197:lv\haase\Abgabe\Name_Vorname` kopieren.

Achtung: Bitte bei diesem Übungszettel nur ein Quelltextfile pro Aufgabe abgeben, d.h., die Funktionsdefinitionen in das File des Hauptprogrammes schreiben. Die Aufpaltung eines Programmes in mehrere Files kommt später.

---

15. **(3 P)** Schreiben Sie die **rekursive** Funktion `fibo`, welche die zweistufige Rekursionsformel der Fibonacci-Zahlen implementiert

$$f(n) := f(n-1) + f(n-2) \qquad n = 2, \ldots$$

mit den Anfangsbedingungen $f(0) = 0$, $f(1) = 1$.
Es sind keine Felder/Arrays zum Speichern der Werte f_n nötig!!
Testdaten (n): 5, 30, 50

16. **(6 P)** Der größte gemeinsame Teiler (ggT) zweier ganzer Zahlen a und b kann recht einfach als rekursive Funktion geschrieben werden:

$$ggT(a,b) = \begin{cases} ggT(b,\, a) & \text{falls } a < b \\ b & \text{falls } b \text{ ein Teiler von } a \text{ ist} \\ ggT(b,\, a \bmod b) & \text{ansonsten.} \end{cases}$$

Nutzen Sie zur Dateneingabe die Kommandozeilenparameter des Hauptprogrammes, wie in §7.6 des Skriptes beschrieben.

Testen Sie Ihre rekursive Funktion `ggT` indem Sie die Testdaten über die Kommandozeile an das Hauptprogramm übergeben, z.B., `bsp_16.exe 24 36`
Testdaten ($ggT(a,b)$): $ggT(24,36)$, $ggT(12,5)$, $ggT(5236,5005)$

17. **(6 P)** Die Sinusfunktion kann als Reihe ausgedrückt werden:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - - + \cdots$$

Der in code_sinus.cpp gegebene Code liest m und x ein, und berechnet den Wert dieser Reihe bis zum m-ten Glied.

Schreiben Sie das gegebene Programm zu Ihrem Code *bsp_17.cpp* so um, daß

- keinerlei Alternativen mehr gebraucht werden,
- der j-Zählzyklus eingespart wird (also nur noch ein i-Zählzyklus übrigbleibt)
- und die `pow`-Funktion nicht mehr benutzt wird.

Die Befehle `break`, `continue`, `goto` sind hierbei (und generell) nicht erlaubt.

18. **(3 P)** Implementieren Sie eine Funktion `riemann`, zum (numerischen) Integrieren einer geg. Funktion $f(x)$ mit Hilfe der Riemann-Summen

$$\int_a^b f(x)dx = \sum_{j=1}^{n} f(a+jh) \cdot h$$

wobei das geg. Intervall von a bis b in (geg.) n gleichgroße Intervalle der Länge h unterteilt wird. Die zu integrierende Funktion ist ebenfalls eine INPUT-Größe der Parameterliste, siehe Beispielfunktion `Bisect3` in §7.8 der Vorlesung (auch in Bisect3.cpp).

Testdaten: sqrt() in $[1, 4]$, cos() in $[0, \pi/2]$, log() in $[1, \mathrm{e}]$

19. **(3 P)** Implementieren Sie eine Funktion `derivative`, die unter Verwendung einer gegebenen Schrittweite h, die numerische Ableitung der (über)gebenen Funktion $f(x)$) an einem vorgegebenen Punkt x berechnet. Verwenden Sie die Formel

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h}$$

Testdaten: sqrt() in $x = 4$, cos() in $x = \pi/6$, log() in $x = 1$,

20. **(4 P)** In dieser Aufgabe müssen Sie sich unbedingt **an die gegebenen Filebezeichnungen halten**, die da wären *bsp_20_fkt.hpp* für das Headerfile und *bsp_20_fkt.cpp* für das Sourcefile Ihrer zu schreibenden/geschriebenen Funktionen. Kopieren Sie Ihre Funktionen `derivative` und `riemann` aus den beiden vorigen Aufgaben in Header- und Sourcefile, und testen Sie deren korrekte Nutzung in einem kurzen Hauptprogramm.

21. **(6 P)** Schreiben Sie verbesserte Funktionen `riemann` und `derivative` welche das bestimmte Integral (bzw. den Ableitungswert) mit einer Genauigkeit von ϵ zurückgeben. Speichern Sie diese Funktionen ebenfalls in den Files *bsp_20_fkt.hpp*, *bsp_20_fkt.cpp*. Testen Sie deren korrekte Nutzung bei $\epsilon = 10^{-4}$ mit den Testdaten aus Aufg. 18 und 19 in Ihrem Hauptprogramm.
Achten Sie insbesondere bei der Differentiation darauf, daß die Schrittweite h nicht zu klein wird (Maschinen-Epsilon in §3.9 bzw. Reihe.cpp).

22. **(6 P)** Die Bestimmung der Nullstelle einer Funktion $f(x)$ ist sehr effizient berechenbar mit der Newton–Iteration[1]

$$x_{n+1} := x_n - f(x_n)/f'(x_n)$$

für einen geeigneten Startwert $x_0 \in [a, b]$ (die math. Voraussetzung, daß die Funktion $x_n - f(x_n)/f'(x_n)$ eine kontraktive Abbildung ist, muß in Ihrem Programm nicht getestet werden).
Nutzen Sie Ihre Funktionen aus den Aufgaben 19 und 20 und erweitern Sie die Files *bsp_20_fkt.hpp*, *bsp_20_fkt.cpp* um Ihre Funktion `newton_diff_approx()`. Testen Sie Ihren Nullstellenloeser (und überprüfen Sie die Lösung) im Hauptprogramm mit den folgenden Funktionen:

Testdaten: $f(x) := \sin(x) - x/2$
$g(x) := -(x - 1.234567)(x + 0.987654)$
$h(x) := 3.0 - \mathrm{e}^x$
$z(x) := \cos(x) - \mathrm{e}^{x \ln |x|}$

23. **(6 P)** Ein Betrag zwischen 0.01 und 99.99 Euro soll in Münzen zu 1,2 Euro und zu 1,2,5,10,20,50 Cent herausgegeben werden. Schreiben sie eine Funktion `stueckelung` welche den übergegebenen Betrag (INPUT) durch die kleinstmögliche Anzahl von Münzen ausdrückt (OUTPUT).

Nutzen Sie Ihre Funktion `stueckelung`, um eine weitere Funktion zu schreiben, welche Ihnen für einen gegebenen Preis und die erhaltene Geldsumme des Kunden das Gewicht des Wechselgeldes berechnet.

Testdaten (Preis, Kunde): (23.22 , 50.00), (17.83 , 20.33), (43.21 , 50.01)

24. **(6 P)** Ein Polynom $p_n(x) = \sum_{k=0}^{n} a_k x^k$ vom Grad n ist durch die Koeffizienten a_k, $k = 0, \ldots, n$ bestimmt und wird im Punkt x ausgewertet.
Implementieren Sie eine Strukur, welche den Polynomgrad und die Koeffizienten speichert. Schreiben Sie eine Funktion, welche die Koeffizienten des Polynoms vom File einliest (Format: n a_0 $\ldots$ a_n) und auch den Speicherplatz hierfür reserviert. Implementieren Sie das Horner-Schema zur Auswertung des Polynoms, wie im Bsp.:

$$p_5(x) = (\{[(a_5 \cdot x + a_4) \cdot x + a_3] \cdot x + a_2\} \cdot x + a_1) \cdot x + a_0$$

und geben Sie die drei Polynomwerte $p_n(x)$ für die Testdaten aus.

Testdaten (Filename, x): (data_24_a.txt, 3.78), (data_24_b.txt, -1.0087), (data_24_c.txt, -0.945),

[1] `http://de.wikipedia.org/wiki/Newton_Iteration`

25. **(10 P)** Simulieren Sie die Spiele eines Volleyballturniers an dem n Mannschaften teilnehmen, welche alle genau einmal gegeneinander spielen (Sie müssen keinen Spielplan aufstellen). Geben Sie die Ergebnisse der einzelnen Spiele (Sätze, Punkte im Satz) aus und präsentieren Sie zum Schluß die gereihte Endtabelle Ihrer Simulation.

    Einige Hinweise:

    - Zählregeln[2]: Drei Gewinnsätze sind zum Sieg erforderlich. Ein Satz zählt bis zu 25 Punkten (5. Satz bis 15 Punkte), wenn dann eine der Mannschaften mindestens 2 Punkte zurückliegt (25:23). Kommt es zum 24:24, muß so lange weitergespielt werden, bis eine Mannschaft 2 Punkte Vorsprung erreicht hat. Die Zahl der Sätze hängt vom Spielverlauf ab. Sie beträgt bei
        - 3 Siegsätzen einer Mannschaft in Folge (3:0) 3 Sätze,
        - 3 Siegsätzen der einen, 1 Siegsatz der anderen Mannschaft (3:1) 4 Sätze,
        - 3 Siegsätzen der einen, 2 Siegsätzen der anderen Mannschaft (3:2) 5 Sätze.

      Ein Spiel kann also höchstens 5 Sätze haben.
    - Nutzen Sie zur Simulation der Zufallszahlen die Funktion `rand()` welche eine ganze Zahl zwischen 0 und $RAND\_MAX$ liefert. Falls erforderlich, würde ich zur Umrechnung in das Intervall $[0, 1]$ den Ausdruck `(rand()+0.0)/RAND_MAX` empfehlen.

      Um bei jedem Programmstart immer wieder andere Zufallszahlenfolgen zu produzieren, muß der Zufallszahlengenerator mit einem (möglichst zufälligen) Wert initialisiert werden. Dies ist z.B. via `srand ( time(NULL) );` oder `srand ( time(NULL) %(n*n) );` möglich. Dies erfordert das Inkludieren des Headerfiles *ctime* .
    - Erstellen Sie zuerst eine Tabelle mit den ungeordneten Punkten etc. der einzelnen Mannschaften. Zum Sortieren können Sie, z.B., den Bubblesort-Algorithmus[3] verwenden. Es gibt bessere Sortieralgorithmen, aber Bubblesort ist sehr leicht zu programmieren.

    Testdaten (n): (4), (10)

[2] `http://www.smash-hamburg.de/regeln/regeln-hv.htm`
[3] `http://en.wikipedia.org/wiki/Bubble_sort`