# Aufgabenblatt 5

Prof. Dr. Oliver Bittel

Es soll eine Klassenschablone (class template) Data mit einer Sortierfunktionalität realisiert werden. Die Klasse gestattet das Einlesen von Datenelementen von einer Datei, deren Sortierung und deren Speicherung zurück auf die Datei. Die Klasse soll mit dem Typ der Datenelemente parameterisiert werden.

Folgende Methoden sollen durch die Klasse bereitgestellt werden:

- Data(filename) Konstruktor; öffnet die Datei filename und liest die in dieser Datei befindlichen Datenelemente in ein entsprechend grosses Feld ein. Annahme: am Anfang der Datei ist die Anzahl der Datenelemente angegeben.
- display() gibt alle Datenelemente am Bildschirm aus.
- sort() sortiert die Daten nach folgendem Verfahren:*Quicksort-Verfahren*, das sich selbst solange rekursiv aufruft, bis die Länge der zu sortierenden Teilfolge kleiner oder gleich einer Zahl M ist (z.B. M = 20) und diese Teilfolge dann mit dem Verfahren *Sortieren durch Einfügen* vollends sortiert. Hinweis: Verwenden Sie private Methoden, die die eigentliche Sortieraufgabe übernehmen und von sort() aufgerufen werden.
- ~Data() Destruktor; speichert u.a. die Datenelemente auf die Datei zurück.

Folgendes Beispiel verdeutlicht die Verwendung der Klasse Data:

```
void main()
{
      Data<double>    dbDat("db_daten.txt");
      Data<int>       intDat("int_daten.txt");
      Data<Complex>   complDat("compl_daten.txt");
      Data<Person>    persDat("pers_daten.txt");

      dbDat.display();// Datenelemente noch unsortiert
      dbDat.sort();
      dbDat.display();// Datenelemente sortiert


      intDat.sort();
      complDat.sort();
      persDat.sort();
}
```

## Teil 1: Programmentwicklung

Gehen Sie bei der Entwicklung der Klassenschablone Data in mehreren Schritten vor:

1. Realisieren Sie zuerst eine Klasse Data zur Sortierung von ganzen Zahlen ohne Verwendung des Schablonenmechanismus (d.h. ohne template). Die Klasse soll genau die vier oben beschriebenen Methoden anbieten. Testen Sie Ihre Klasse.

   Ändern Sie nun Ihre Klasse um in eine Klassenschablone. Testen Sie Ihre Schablone, in dem Sie den Typparameter (Parameter für den Typ der Datenelemente) mit Basisdatentypen wie int und double instantiieren, z.B

```
void main()
{
      Data<double>    dbDat("db_daten.txt");
      Data<int>       intDat("int_daten.txt");

      dbDat.sort();
      intDat.sort();
}
```

Hinweis: Bei Verwendung des GNU-C++-Compilers sollte Ihre data.h-Datei außer der Definition der Klassenschablone auch die Definitionen der Methodenschablonen enthalten. Die data.c-Datei entfällt dann.

2. Instantiieren Sie den Typparameter Ihrer Klassenschablone nun mit den selbst definierten Klassen Complex und Person:

```
void main()
{
      Data<Complex>     complDat("compl_daten.txt");
      Data<Person>      persDat("pers_daten.txt");

      complDat.sort();
      persDat.sort();
}
```

Die Personendaten, die jeweils aus Vorname, Familienname und Geburtsdatum bestehen, sollen nach dem Familiennamen lexikographisch sortiert werden.

Die komplexen Zahlen sollen nach ihren Beträgen sortiert werden; der Betrag einer komplexen Zahl x+y∗i ist $\sqrt{x^2+y^2}$.

Wichtig: Ihre Klasse Data setzt einen Vergleichs-, einen Eingabe- und einen Ausgabeoperator für die Datenelemente voraus. Sehen Sie daher für Ihre Klassen Complex und Person geeignete Operatoren vor. Definieren Sie diese als befreundete Funktionen (siehe Seite 3 u. 4).

## Teil 2: Laufzeitmessung

Testen Sie durch Laufzeitmessungen die Effizienz Ihres Sortierverfahrens. Beschränken Sie sich dabei auf das Sortieren von int-Zahlen. Die benötigte CPU-Zeit läßt sich mit der Bibliotheksfunktion **`clock`** messen:

```
clock_t clock(void);
```

**`clock_t`** ist als **`unsigned int`** definiert. **`clock()`** liefert die CPU-Zeit in Uhrenticks zurück, die das Programm seit Beginn seiner Ausführung verbraucht hat. Wird die Anzahl der Uhrenticks durch die Konstante **`CLOCKS_PER_SEC`** dividiert, erhält man die CPU-Zeit in Sekunden. Folgende Definitionsdatei ist einzufügen:

```
#include <time.h>
```

Geeignete Test-Dateien können mit einem Zufallsgenerator generiert werden. Verwenden Sie dazu die Bibliotheksfunktion

```
int rand(void);
```

**`rand()`** liefert eine ganzzahlige Pseudozufallszahl, die im Bereich von **`0`** bis **`RAND_MAX = 32767`** liegt. Es ist notwendig, folgende Definitionsdatei einzufügen:

```
#include <stdlib.h>
```

Führen Sie Messungen für unterschiedlich große Dateien durch: N = 10000, N = 20000, N = 30000, N = 40000. Variieren Sie dabei die Werte von M: M = 0, M = 20 und M = N sind ausreichend. Tragen Sie bitte die gemessenen CPU-Zeiten in die Tabelle auf der nächsten Seite ein.

Hinweis: Bei einer schnellen CPU können die Zeiten so klein werden, dass sie nicht messbar sind und zu 0 werden. Lassen Sie dann einfach die Sortierroutine für dieselbe Datenmenge k-mal laufen und berechnen aus der Gesamtzeit den Mittelwert.

## Teil 3: Optimierung durch 3-Median-Strategie

Bei nahezu vorsortierten Daten zeigt das Quicksort-Verfahren ein schlechtes Verhalten. Verbessern Sie Ihr Sortierverfahren mit Hilfe der 3-Median-Strategie. Vergleichen Sie durch Laufzeitmessungen das neue mit dem alten Verfahren. Verwenden Sie dabei Dateien mit vorsortierten int-Zahlen. Variieren Sie die Anzahl der Daten: N = 10000, 20000, 30000, 40000. Tragen Sie bitte die Ergebnisse in die Tabelle auf der nächsten Seite ein.

**Abgabe:**

- Vorführen und Erklären des Programms
- Tabelle mit Laufzeitmessungen. Variieren Sie M und N wie in der Aufgabe beschrieben!

**CPU-Zeiten in msec für unsortierte Daten**

| | M= 0 | M = 20 | M = N |
|---|---|---|---|
| N = 10000 | | | |
| N = 20000 | | | |
| N = 30000 | | | |
| N = 40000 | | | |

**CPU-Zeiten in msec für sortierte Daten**

| | M = 20;<br>ohne Median-Strategie | M = 20;<br>mit Median-Strategie |
|---|---|---|
| N = 10000 | | |
| N = 20000 | | |
| N = 30000 | | |
| N = 40000 | | |

# Befreundete Funktionen (Friend-Funktionen)

Befreundete Funktionen werden wie herkömmliche Funktionen definiert und aufgerufen. Sie sind keine Methoden! Sie haben jedoch im Gegensatz zu gewöhnlichen Funktionen Zugriff auf den privaten Datenteil von Objekten, mit denen sie befreundet sind. Die Freundschaft zwischen einer Funktion und einer Klasse wird hergestellt, indem der Funktionsprototyps mit dem Schlüsselwort friend in die Klassendefinition geschrieben wird.

Beispiel:

```
class Complex
{
public:
   friend Complex plus(Complex x, Complex y); // befreundete Funktion
   Complex plus_m(Complex x);                 // Methode
   // ...
private:
   double real;
   double im;
};

Complex plus(Complex x, Complex y)
{
      Complex z;
      z.real = x.real + y.real;
      z.im = x.im + y.im;
      return z;
}

Complex Complex::plus_m(Complex y)
{
      Complex z;
      z.real = real + y.real;
      z.im = im + y.im;
      return z;
}

void main()
{
      Complex a, b, c;
      //...
      a = plus(b,c);    // a = b+c
      a = b.plus_m(c);  // a = b+c
}
```

Vergleichen Sie den Unterschied zwischen der befreundeten Funktion plus und der Methode plus_m.

Als befreundete Funktionen können auch überladene Operatoren verwendet werden:

```
class Complex
{
public:
   friend Complex operator+(Complex x, Complex y);
   friend Complex operator-(Complex x, Complex y);
   friend int operator<(Complex x, Complex y); // Kleiner-Op.
   // ...
};
```

```
void main()
{
	Complex a, b, c;
	//...
	a = a - b + c;
	if (a < b)
		// ...;
}
```

Eine sehr wichtige Anwendung ist die Definition überladener Ein- und Ausgabeoperatoren als befreundete Funktionen, z.B.:

```
class Complex
{
public:
	friend ostream& operator<<(ostream& s, const Complex& x);
	friend istream& operator>>(istream& s, Complex& x);
	// ...
private:
	double real;
	double im;
};

// Ueberladen des Ausgabeoperators:
ostream& operator<<(ostream& s, const Complex& x)
{
	s << x.real << "+" << x.im << "*i";
	return s;
}

// Ueberladen des Eingabeoperators:
istream& operator>>(istream& s, Complex& x)
{
	s >> x.real;
	s.ignore(1);	// Ignoriere "+"
	s >> x.im;
	s.ignore(2);	// Ignoriere "*i"
	return s;
}

void main()
{
	Complex a, b, c;
	//...

	// Standard-Ein/Ausgabe
	cin >> a >> b;
	cout << a+b << endl;

	// Datei-Ein/Ausgabe
	ifstream fin("eingabe.txt");
	ofstream fout("ausgabe.txt");

	while (fin >> c)
	{
		// bearbeite c: ...
		fout << c << endl;
	}

}
```