BERGISCHE UNIVERSITÄT
GESAMTHOCHSCHULE WUPPERTAL
GAUSS-STRASSE 20
42097 WUPPERTAL
(Korrespondenzanschrift)
42119 WUPPERTAL
(Lieferanschrift)
TELEX 8 592 262 bughw
TELEFAX (0202) 439-2901
TELEFON (0202) 439-1

Fachbereich 7

MATHEMATIK

Prof. Dr. Hans-Jürgen Buhl
Praktische Informatik / Numerik

e-mail: Juergen.Buhl@math.uni-wuppertal.de

Einführung in die Informatik und Programmierung (Informatik I)

WS2000/2001 – Übungsblatt 11

17. Januar 2001
Bearbeitungstermin: 4. KW

Aufgabe 1. *Primzahlsuche: Teilerkandidaten, 4 Punkte*
Anstatt alle ungeraden positiven Zahlen außer 1 als mögliche Teilerkandidaten bei der Primzahlsuche einzusetzen (vergleiche Aufgabe 1 und 2 / Übungsblatt 9), ist es möglich, aus der folgenden einfachen Aussage der Zahlentheorie Nutzen zu ziehen:

Alle Primzahlen außer 2, 3, 5 sind von der Form

$$30 \cdot k + a \text{ mit } k \in \mathbb{N}_0 \text{ und } a \in \{1, 7, 11, 13, 17, 19, 23, 29\}$$

(siehe etwa: Harald Scheid: Zahlentheorie, BI-Wissenschaftsverlag, 1994).

Ändern Sie Ihr Programm (Aufgabe 2 / Übungsblatt 9) so ab, dass nur noch diese neue Kandidatenmenge benutzt wird.

Aufgabe 2. *Schaltjahr, 4 Punkte*
Beweisen sie, dass die Mengen J_1, J_4, J_{100} und J_{400}

$$\begin{aligned} J_1 &= \mathbb{N} \setminus 4\mathbb{N} \\ J_4 &= 4\mathbb{N} \setminus 100\mathbb{N} \\ J_{100} &= 100\mathbb{N} \setminus 400\mathbb{N} \\ J_{400} &= 400\mathbb{N} \end{aligned}$$

eine disjunkte Zerlegung von $\mathbb{N}$ sind.
Schreiben sie dann Funktionen

```
bool istSchaltjahr(unsigned long Jahr)
```

und

```
unsigned long lastDayOfMonth(Monate m, unsigned long Jahr).
```

Definieren Sie dazu einen `enum`-Datentyp `Monate` mit Literalen `Jan, Feb,`

..., Dez derart, dass der Code für Januar die ganze Zahl 1, für Februar die ganze Zahl 2, ... ist.

Aufgabe 3. *Messung der CPU-Zeit eines Algorithmus, 4 Punkte*
Testen Sie das folgende Programm, um die CPU-Zeit einer Schleife zu messen:

```
////////////////////////////////
// Datei:   TakeClock.cc
// Version: 1.0
// Zweck:   Zeitmessung von Algorithmenteilen,
// Autor:   Hans-Juergen Buhl
// Datum:   17.09.1998
////////////////////////////////

#include <iostream>
#include <iomanip>
#include <cmath>

#include <ctime>
#include <unistd.h>

using namespace std;

int main()
{
  clock_t StartTime(clock());

  // ...
  // sleep(2);
  double f(sin(1.4567));
  for (int i = 0; i<1000000; i++) f = sin(f);
  // ...

  clock_t Time(clock()-StartTime);

  cout <<  "Der Algorithmus dauerte "
       <<  setiosflags(ios::scientific)
       <<  setw(12) << setprecision(4)
       <<  static_cast<double>(Time)  / CLOCKS_PER_SEC
       <<  " CPU-Sekunden (Auflösung = "
       <<  resetiosflags(ios::scientific)
       <<  1.0 / CLK_TCK
       <<  " Sekunden)" << endl;

  return 0;
}
```

Vergleichen Sie die Ergebniszeit bei mindestens 5-maligem Start von TakeClock. Interpretieren Sie!

Entkommentieren Sie die Anweisung `sleep(2)`, übersetzen Sie erneut, und vergleichen Sie die Ergebnisse eines mehrmaligen Starts dieser Version untereinander und mit denen des Ablaufs der vorherigen Version.
Erstellen Sie eine Tabelle, die die Rechenzeiten für die folgenden double-Operationen gegenüberstellt: `+, -, *, /, sqrt, sin, tan, atan`

Aufgabe 4. *Rechenzeitmessung, 2 Punkte*
Testen Sie die Rechenzeit für Ihr Programm von Aufgabe 1 und für Ihr Programm von Aufgabe 2 / Übungsblatt 9. Interpretieren Sie die Ergebnisse.

Aufgabe 5. *Objekte mit „lazy evaluation“ redundanter Attribute, 6 Punkte*
Testen Sie das folgende Programm:

```
////////////////////////////////
// Datei:   Function.cc
// Version: 1.0
// Zweck:   lazy evaluation
// Autor:   Hans-Juergen Buhl
// Datum:   16.12.1998
////////////////////////////////

#include <iostream>
#include <string>
#include <cmath>
#include <cassert>

using namespace std;

typedef double (*FKT)(double);

class Function{

  double start_x, end_x;
  int nr_intervals; // > 0
  FKT fkt;

  bool last_int_value_valid;
  double last_int_value; // valid iff (last_int_value_valid == true)

  bool last_slope_value_valid;
  double last_slope_value; // valid iff (last_slope_value_valid == true)

  // ...

public:

  Function(FKT f, double x_l = 0.0, double x_r = 2.0, int n = 8):
        start_x(x_l), end_x(x_r), fkt(f),
        last_int_value_valid(false),
        last_slope_value_valid(false)
```

```
        {
           assert(n>0);
           nr_intervals = n;
        };

  void set_nr_intervals(const int n)
  {
      assert(n>0);
      nr_intervals = n;
      last_int_value_valid = false;
      last_slope_value_valid = false;
  };

  int get_nr_intervals() const
  {
      return nr_intervals;
  };

  double get_integral_value()
  {
      if (!last_int_value_valid)
      {
               // ...
               // algorithm for numerical computation of integral value,
               // last_int_value = algorithm result;
               last_int_value_valid = true;
      };
      return last_int_value;
   };
};

double p(const double x)
{
      return ((15.0 * x + 4.0) * x + 3.5) * x + 2.25;
};

int main()
{
 Function sin0_2pi(sin, 0.0, 2.0*M_PI, 16);
 // test usage of Function methods

 Function poly(p);
 // further tests ...

 return 0;

}
```

Ergänzen Sie dazu die Methode `get_integral_value()` um einen „nume-

rischen Integrationsalgorithmus“ nach der Trapezregel. Planen Sie Tests für diese Implementierung und führen Sie diese durch Modifikation der `main()`-Funktion durch.
Ergänzen Sie analog eine Methode `get_slope_value()` und testen Sie (approximative größte Steigung der Funktion im Intervall von `start_x` bis `end_x`).
Modifizieren Sie die Methode `Function(FKT f, double x_l = 0.0, double x_r = 2.0, int n = 8)` und neue Methoden `set_start_x(const double x_l)` sowie `set_end_x(const double x_r)` so, daß *erzwungenermaßen* die Klasseninvariante „`start_x < end_x`“ gilt. Vergessen Sie nicht, einen entsprechenden Kommentar in der Zeile „`double start_x, end_x;`“ unterzubringen!
Diskutieren Sie Vor- und Nachteile der „lazy evaluation“.