Info2 (Ahaus) – Sommersemester 17

Aufgabenblatt 2: Funktionen und Datenstrukturen

Kapitel 1: Praktikumsaufgaben

2.1.1: Datenstruktur komplex

Erstellen Sie (noch ohne die Verwendung von Klassen und Überladung) eine Datenstruktur `komplex` zur Speicherung einer komplexen Zahl. Erstellen Sie dann eine Reihe von Funktionen, die auf dieser Datenstruktur arbeiten. Insbesondere sollten darunter Funktionen zur Initialisierung einer komplexen Zahl und zum Rechnen mit komplexen Zahlen (z. B. Addition, Multiplikation, Betrag, …) sein.

Schreiben Sie in einer separaten Datei ein Testprogramm, das alle Funktionen intensiv testet.

Tipp 1

Deklarieren Sie zuerst eine passende Datenstruktur komplex, die die notwendigen Elemente einer komplexen Zahl, also den Real- und Imaginäranteil verwaltet. Die Struktur könnte beispielsweise folgendermassen aussehen:

```
struct komplex
   {
   double re; // Realteil
   double im; // Imaginaerteil
   };
```

Beginnen Sie mit der Implementierung wichtigsten Funktionen. Implementieren Sie auch eine Funktion, um zwei Komplexe Zahlen auf Gleichheit zu prüfen, um sich Ihre Tests zu vereinfachen:

```
komplex komp_init( double r = 0, double i = 0 ); // Initialisierung einer komplexen Zahl mit
Defaultparametern
int komp_equal(komplex a, komplex b ); // Vergleich zweier komplexer Zahlen auf Gleichheit
```

Erstellen Sie direkt zu Beginn eigene Tests, die Ihre Funktionen ausgiebig testen.

2.1.2: Datenstruktur datum

Implementieren Sie (noch ohne die Verwendung von Klassen und Überladung) eine Datenstruktur `datum` zur Speicherung eines Kalenderdatums (Tag, Monat, Jahr). Vereinfachend gehen Sie davon aus, dass ein Jahr 12 Monate mit je 30 Tagen hat. Erstellen Sie dann eine Reihe von Funktionen, die auf dieser Datenstruktur arbeiten. Insbesondere sollten darunter Funktionen zur Initialisierung eines korrekten Kalenderdatums und zum Rechnen mit Kalenderdaten (z. B. Datum + Tage, Differenz zwischen zwei Kalenderdaten, ...) sein.

Schreiben Sie in einer separaten Datei ein Testprogramm, das alle Funktionen intensiv testet.

2.1.3: Datenstruktur mystring

Implementieren Sie (noch ohne die Verwendung von Klassen) eine Datenstruktur `mystring`. Diese Datenstruktur soll intern einen dynamisch allokierten Puffer für eine Zeichenkette enthalten und ausschließlich durch die folgende

Schnittstelle bedient werden:

```
mystring *str_create()
```

Legt einen leeren String an.

```
void str_destroy( mystring *s)
```

Beseitigt einen mit str_create angelegten String `s`.

```
void str_set( mystring *s, char *buf)
```

Kopiert den im Buffer buf (0-terminiert) übergebenen Inhalt in den String `s`.

```
const char *str_get( mystring *s)
```

Liefert (z.B. für Ausgabezwecke) einen konstanten Zeiger auf die interne Zeichenkette.

```
void str_add( mystring *s, mystring *t)
```

Fügt den String `t` an den String `s` an.

```
void str_insert( mystring *s, mystring *t, int pos)
```

Fügt den String `t` an der Position pos in den String `s` ein, sofern es sich um eine vernünftige Positionsangabe handelt.

```
int str_length( mystring *s)
```

Liefert die Länge des Strings `s`. Ein leerer String wird an der Länge 0 erkannt.

```
int str_test( mystring *s, mystring *t)
```

Überprüft, ob der String `t` im String `s` vorkommt und gibt entsprechend 1 oder 0 zurück.

```
int str_compare_cs( mystring *s, mystring *t)
```

Vergleicht die beiden Strings `s` und `t` case-sensitive, d.h. mit Berücksichtigung von Groß- und Kleinschreibung. Gibt bei Gleichheit 1 bei Ungleichheit 0 zurück.

```
int str_compare_cis( mystring *s, mystring *t)
```

Vergleicht die beiden Strings `s` und `t` case-insensitive, d.h. ohne Berücksichtigung von Groß- und Kleinschreibung. Gibt bei Gleichheit 1 bei Ungleichheit 0 zurück.

```
void str_extract(mystring *s, mystring *t, int from, int to)
```

Extrahiert den Teilstring von `from` bis `to` aus dem String `s`, sofern es sich bei `from` und `to` um vernünftige Positionsangaben handelt, und kopiert diesen Teilstring in den String `t`.

```
void str_setchar(mystring *s, int pos, char c)
```

Setzt den Buchstaben `c` an die Position pos im String `s`, sofern es sich bei `pos` um eine vernünftige Positionsangabe handelt.

```
char str_getchar(mystring *s, int pos)
```

Liefert den Buchstaben an der Position `pos` im String `s` oder 0, wenn die Position im String nicht existiert. Den zur Speicherung der internen Daten erforderlichen Speicher allokieren Sie bei Bedarf in diesen Funktionen.

Schreiben Sie in einer separaten Datei ein Testprogramm, das alle Funktionen des neuen Datentyps intensiv testet.

Die Headerdatei, die die vorgegebenen Schnittstellen enthält, finden Sie als Anhang dieser Aufgabe.

Kapitel 2: Vertiefung und Selbsttest

2.2.1: Datenstruktur vektor

Implementieren Sie (noch ohne die Verwendung von Klassen und Überladung) eine Datenstruktur `vektor` zur Speicherung eines dreidimensionalen reellen Vektors. Erstellen Sie dann eine Reihe von Funktionen, die auf dieser Datenstruktur arbeiten. Insbesondere sollten darunter Funktionen zur Initialisierung eines Vektors und zum Rechnen mit Vektoren (z. B. Addition, Multiplikation mit Skalar, Skalarprodukt, Vektorprodukt, Länge, Normalisierung, …) sein.

Schreiben Sie in einer separaten Datei ein Testprogramm, das alle Funktionen intensiv testet.

2.2.2: Datenstruktur zeit

Erstellen Sie (noch ohne die Verwendung von Klassen und Überladung) eine Datenstruktur `zeit` zur Speicherung einer Uhrzeit (Stunde, Minute, Sekunde). Erstellen Sie dann eine Reihe von Funktionen, die auf dieser Datenstruktur arbeiten. Insbesondere sollten darunter Funktionen zur Initialisierung einer Uhrzeit und zum Rechnen mit Zeiten (z. B. Uhrzeit + Minuten, Differenz zwischen zwei Uhrzeiten, ...) sein.

Schreiben Sie in einer separaten Datei ein Testprogramm, das alle Funktionen intensiv testet.