Fachhochschule Bingen

Programmieren

Abstrakte Datentypen
Modulare Programmierung

Prof. Dr. Maximilian Mengel,
Professur Programmiermethodik,
Grundlagen der Informatik und Multimedia
Gebäude 1, Raum 212
Tel.: 06721-409 152
E-Mail: mengel@fh-bingen.de

Abstrakte Datentypen

- Viele Probleme lassen sich ohne geeignete Datentypen kaum lösen. Mit entsprechenden Datentypen jedoch wird die Lösung häufig sehr einfach
  - Berechnung von Schwingkreisen benötigt komplexe Zahlen als Datentyp
  - 2D-Planungen benötigen 2D-Koordinaten als Datentyp
- In C gibt es jedoch weder komplexe Zahlen noch 2D-Koordinaten als Datentyp

Datentypen und Funktionen

- Durch neue abstrakte Datentypen (meistens per struct) können auf die jeweilige Anwendung angepaßte Variablen benutzt werden.
- Um mit diesen Variablen zu arbeiten, muß jedoch auf Operatoren der zugrunde liegenden Datentypen zurückgegriffen werden
- => Funktionen und Prozeduren zur Manipulation der neuen Datentypen

Beispiel: komplexe Zahlen

- Datentyp

```
struct complex{
  double re;
  double im;
  };
```

- Funktionen und Prozeduren
  - Addition, Subtraktion, Multiplikation, …
  - Betrag, Winkel
  - writecomplex, readcomplex

Module

- Idee:
  - Datenstrukturen und die zugehörigen Operatoren werden unabhängig von der „restlichen“ Programmierung in einem **Modul** entwickelt und abgelegt
- Vorteil
  - Testen eines Moduls unabhängig vom Programm
  - Unabhängige Entwicklung von Modulen möglich
  - Mehrfache Verwendung einmal entworfener Module

Exkurs: die Programmiersprache Modula

- Strukturen und Syntax von Pascal
- Neues Konzept: Modul
  - Besteht aus zwei Dateien
    - definition module
    - implementation module

Modula: Definition Module

- Interface eines Moduls benennt alle exportierten Datentypen, alle nutzbaren Funktionen und Prozeduren sowie eventuelle Konstanten
- Beispiel Interface
  - Datentyp complex
    - struktur bestehend aus Real- und Imaginärteil
  - Funktion add
    - Parameter: zwei Werte vom Typ complex
    - Rückgabe: eine Wert vom Typ complex
  - Funktion sub
    - siehe add()
  - Funktion ...

Modula: Implementation Module

- Implementierung der exportierten Datenstrukturen und aller Prozeduren und Funktionen
  - Hilfsfunktionen können intern implementiert und genutzt werden, sind jedoch von außerhalb nicht nutzbar
  - Benutzer eines Moduls sollen und dürfen kein Wissen über die konkrete Implementierung besitzen

Modula: Vorteile/Nachteile

- Vorteile
  - Aufteilung eines Programms in mehrere (unabhängige) Module
  - Kapselung von Datentyp und Operatoren
  - Verbergen der Implementierung
- Nachteil
  - Datentyp ist exportiert
    - Genau eine Datenrepräsentation
    - Direkter Zugriff auf die Daten ist möglich
  - Keine eigenen Operatoren möglich: +, *, …
- => Objekt orientierte Sprachen (z.B. C++)

Abstrakte Datentypen in C

- Aufteilung der Entwurfsarbeit in
  - Design verschiedener abstrakter Datentypen (möglichst unabhängig von der Anwendung)
  - Design der eigentlichen Anwendung
- => Logische Aufteilung von Programmen in
  - Anwendungsspezifische Programmteile
  - Abstrakte Datentypen
- Vorteil:
  - Die abstrakten Datentypen können per Copy-and-Paste in anderen Anwendungen wieder verwendet werden
  - Pflege, Verbesserung und Fehlerbehebung vereinfachen sich

Aufteilung auf Dateien

- In C kann ein Projekt aus mehreren Source-Dateien bestehen
  - Die eigentliche Anwendung
  - Verschiedene Module mit benötigten Datentypen und deren Operationen
- Damit dies möglich ist, müssen jedoch verschiedene Teile (entsprechend dem definition Module aus Modula) allgemein bekannt sein
- Dies wird durch Header-Dateien erreicht

Header-Datei

- In einer Header-Datei werden, wie in einem definition Module von Modula, folgende Teile bekannt gemacht
  - exportierten Datentypen,
  - nutzbaren Funktionen und Prozeduren
  - eventuelle globale Konstanten und Variablen
- In der Datei mit der eigentlichen Anwendung wird die Header-Datei per #include eingefügt

Header-Datei: Beispiel complexe Zahlen

- Headerfile: complex.h:

```
typedef struct
{
  double re, im;
} t_complex;

t_complex cadd(t_complex a, t_complex b);

t_complex cmult(t_complex a, t_complex b);

double betrag (t_complex a);
……
```


Implementierung eines Moduls

- Die Implementierung des Moduls geschieht in einer separaten Datei
  - Einbinden der Header-Datei per **`#include`**
  - Eventuell Nutzung weiterer Module
  - Implementierung jeder Prozedur oder Funktion der Header-Datei
  - Vereinbarung und Implementierung eventuell benötigter Hilfsfunktionen und -prozeduren

Implementierung: Beispiel complexe Zahlen

- Implementierung: complex.c:

```
#include <math.h>
#include "complex.h"

t_complex cadd(t_complex a, t_complex b)
{
   t_complex c = {0,0};
   c.re = a.re + b.re;
   c.im = a.im + b.im;
   return c;
}

t_complex cmult(t_complex a, t_complex b)
{
……}
```


Mehrfache Deklaration

- In C dürfen Prozeduren, Funktionen, Datentypen, Variablen und Konstanten immer genau einmal deklariert bzw. implementiert werden
  - Eine Header-Datei (in der ja diese Deklarationen stattfinden) darf deshalb auch nur einmal in eine Source-Datei eingefügt werden
  - Wenn verschiedene abstrakte Datentypen aufeinander aufbauen, wird dies z.T. sehr schwierig
- Um dies zu vereinfachen, existiert die bedingte Übersetzung

Bedingte Übersetzung

- Der C Preprozessor kann benutzt werden, um abhängig von Bedingungen die Übersetzung von Quell-Code durchzuführen, oder zu unterbinden
- Folgende Direktiven existieren:
  - #if BEDINGUNG
  - #elif BEDINGUNG
  - #else
  - #ifdef NAME
  - #ifndef NAME
  - #endif

Bedingte Übersetzung

- Die Bedingungen beim **`#if`** bzw. **`#elif`** werden hierbei gemäß C-Konvention (zur Übersetzungszeit) interpretiert. Häufig werden hierbei Compiler-Flags ausgewertet
  - **`#if DEBUG ...`**
  - Der Ausdruck **`defined(NAME)`** liefert eine 1 wenn NAME durch den Preprozessor definiert wurde (**`#define ...`**)
- Ausdrücke mit **`#ifndef`** bzw. **`#ifdef`** stehen als Abkürzung für:
  - **`#if !defined(NAME)`**
  - **`#if defined(NAME)`**

Beispiele: Bedingte Übersetzung

- Plattform abhängige Übersetzung

```
#if SYSTEM == BSD
  ...
#elif SYSTEM == MSDOS
  ...
#else
  ...
#endif
```

- Globale „Schalter“

```
#define DEBUD

#ifdef DEBUG
  printf(“Debug Ausgabe wert: %d“,wert) ;
#endif
```


Beispiel: Headerfile

- Die bedingte Übersetzung wird auch genutzt, um die Problematik mehrfacher Deklarationen zu entschärfen:
  - Header-Datei complex.h:

```
#ifndef __complex
#define __complex

typedef struct
{
  double re, im;
} t_complex;
... ...
#endif
```

Klausuraufgabe Prog 2 (WS 02/03)

- Um die Bruchrechnung mathematisch korrekt durchführen zu können kann man sich einen abstrakten Datentyp Brüche definieren, der aus einer entsprechenden Struktur sowie den zugehörigen Funktionen besteht.
  - a) Definieren Sie eine Struktur Bruch, die einen Bruch bestehend aus einem vorzeichenbehafteten Zähler und einem vorzeichenlosen Nenner repräsentieren kann.
  - b) Um Brüche kürzen zu können benötigt man häufig die Funktion des größten-gemeinsamen-Teilers. Implementieren Sie die Funktion ggt() die folgende mathematischen Sachverhalt für zwei **positive Zahlen** A und B ausnutzt:

    A>B: ggt(A,B) = ggt(A-B,B)

    A<B: ggt(A,B) = ggt(A,B-A)

    A=B: ggt(A,B) = A

... Klausuraufgabe Prog 2 (WS 02/03)

  - c) Implementieren Sie eine Prozedur kuerzen(), die einen ungekürzten Bruch übergeben bekommt und diesen dann kürzt. Beachten Sie hierbei das Vorzeichen des Zählers!
  - d) Um zwei positive Brüche zu addieren kann man folgenden mathematischen Zusammenhang ausnutzen:

$$\frac{Z1}{N1} + \frac{Z2}{N2} = \frac{Z1 * N2 + Z2 * N1}{N1 * N2}$$

    Implementieren Sie eine Funktion add(), die zwei Brüche übergeben bekommt und das Ergebnis zurückgibt. Ihr Programm soll die Vorzeichen selbstverständlich korrekt berücksichtigen.
  - e) Implementieren Sie eine weitere Funktion sub(), mit der von einem Bruch ein anderer Bruch abgezogen werden kann. Überlegen Sie sich inwieweit Sie bereits implementierte Funktionen nutzen können.
- Zu beachtende Hinweise:
  - Die Ergebnisse Ihrer Funktionen sollten immer soweit wie möglich gekürzt sein!