Einführung in die Programmierung

Bachelor of Science

Prof. Dr. Rethmann

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

WS 2009/10

Übersicht

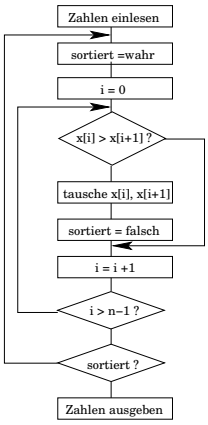
- Arrays
- Datentypen, Operatoren und Kontrollstrukturen
- Funktionen und Zeiger
- Zeichenketten
- Dateioperationen und Standardbibliotheken
- *strukturierte Programmierung*
- modulare Programmierung

Unstrukturierte Programmierung

Früher verwendete man *Sprünge* in Programmen und *Flussdiagramme* zur Darstellung des Ablaufs.

Nachteil: Sprünge an jede beliebige Stelle erlaubt → Spaghetti-Code

In der Folge: Beschränken auf wenige Kontrollstrukturen, Top-Down Entwicklung, verwenden von *Prozeduren* als Strukturmittel.



Strukturierte Programmierung

Was ist das?

- Funktionen und Prozeduren werden dazu benutzt, Programme zu organisieren. (`strToInt`, `getHostByName`, ...)
- Oft gebrauchte Funktionen, Prozeduren und Daten werden in Bibliotheken (Libraries) zur Verfügung gestellt. (`sqrt`, `sin`, `printf`, `scanf`, `malloc`, ...)
- Programmiersprachen unterstützen diesen Stil, indem sie Techniken für die Argumentübergabe an Funktionen und das Liefern von Funktionswerten bereitstellen.
- *Beispiele:* Algol68, FORTRAN, Pascal und C
- Gegenbeispiele: Assembler, Cobol, Basic

Strukturierte Programmierung

Ziele:

- Wartbarer und erweiterbarer Code.
- Wiederverwendung von Algorithmen.
- Wiederverwenden von Code nicht durch Copy&Paste, sondern durch allgemeingültige Funktionen.

Probleme:

- *Typisierung:* z.B. tauschen zweier Werte
  - `void swap1(int *a, int *b)`
  - `void swap2(double *a, double *b)`
- *allgemeine Datentypen:* Wie können wir Vergleichsoperatoren z.B. zum Sortieren bereitstellen?

Sortieren

Ziel:

- Sortieren durch Prozedur `sort` verfügbar machen
- `sort` soll unabhängig vom verwendeten Datentyp sein

Annahmen:

- nur internes Sortieren (Werte in Vektor gespeichert)
- die Anzahl der Datensätze ist bekannt
- Datensätze werden getauscht, wir legen keinen Index an

Anmerkung:

- Es geht hier nicht um effiziente Sortierverfahren, sondern um Programmiertechniken.
- effiziente Sortierverfahren werden in der Veranstaltung „Algorithmen und Datenstrukturen“ besprochen

erster Versuch

```
#include <stdio.h>
...
int main(void) {
    long u[] = {9, 8, 7, 6, 5, 4, 3, 2, 1};
    double v[] = {5.0, 4.0, 3.0, 2.0, 1.0};

    sortLong(u, 9);
    outputLong(u, 9);

    sortDouble(v, 5);
    outputDouble(v, 5);

    return 0;
}
```


erster Versuch

```
void sortLong(long *a, int n) {
    int i, j;

    for (i = 0; i < n; i++)
        for (j = i + 1; j < n; j++)
            if (a[i] > a[j]) {
                long t = a[i];
                a[i] = a[j];
                a[j] = t;
            }
}

void outputLong(long *a, int n) {
    int i;

    for (i = 0; i < n; i++)
        printf("%ld\n", a[i]);
}
```

erster Versuch

```
void sortDouble(double *a, int n) {
    int i, j;

    for (i = 0; i < n; i++)
        for (j = i + 1; j < n; j++)
            if (a[i] > a[j]) {
                double t = a[i];
                a[i] = a[j];
                a[j] = t;
            }
}

void outputDouble(double *a, int n) {
    int i;

    for (i = 0; i < n; i++)
        printf("%lf\n", a[i]);
}
```


erster Versuch

Probleme:

- die Funktionalitäten sind mehrfach implementiert
  - Sortieren für `long`, `double` und ...
  - Ausgabe für `long`, `double` und ...
  - Objekte tauschen für `long`, `double` und ...
- Wiederverwendung nur durch Copy&Paste

⇒ Fehler sind an vielen Programmstellen zu beseitigen!

Lösung:

- Unabhängigkeit vom Datentyp durch Zeiger auf `void`.
- Objekte tauschen durch Prozedur `swap()` realisieren.

Objekte vertauschen: 1. Versuch

```
void swap(void *a, int x, int y) {
    void t = a[x];
    a[x] = a[y];
    a[y] = t;
}
```

So nicht!

- Der Wert eines Objekts vom Typ `void` kann in keiner Weise verarbeitet werden, er darf weder explizit noch implizit in einen anderen Typ umgewandelt werden.
  Compiler: `variable or field 't' declared void`
- Index-Operator funktioniert bei Zeiger auf `void` nicht.
  Compiler: `warning: dereferencing 'void *' pointer`

Objekte vertauschen: 2. Versuch

```
void swap(void *a, void *b) {
    void *t;
    *t = *a;
    *a = *b;
    *b = *t;
}
```

So nicht!

Der Inhaltsoperator funktioniert bei Zeiger auf `void` nicht!

der Compiler liefert:

```
invalid use of void expression
warning: dereferencing 'void *' pointer
```


Objekte vertauschen

Inhalte byte-weise vertauschen:

```
void swap(void *a, void *b, int size) {
    char c;
    int i;
    char *x = a;
    char *y = b;

    for (i = 0; i < size; i++) {
        c = *(x+i);
        *(x+i) = *(y+i);
        *(y+i) = c;
    }
}
```


Reflexion

Was haben wir erreicht?

- gemeinsame Funktionalität `swap` als eine in sich abgeschlossene Funktion bereitgestellt
- `swap` ist unabhängig vom verwendeten Datentyp

Was ist noch zu tun?

- beschränken auf eine Funktion `sort`
- beschränken auf eine Funktion `output`

zweiter Versuch

```
...
typedef enum {SHORT, LONG, DOUBLE, CHAR} type_t;
...
int main(void) {
  long u[] = {9, 8, 7, 6, 5, 4, 3, 2, 1};
  double v[] = {6.0, 5.0, 4.0, 3.0, 2.0, 1.0};

  sort(u, 9, LONG);
  output(u, 9, LONG);

  sort(v, 6, DOUBLE);
  output(v, 6, DOUBLE);

  return 0;
}
```


zweiter Versuch

```
void sort(void *a, int n, type_t type) {
  int i, j;

  for (i = 0; i < n; i++)
    for (j = i + 1; j < n; j++)
      if (type == LONG) {
        long *ta = a;

        if (ta[i] > ta[j])
          swap(&ta[i], &ta[j], sizeof(long));
      } else if (type == DOUBLE) {
        double *ta = a;

        if (ta[i] > ta[j])
          swap(&ta[i], &ta[j], sizeof(double));
      } ...
}
```

zweiter Versuch

```
void output(void *a, int n, type_t type) {
   int i;

   for (i = 0; i < n; i++)
      if (type == LONG)
         printf("a[%d] = %ld\n", i,
                   ((long *) a)[i]);

      else if (type == CHAR)
         printf("a[%d] = %c\n", i,
                   ((char *) a)[i]);

      else if (type == DOUBLE)
          printf("a[%d] = %f\n", i,
                    ((double *) a)[i]);

      else ...
}
```


zweiter Versuch

Probleme:

- lange, unübersichtliche `if/else`-Anweisungen
- Erweiterung auf selbstdefinierte Datentypen wie `Kunde`, `Rechnung` oder `Vertrag` ist nur möglich, falls Source-Code der Funktionen `sort` und `output` vorliegt

→ die `if/else`-Anweisungen müssen erweitert werden

Lösung:

- Vergleichsoperation pro Datentyp implementieren und an die Sortierfunktion übergeben.
- Operation zur Darstellung des Datentyps als Zeichenkette implementieren und an die Ausgabefunktion übergeben.

⇒ *Zeiger auf Funktionen*

Zeiger auf Funktionen

Jede Funktion besitzt eine Adresse:

```
int min(int a, int b) {
    return (a < b) ? a : b;
}
int max(int a, int b) {
    return (a > b) ? a : b;
}

void main(void) {
    int (*fp)(int, int);

    fp = min;          /* fp zeigt auf min */
    printf("min(5, 7) = %d\n", (*fp)(5, 7));

    fp = max;          /* fp zeigt auf max */
    printf("max(5, 7) = %d\n", (*fp)(5, 7));
}
```


Zeiger auf Funktionen

`int (*fp)(int, int)`

- `fp` ist ein Zeiger auf eine Funktion, die einen `int`-Wert liefert und zwei `int`-Werte als Parameter verlangt.
- `*fp` kann für `min` und `max` benutzt werden.
- nicht verwechseln mit `int *fp(int, int)`: Funktion, die zwei `int`-Werte als Parameter hat und einen Zeiger auf einen `int`-Wert als Ergebnis liefert!

Zeiger auf Funktionen können

- zugewiesen,
- in Vektoren eingetragen,
- an Funktionen übergeben werden usw.

Zeiger auf Funktionen

```
#include <stdio.h>

int main(void) {
    int zahl;
    int (*fptr[])(const char *, ...)
            = { scanf, printf };

    (*fptr[1])("Zahl? ");
    (*fptr[0])("%d", &zahl);
    (*fptr[1])("Die Zahl lautet %d\n", zahl);

    return 0;
}
```


Zeiger auf Funktionen

Beispiele aus der Standardbibliothek:

- ```
  void qsort(void *base, size_t nmemb,
             size_t size,
             int (*compar)(void *, void *))
  ```
  sortiert ein Array mit `nmemb` Elementen der Größe `size`, das erste Element steht bei `base`
- ```
  void *bsearch(void *key, void *base,
                size_t nmemb, size_t size,
                int (*compar)(void *, void *))
  ```
  durchsucht ein Array mit `nmemb` Elementen der Größe `size` (erstes Element bei `base`) nach einem Element `key`
- bei beiden Funktionen muss jeweils eine Vergleichsfunktion `compar` angegeben werden

Zeiger auf Funktionen

Beispiele aus der Standardbibliothek: (Fortsetzung)

- `void exit(int status)` beendet ein Programm normal. Die `atexit`-Funktionen werden in umgekehrter Reihenfolge ihrer Hinterlegung durchlaufen.
- `int atexit(void (*fcn)(void))` hinterlegt Funktion `fcn`. Liefert einen Wert ungleich 0, wenn die Funktion nicht hinterlegt werden konnte.

Zeiger auf Funktionen werden oft bei GUI-Elementen verwendet, wo sie als *callback function* bei einem Benutzer-Event (Ereignis) aufgerufen werden!

Sortieren

Was ist zu tun?

- lange, unübersichtliche `if/else`-Anweisungen ersetzen
- Vergleichsfunktionen pro Datentyp implementieren und an die Sortierfunktion als Parameter übergeben
- pro Datentyp eine Darstellung als Zeichenkette implementieren und an die Ausgabefunktion als Parameter übergeben

Sortieren

```
...
int main(void) {
    char t[] = {'f', 'e', 'd', 'c', 'b', 'a'};
    long u[] = {9, 8, 7, 6, 5, 4, 3, 2, 1};
    double v[] = {6.0, 5.0, 4.0, 3.0, 2.0, 1.0};

    sort(t, 6, sizeof(char), cmpChar);
    output(t, 6, charToString);

    sort(u, 9, sizeof(long), cmpLong);
    output(u, 9, longToString);

    sort(v, 6, sizeof(double), cmpDouble);
    output(v, 6, dbleToString);

    return 0;
}
```


Sortieren

```
void sort(void *a, int n, int size,
          int (*cmp)(void *, void *)) {
    int i, j;

    for (i = 0; i < n; i++)
        for (j = i + 1; j < n; j++) {
            char *x, *y;

            x = (char *) a + i * size;
            y = (char *) a + j * size;
            if ((*cmp)(x, y) > 0)
                swap(x, y, size);
        }
}
```


Sortieren

```
int cmpLong(void *a, void *b) {
    long *x = a;
    long *y = b;

    return *x - *y;
}
int cmpDouble(void *a, void *b) {
    double *x = a;
    double *y = b;

    return *x - *y;
}
int cmpChar(void *a, void *b) {
    char *x = a;
    char *y = b;

    return *x - *y;
}
```


Sortieren

```
void output(void *a, int n,
      char * (*toString)(void *, int, char *)) {
   int i;
   char str[80];

   for (i = 0; i < n; i++)
      printf("a[%d] = %s\n", i,
          toString(a, i, str));
}

char * charToString(void *x, int pos, char *s) {
    char *a = x;

    sprintf(s, "%c", a[pos]);
    return s;
}
```


Sortieren

```
char * longToString(void *x, int pos, char *s) {
    long *a = x;

    sprintf(s, "%ld", a[pos]);
    return s;
}

char * dbleToString(void *x, int pos, char *s) {
    double *a = x;

    sprintf(s, "%.2lf", a[pos]);
    return s;
}
```


Sortieren

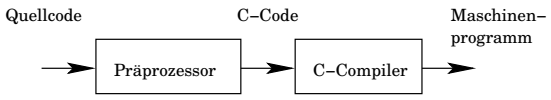
Frage: Anstelle von mehreren Sortierprozeduren haben wir jetzt mehrere Vergleichsfunktionen. Warum soll das besser sein?

- die alten `sort`-Funktionen enthalten mittels Copy&Paste duplizierten Code → fehleranfällig, schlecht erweiterbar
- für jeden selbstdefinierten Typ müssen wir sowieso
  - eine Vergleichsfunktion bereitstellen, denn: Nach welchem Schlüssel soll sortiert werden?
  - eine Darstellung als Zeichenkette bereitstellen, denn: Wie soll die Ausgabe formatiert sein?

Der C-Präprozessor bietet weitere Möglichkeiten, den Code maschinen- und datentyp-unabhängig zu schreiben.

C-Präprozessor

Der Präprozessor bearbeitet den Programm-Code vor der eigentlichen Übersetzung.



Präprozessordirektiven sind Programmzeilen, die mit einem `#`-Zeichen beginnen:

- `#include`: Inhalt einer Datei während der Übersetzung einfügen
- `#define`: Einen Namen durch eine beliebige Zeichenfolge ersetzen (parametrisierbar)

C-Präprozessor: `#include`

Eine Quellzeile wie

`#include <filename>` oder `#include "filename"`

wird durch den Inhalt der Datei `filename` ersetzt:

- `"filename"`: die Suche nach der Datei beginnt dort, wo das Quellprogramm steht.
- `<filename>`: die Datei wird in einem speziellen Verzeichnis gesucht. (Linux: `/usr/include/`)

C-Präprozessor: #include

Für den Inhalt der eingefügten Dateien gibt es keine Einschränkungen.

Aber: In der Regel werden nur Definitionsdateien eingebunden.

Definitionsdateien enthalten:

- `#define`-Anweisungen,
- weitere `#include`-Anweisungen,
- Typdeklarationen und
- Funktionsprototypen

Die Deklarationen eines großen Programms werden zentral gehalten: Alle Quelldateien arbeiten mit denselben Definitionen und Variablendeklarationen.

C-Präprozessor: #define

`#define <name> <ersatztext>` bewirkt, dass im Quelltext die Zeichenfolge `name` durch `ersatztext` ersetzt wird.

Der Ersatztext ist der Rest der Zeile. Eine lange Definition kann über mehrere Zeilen fortgesetzt werden, wenn ein `\` am Ende jeder Zeile steht, die fortgesetzt werden soll.

```
#define PRIVATE static
#define PUBLIC
#define ERROR \
    printf("\aEingabedaten nicht korrekt\n");
```

Der Gültigkeitsbereich einer `#define`-Anweisung erstreckt sich von der Anweisung bis ans Ende der Datei.

Bezeichner von Makros werden in der Regel groß geschrieben, einzelne Wörter werden durch Unterstriche getrennt:

```
#define MIN_WIDTH  256
#define MAX_SCORE 1000
```


C-Präprozessor: #define

Makros mit Parametern erlauben, dass der Ersatztext bei verschiedenen Aufrufen verschieden sein kann:

```
#define MAX(A, B) ((A) > (B) ? (A) : (B))
```

Ein Makroaufruf ist kein Funktionsaufruf! Ein Aufruf von `MAX` wird direkt im Programmtext expandiert. Die Zeile

```
x = MAX(p + q, r + s);
```

wird ersetzt durch die Zeile

```
x = ((p + q) > (r + s) ? (p + q) : (r + s));
```

Textersetzung! Keine Auswertung von Ausdrücken!

Wichtig: hier wird der ternäre Operator `?:` verwendet, kein `return`! Überlegen Sie, warum nicht.

C-Präprozessor: #define

Hinweis: Anders als bei Funktionen genügt eine einzige Definition von `MAX` für verschiedene Datentypen.

Vorsicht: Im Beispiel werden Ausdrücke eventuell zweimal bewertet. Das führt bei Operatoren mit Nebenwirkungen (Inkrement, Ein-/Ausgabe) zu Problemen.

```
i = 2;
j = 3;
x = MAX(i++, j++);     /* i?? j?? x?? */
```

Vorsicht: Der Ersatztext muss sorgfältig geklammert sein, damit die Reihenfolge von Bewertungen erhalten bleibt:

```
#define SQR(x) x * x
```

Was passiert beim Aufruf `SQR(z + 1)`?

C-Präprozessor: #define

Makros können bereits definierte Makros enthalten:

```
#define SQR(x) (x) * (x)
#define CUBE(x) SQR(x) * (x)
```

Die Gültigkeit einer Definition kann durch `#undef` aufgehoben werden:

```
#undef MAX
#undef CUBE
int MAX(int a, int b) ...
```

Textersatz findet nur für Namen, aber nicht innerhalb von Zeichenketten statt.

C-Präprozessor: #define

Textersatz und Zeichenketten:

- `#<parameter>` wird durch `"<parameter>"` ersetzt.
- Im Argument wird `"` durch `\"` und `\` durch `\\` ersetzt.
- Resultat: gültige konstante Zeichenkette.

Beispiel: Debug-Ausgabe

```
#define DPRINTF(expr) \
        printf(#expr " = %f\n", expr);
...
DPRINTF(x/y);
/* ergibt: printf("x/y" " = %f\n", x/y); */
```


C-Präprozessor: #define

Mittels `##` können Argumente aneinandergehängt werden.

Beispiel:

```
#include <stdio.h>
#define PASTE(head, tail) head ## tail

void main(void) {
    int x1 = 42;
    printf("x1 = %d\n", PASTE(x, 1));
}
```

Frage: Wozu braucht man das?

C-Präprozessor: #define

Auszug aus `stdint.h`:

```
#if __WORDSIZE == 64
#  define __INT64_C(c)  c ## L
#  define __UINT64_C(c) c ## UL
#else
#  define __INT64_C(c)  c ## LL
#  define __UINT64_C(c) c ## ULL
#endif
```

Auszug aus `linux/ext2_fs.h`:

```
#define clear_opt(o,opt)  o &= ~EXT2_MOUNT_##opt
#define set_opt(o,opt)    o |= EXT2_MOUNT_##opt
```

C-Präprozessor: Beispiel

```
#define SWAP(A, B, T) \
{                     \
    T t = A;          \
                      \
    A = B;            \
    B = t;            \
}

void sort(long *a, int n) {
    int i, j;

    for (i = 0; i < n; i++)
        for (j = i + 1; j < n; j++)
            if (a[i] > a[j])
                SWAP(a[i], a[j], long)
}
```


C-Präprozessor: #if

Der Präprozessor selbst kann mit bedingten Anweisungen kontrolliert werden, die während der Ausführung bewertet werden:

- Texte abhängig vom Wert einer Bedingung einfügen.
- Programmteile abhängig von Bedingungen übersetzen.

Anwendung:

- um Definitionsdateien nur einmal einzufügen,
- für Debug-Zwecke und
- für systemabhängige Programmteile.

C-Präprozessor: #if

`#if <expr>` prüft den Ausdruck (konstant, ganzzahlig).

Ist der Ausdruck „wahr“, werden die folgenden Zeilen bis `else`, `elif` bzw. `endif` eingefügt und damit übersetzt.

Beispiel:

```
#define DEBUG_LEVEL 2
...
#if (DEBUG_LEVEL == 1)
    printf("buchtitel(%d): %s\n", isbn, titel);
#elif (DEBUG_LEVEL == 2)
    printf("buchtitel(%d): %s\n", isbn, titel);
    printf("buchautor(%d): %s\n", isbn, autor);
    printf("erschjahr(%d): %s\n", isbn, datum);
#endif
```


C-Präprozessor: #ifdef

`#ifdef <makro>` bzw. `#ifndef <makro>` prüft, ob ein Makro definiert bzw. nicht definiert ist.

Äquivalente Schreibweise:

```
#if defined <makro>
#if !defined <makro>
```

Beispiel: systemabhängige Übersetzung

```
#ifdef ALPHA
#   include "arch/alpha/semaphore.h"
#elif defined INTEL || defined I386
#   include "arch/i386/semaphore.h"
#endif
```


Mehrfaches Einbinden verhindern

So bitte nicht!

- datei1.h

```
#include "datei3.h"
float f = 1.0;
```

- datei2.h

```
#include "datei3.h"
char c = 'a';
```

- datei3.h

```
int glob = 1;
```


Mehrfaches Einbinden verhindern

main.c

```
#include <stdio.h>
#include "datei1.h"
#include "datei2.h"
#include "datei3.h"

int main(void) {
    glob = 42;

    printf("f = %f, c = %c, glob = %d\n",
           f, c, glob);
    return 0;
}
```


Mehrfaches Einbinden verhindern

datei3.h

```
#ifndef _DATEI3_H
#define _DATEI3_H

int global = 3;

#endif
```

Erklärung:

- Beim ersten Einfügen wird `_DATEI3_H` definiert.
- Wird die Datei nochmals eingefügt, ist `_DATEI3_H` definiert und alles bis zum `#endif` wird übersprungen.