Programmiertechnik 1

Unit 10: Strukturen

Ablauf

- Grundlagen
- Strukturen und Funktionen
- Felder von Strukturen
- Zeiger auf Strukturen
- Selbstreferentielle Strukturen
- typedefs, unions, bit-fields
- Objekte

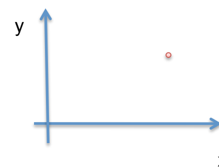
Grundlagen

- C Strukturen == Records in Pascal
  - Sammlung mehrere Variablen verschiedenen Typs
  - Gruppierung unter einem gemeinsamen Namen
- Organisation komplexer Datenstrukturen
  - Insbesondere in großen Programmen
  - Verwandte Variablen können als Einheit betrachtet werden
- Beispiele:
  - Adressinformation mit Name, Straße, Hausnummer, PLZ, Stadt
  - Koordinaten – n-Tupel von Punkten
- ANSI C
  - Definition von Zuweisungsregeln für Strukturen
  - Strukturen können Funktionsargumente und Rückgabewerte sein

Beispiel Koordinatensystem

- Schlüsselwort struct
  - Liste von Deklarationen in geschweiften Klammern
  - Optionaler Name: *structure tag* – hier: point
  - Tag kann als Name für die Struktur benutzt werden

```
struct point {
    int x;
    int y;
};
```



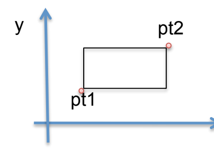
  - Variablen in der Struktur heißen *members*
  - Strukturen spannen neuen Namensraum auf
    - Member-Variablen dürfen denselben Namen wie globale Variablen tragen
  - Struktur-Deklaration definiert einen Typ
    - Können Variablen vom Struktur-Typ deklarieren

```
struct point pt;                      /* Oben deklarierte Struktur als Typ nutzen */
struct point maxpt = { 320, 200 } ;   /* Initialisierung */
```

Deklaration und Verwendung

- Deklaration ohne Variablenliste reserviert keinen Speicher
  - Tag kann später zur Deklaration von Variablen benutzt werden
  - Automatische Strukturen können durch Zuweisung oder durch Funktionsaufruf initialisiert werden

```
struct point Ursprung () {
    struct point x = { 0, 0 };
    return x;
}
struct point pt = Ursprung();
```



Distanz trigonometrisch berechnen...

- Zugriff auf member einer Struktur
  - Syntax: *structure-name . Member*

```
printf(%d,%d", pt.x, pt.y );
```

point.c

Verschachtelte Strukturen

- Beispiel Rechteck → dargestellt als Paar von Punkten

```
struct rect {
    struct point pt1;
    struct point pt2;
};
```

  - rect-Struktur enthält zwei point-Strukturen
  - Zugriff auf Struktureelemente (member) durch wiederholte Anwendung des "."-Operators

```
struct rect screen;
int x1 = screen.pt1.x;
int y1 = screen.pt1.y;
```

Strukturen und Funktionen

- Legale Operationen auf Strukturen:
  - Kopieren
  - Zuweisung
  - Adresse ermitteln mit "&"
  - Zugriff auf Strukturelemente mit "."
- Zuweisung schließt Verwendung als Funktionsargumente ein
- Strukturen können nicht miteinander verglichen werden
  - Expliziter Vergleich der Elemente muss händisch codiert werden
- Übergabe von Strukturen an Funktionen: 3 Ansätze
  - Separate Übergabe der Strukturelemente
  - Übergabe der gesamten Struktur
    - Es wird eine Kopie angefertigt; ineffizient bei großen Strukturen
  - Übergabe eines Zeigers auf die Struktur

Beispiel Funktionsargumente

- Struktur als Rückgabewert

```
struct point makepoint( int x, int y ) {
    struct point temp;

    /* x und y sind lokale Variablen (Parameter)   */
    /* kein Konflikt mit Strukturelementen x und y */
    temp.x = x;
    temp.y = y;
    return temp;
}
```

- Argumente und Rückgabewert sind Strukturen

```
struct point addpoint( struct point p1, struct point p2 ) {
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
```

rectangle.c

Zeiger auf Strukturen

- Als Funktionsparameter: (...und in selbstreferentiellen Strukturen..., später)
  - Übergabe eines Zeigers ist effizienter als Übergabe einer Struktur
  - Syntax:

    ```
    struct point * pp:
    ```
  - pp ist ein Zeiger auf eine Struktur
    - pp ist zunächst nicht initialisiert!
    - Dereferenzierung verursacht Laufzeitfehler
- Initialisierung

  ```
  struct point origin;
  pp = & origin;
  printf(”origin is: (%d,%d)\n”, (*pp).x, (*pp).y );
  ```
  - Member-Zugriff erfordert Klammern: ”.”-Operator hat Vorrang vor ”*”-Operator
  - Alternative Syntax:

    p->*member-of-structure*

    ```
    printf(”origin is: (%d,%d)\n”, pp->x, pp->y );
    ```


Mehr Syntax

- Eigenarten des „->“-Operators
  - (minus Zeichen gefolgt von > Zeichen)
  - „.“ und „->“ sind von links nach rechts assoziativ
    - Haben Vorrang vor allen anderen Operatoren
    - Keine Klammersetzung nötig...
- Verwendung

  ```
  struct rect r, *rp = r;
  ```
  - Folgende Ausdrücke sind äquivalent

    ```
    r.pt1.x
    rp->pt1.x
    (r.pt1).x
    (rp->pt1).x
    ```

```
struct {
        int len;
        char * str;
} *p;

++p->len      → inkrementiert len
                implizite Klammerung ++(p->len)
*p->str       → Zeichenkette, auf die str verweist
*p->str++     → inkrementiert str-Zeiger nach Zugriff
(*p->str)++   → inkrementiert erstes Zeichen der
                Zeichenkette
*p++->str     → inkrementiert p nach Zugriff auf
                Zeichenkette
```

Im Zweifelsfall explizit Klammern setzen!!

Felder von Strukturen

- Wollen Schlüsselworte zählen
  - Lösung 1: Zwei Felder
    - char *keyword[NKEYS];
    - int keycount[NKEYS];
  - Lösung 2 (besser): Feld von 2-Tupeln (keyword, Anzahl)
    - ```
      struct key {
          char * word;
          int count;
      } keytab[NKEYS];
      ```
  - Jedes Feldelement ist eine Struktur
  - Alternative Syntax:
    - ```
      struct key {
          char * word;
          int count;
      };
      stuct key keytab[NKEYS];
      ```

wordcount.c

sizeof

- Größen von Strukturen variieren
- Problem:
  - Allokation von Speicher für eine Struktur
  - ...für Felder von Strukturen
- Lösung:
  - sizeof-Operator
  - Einstellig, wird zur Compile-Zeit ausgewertet
  - Anwendbar auf Objekte und Typen

```
sizeof ( struct point );
sizeof ( int );
struct point * pp = malloc ( sizeof ( struct point ) * num_elem );
```

  - Nicht anwendbar um Länge von Zeichenketten zu ermitteln!!

sizeof.c

Zeiger auf Strukturen

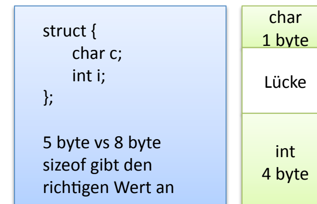
- Syntax:

```
struct key {
      char * word;
      int count;
};
struct key * keytab = malloc( sizeof(struct key) * argc );
```

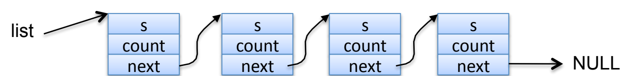
  - keytab ist ein Zeiger auf ein Feld von Strukturen
    - Kann syntaktisch als Feld behandelt werden

```
keytab[i].word = malloc( strlen(words[i]) + 1 );
strcpy( keytab[i].word, words[i] );
keytab[i].count = 0;
```

  - Zeigerarithmetik berücksichtigt Größe der Elemente
    - sizeof-Operator liefert Größe der einzelnen Feldelemente
  - Alignment
    - Feld hat womöglich „Löcher"
    - Nicht alle Mitglieder einer Struktur müssen im Speicher direkt hintereinander liegen



Selbstreferentielle Strukturen



- Strukturen können aus Elemente gleichen Typs verweisen
  - Listen, Bäume, Hash-Tables
  - Komplexe Datenstrukturen

```
struct list_elem {
    char * s;
    int count;
    struct list_elem * next;
};
struct list_elem * list;
```

Einschub: Konstruktion von Datentypen

- Ein Datentyp ist eine Mengen von Daten zusammen mit einer Familie von Operationen
- abstrakter Datentyp:
  - Beschrieben wird lediglich die Menge und die Semantik der Operationen, nicht aber die interne Repräsentation der Daten oder die Implementierung der Operationen
- Implementierung in C:
  - Typdeklaration und Deklaration von Zugriffsfunktionen in header-Datei
  - Implementierung in zugehöriger C-Datei
  - Keine explizite Möglichkeit des „information hiding"
  - Deklaration legt schon viele Implementationsdetails offen (Speicher-Abbild)

Konstruktion von Mengen

- Konstruktion neuer Mengen (evtl. auf Basis existierender Mengen M, M_i)
- Aufzählung: Menge $\{e_1, e_2, e_3, ..., e_n\}$
  - C, C++, Java 5: enum (enumerations)
- Teilmengen: Gegeben Prädikat P(x), bilde $\{x \in M \mid P(x)\}$
  - Abbildung über Zugriffsfunktionen
- k-te Potenz: Gegeben $k \in N$: Menge M^k aller k-Tupel $(a_i, a_i, ..., a_i)$ mit $a_i \in M$
  - C, C++: Array (Felder) "von M" der Länge k
  - Python: Liste (Beschränkung auf Länge per Konvention)
- Kartesisches Produkt: $M_1 \times M_2 \times ... \times M_k$: Menge der k-Tupel $(a_i, a_i, ..., a_i)$ mit $a_i \in M_i$
  - C: struct
  - Python: Tupel, Klasse

Mengenkonstruktionen (contd.)

- Disjunkte Vereinigung: Gegeben M_i: $M=\{(i,m) \mid i \in I \wedge m \in M_i\}$
  - C: union (aber: keine discriminated union)
  - Python: Konvention in Variablenverwendung
- Potenzmenge: $\wp(M)$ (Menge aller Teilmengen von M)
  - C: nur Repräsentation als bitset
    (Zahl der Elemente durch Wordbreite beschränkt)
  - Python: set
- Folgenraum: M* (Menge aller endlichen Folgen aus Elementen von M
  - Python: Liste
- Unendliche Folgen: M^∞
  (Menge aller unendlichen Folgen von Elementen aus M)
  - beschränkt auf berechenbare Folgen
  - funktionale Sprachen; C, C++, Python: Generatoren (Muster)

Mengenkonstruktionen (contd.)

- Funktionenraum:
  - Menge aller Abbildungen von I nach M
  - Funktionen
  - Python, C++, Java: Dictionaries (Mappings)
- Induktive Definition:
  - Definition der Menge durch ein Induktionsschema

Listen

- nummerierte Folgen von Werten
  - Indizierung beginnt bei 0
- Konvention: Werte sollten alle den gleichen Typ haben
  - Repräsentation von M^k und M^*
- Darstellung: Werte kommagetrennt in eckigen Klammern
  - [1,2,5]
  - [“Hallo”, “Welt”]
  - []
- Operationen: L+L, L*i, L[i], L[i:j]
  - Beispiel: N^k: [0]*k
- Funktionen:
  - len()
  - Methoden - In C implementiert als Zugriffsfunktionen
  - .append(V), .index(V), .reverse(), .sort()

Tupel

- Kreuzprodukt von k Typen
  - per Konvention: Im gleichen Kontext haben Tupel sollten Tupel immer die gleiche Länge haben, und die jeweiligen Elemente den gleichen Typ
- nicht änderbar (immutable): Elemente werden bei Erzeugung festgelegt
- Werte kommagetrennt
  - “Erika”, “Mustermann”, 42
  - In C repräsentiert durch Strukturen
- Operationen: T+T, T*i, T[i], T[i:j]
  - zusätzlich: tuple unpacking bei Zuweisung
  - x,y,z = t – Zuweisungsoperator für Strukturen arbeitet Elementweise
- Funktionen: sizeof(T);

Kartesisches Produkt mittels struct

- Klassen: Zusammenfassung von Zustand (Daten) und Verhalten
  - Verzicht auf Verhalten: “Datensätze” (records, structures)
  - C bietet keine Klassen als Sprachkonstrukt
  - Ausweg: Strukturen mit Zeigern auf Zugriffsfunktionen (C++)
- Elemente des Datensatzes über Attributnamen ansprechbar
- Klasse Person:
  - ```
    struct person {
      char * vorname;
      char * name;
      int alter;
    };
    ```
- Initialisierung statisch: struct person em = { “Erika”, “Mustermann”, 42 };
- em.alter += 1;
  printf (”%s hat Geburtstag: %d\n”, em.vorname, em.alter);

Dictionaries

- Mapping-Typ: Abbildung von Typ K auf Typ V
  - K: “Schlüssel” (key)
  - V: “Wert” (value)
- “endliche Funktion”: Abbildung ist nur für endliche Teilmenge von K definiert
- “Variable Funktion”: Abbildung kann geändert werden
- Implementierung in C:
  - Listen oder Bäume: Selbstreferentielle Strukturen
  - Felder von Strukturen
  - i.d.R. “immutable” (Schlüssel verändert seinen Wert nicht)
- Typische Zugriffsmethoden: .keys, .values, .items, .has_key
    ```

Dateien

- Dateien sind endliche Folgen, Ströme (streams) potentiell unendliche Folgen von Datensätzen eines beliebigen, aber fest gewählten Typs
  - Pascal: Festlegung des Typs mit Sprachmitteln
  - C, Java, Python, ...: Typ ist (fast immer) Byte
    - evtl. Folge von Zeichen, evtl. Folge von Zeilen
- Dateien: Speicherung von Daten zwischen Programmläufen – "persistenter" Speicher (persistent memory)
- Ströme: Austausch von Daten zwischen aktiven Programmen
  - z.B. TCP/IP-Verbindungen

Dateien (contd.)

- Lesemodus oder Schreibmodus
  - "append" (anhängen): Spezialform des Schreibmodus
- Binär- oder Textmodus
  - Unterschied nur für Windows relevant:
  - Textmodus: Zeilenenden ('\n') werden automatisch in CRLF konvertiert
  - Kein Dateiende-Zeichen – aber Rückgabewert EOF bei Lesen nach Dateiende
- Dateizeiger (file pointer):
  - Angabe der Lese- oder Schreibposition in der Datei
  - initial: Anfang der Datei (außer "append")
- Spezialdateien für Terminalinteraktion:
  - Standardeingabe, Standardausgabe, Standardfehlerausgabe

Umgang mit Dateien

- Zyklus für Verarbeitung von Dateien:
  - Öffnen der Datei (Angabe des Dateinamens und des Modus)
  - Ein/Ausgabeoperationen (lesen, schreiben)
  - Schließen der Datei
  - Betriebssystem schließt Dateien automatisch bei Programmende
- C: Öffnen der Datei mit Funktion open()

```
#include <fcntl.h>
int open(const char *path, int oflag, ...);
```

- Flaggen:
  - O_RDONLY, O_WRONLY, O_RDWR   open for reading/writing/both
  - O_APPEND   append on each write
  - O_CREAT   create file if it does not exist.... etc.
- Ergebnis von open:
  - geöffneter Dateidescriptor (kleine int Zahl)

Operationen auf Dateien

- Operationen auf geöffneten Dateideskriptoren:
  - int read(int fd, char * buf, int len);
  - int write(int fd, char * buf, int len);
  - off_t lseek(int fildes, off_t offset, int whence);
    - Whence:
      SEEK_SET – Lesezeiger relativ zu Dateianfang positionieren
    - SEEK_CUR – Lesezeiger relativ zu aktueller Position einstellen
    - SEEK_END – Lesezeiger relativ zu Dateiende einstellen
- Schließen der Datei mit close()
  - gepufferte Daten werden an das Betriebssystem übergeben
  - passiert u.U. auch automatisch, spätestens mit Ende des Programms

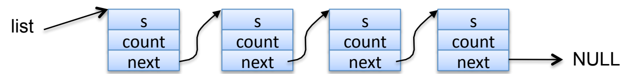
Induktiv definierte Typen

- Induktionsschema: Elemente des Typs werden induktiv aus anderen Elementen erzeugt
  - natürliche Zahlen
  - Listen
  - Bäume
  - Stacks
  - Strings
  - ...
- Implementierungsstrategie: Speicherung der Werte auf Halde (heap)
  - Speicherverwaltung: Anforderung und Freigabe von Speicher
  - Teilweise automatisch: Java (garbage collection), Python (Referenzzählung)
  - In C: Verwaltung des heap obliegt dem Programmierer

Beispiel: Induktive Definition

- Dateiverzeichnis: verschachtelte Verzeichnisse + Dateien
- Def.: Ein VBaum besteht aus
  - einem Dateinamen, oder
  - einem Paar (Verzeichnisname, VBaumListe)
- Def.: Eine VBaumListe ist
  - leer, oder
  - besteht aus einem VBaum und einer VBaumListe
- Darstellung im Speicher: “besteht aus” kann nicht durch Zusammenfügen von Speicherblöcken entstehen
  - Größe eines Werts von VBaumListe ließe sich nicht angeben
- Lösung: Zeiger (pointer) / Referenzen
  - Variablen enthalten nicht den eigentlichen Wert, sondern nur einen Verweis (Adresse) des Werts

Einfach verkettete Listen



- Elemente sind Strukturen
  - Enthalten Zeiger auf nächstes Element
  - Terminierung der Liste durch NULL-Zeiger
  - Initiale Liste ist leer → Darstellung durch NULL-Zeiger
- Operationen
  - Insert – Erzeugen und Einfügen eines Elementes vor dem aktuellen Element
  - Append – Erzeugen und Einfügen eines Elementes am Listenende
  - Sort – Sortieren der Liste nach Werten der Listenelemente
  - Next – Gibt einen Zeiger auf das nächste Elemente zurück
  - Prev – Gibt einen Zeiger auf das vorherige Element zurück
  - Len – Gibt die Zahl der Elemente zurück
  - Delete – Entfernt das aktuelle Element aus der Liste

datastore.c

evlist.[ch]

Definition von Typen

- C-Konstrukt typedef erlaubt Deklaration neuer Typen
  - Eigentlich werden neue Namen für existente C-Typen vergeben
  - Es gelten die üblichen C Typkonvertierungsregeln
- Beispiele:
  - typedef int Length;
    - Definition eines Typs Length der synonym zu int verwendet werden kann
  - typedef char* String;
    - Definition eines Zeichenketten-Typs
    - Deklarierter Type steht „an Stelle" des Variablennamens
  - Verwendung wie der ursprüngliche C Typ

```
String p, lines[MAX], alloc( int );     /* Variablendefinitionen */
int strcmp( String, String );           /* Funktionsdeklaration  */
p = (String) malloc( 100 );             /* Platz für Zeichenkette mit Länge 99 */
```

typedef (contd.)

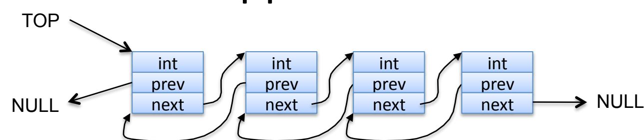
- Mit typedef werden Typvereinbarungen besser lesbar
  - für selbsreferentielle Strukturen

```
typedef struct tnode *treeptrT;          /* Deklaration */
typedef struct tnode {                   /* Definition   */
        char * word;
        int count;
        Treeptr left;
        Treeptr right;
} treenodeT;                             /* Neuer Typname */
```

  - Namenskonvention:
    - Typ mit T kennzeichnen, KamelSchreibWeise, Benutzung von _ underscore
    - Charles Simonyis Ungarische Notation für Variablennamen:
    - {Präfix} {Datentyp} {Bezeichner} –
      - hWndProc(HWND hW, UINT wMsg, WPARAM wParam, LPARAM lParam) /* Win 32 */
  - Funktionszeiger mit typedef vereinbaren

```
typedef int (*PFI) ( char *, char * );     /* Zeiger auf int Funktion mit 2 char* args */
PFI strcmp, numcp;
```


Doppelt verkettete Liste



- Wollen einen Stack von Integer-Zahlen implementieren
- Doppelt-verkettete Liste vereinfacht Navigation
- Operationen:
  - push (Stack, int);
  - int pop(Stack);
  - int top(Stack);
  - void swap (Stack);
  - Stack allocStack();
  - void freeStack();

```
typedef stackElemT * stackElemPtr;
typedef struct _stack {
        top stackElemPtr;
} * Stack;

struct stackElem {
        int val;
        stackElemPtr next;
        stackElemPtr prev;
} stackElemT;
```

teststack.c

istack.[ch]

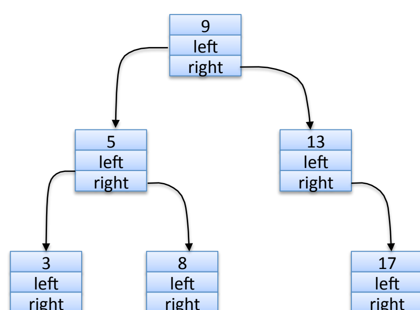
Fehlerbehandlung

- Globale Variable errno
  - Benutzt von vielen Unix-Systemaufrufen
  - Kann mit perror() aus der stdlib.h ausgewertet (gedruckt) werden
  - Problematisch bei multithreading
    - Unser istack.c benutzt ähnliches Konzept (error-Variable)
- Funktionen können Error-Code zurückgeben
  - Bspw. bei End-of-File (EOF-Zeichen)
  - Funktioniert, wenn im Rückgabetyp „Lücken“
- Abstrakte Datentypen
  - Datentyp sollte Vorkehrungen für Fehlermeldungen enthalten
  - Erweitern Stack-Struktur entsprechend
    - Problem: Nicht-initialisierte Stack-Struktur
    - Müssen globale-Fehlervariable und Stack-spezifische Variable testen

test2stack.c

i2stack.[ch]

Binärbäume



- Selbstreferentielle Strukturen
- Einfügen und Suche in O(log n)
- Linker Teilbaum:
  - Elemente mit kleinerem Wert
- Rechter Teilbaum:
  - Elemente mit größerem Wert
- Ausgabe der sortierten Werte
  - Traversieren des Baumes von links nach rechts
  - Tiefe-zuerst-Suche
  - Rekursiver Algorithmus
- Problem: Balanzierung

- Effiziente Implementierung von Mengen mit beliebiger Kardinalität

Unions

- Variable, die Objekte verschiedenen Typs enthalten kann
  - ...zu verschiedenen Zeitpunkten
  - Compiler kümmert sich um Größe und Ausrichtung entsprechend der Datentypen
  - Unabhängig von unterliegender Rechnerarchitektur
- Vergleichbar mit Rekord-mit-Variantenteil in Pascal
  - Aber: C unions haben keine Diskriminante
  - Benutzer muss wissen, welcher Datentyp in union gespeichert ist
  - Syntax ähnlich wie bei Strukturen

```
union u_tag {
   int ival;
   float fval;
   char * sval;
} u;
```

```
struct {
        char * name;
        int flags, utype;
        union {
                int ival;
                float fval;
                char * sval;
        } u;
} sym[NSYM];
```

```
if (sym[i].utype == INT)
        printf(”%d\n”, sym[i].u.ival );
else if (sym[i].utype == FLOAT)
        printf(”%f\n”, sym[i].u.fval );
else if (sym[i].utype == STRING)
        printf(”%s\n”, sym[i].u.sval);
else
        prinf(”bad type\n”);
```

  - Idee: Betten union und Typinformation in Struktur ein
    - Beispiel: Symboltabelle

Unions (contd.)

- Implementierung:
  - Union ist eine Struktur in der alle Elemente offset 0 haben
    - Elemente „liegen übereinander“
  - Groß genug um das „breiteste“ Element aufzunehmen
  - Alignment (Ausrichtung) passend für alle Elemente
    - Bspw. müssen Gleitkommazahlen auf Wortgrenzen beginnen
    - Unions können verwendet werden um Ausrichtung zu erzwingen
- Operationen
  - Zuweisung, Kopieren
  - Adresse nehmen (&)
  - Zugriff auf Elemente (->, .)
  - Kein Vergleich! → wie bei Strukturen
- Initialisierung
  - Nur möglich mit Werten vom Typ des ersten Elements
    - In unserem Beispiel: int

Bit Fields

- Repräsentation von Mengen in Integer-Zahl
  - Kompakte Informationscodierung
  - Manipulation von bit-Flaggen bei Gerätezugriffen, etc.
- Beispielproblem: Verarbeitung von C storage classes im Compiler
  - Bekannte Lösungsalternativen:
    - Präprozessor-Definitionen:
      ```
      # define KEYWORD     01
      # define EXTERNAL    02
      # define STATIC      04
      ```
    - Enumerations
      ```
      enum { KEYWORD = 01, EXTERNAL = 02, STATIC = 04 };
      ```
  - Verwendung:
    - `flags |= EXTERNAL | STATIC;` /* zwei Flaggen einschalten */
    - `flags &= ~(EXTERNAL | STATIC);` /* zwei Flaggen ausschalten */
    - `if ((flags & (EXTERNAL | STATIC)) == 0)...` /* true wenn beide Flaggen aus */

Bit Fields (contd.)

- Repräsentation der Flaggen als bit field
  ```
  struct {
      unsigned int is_keyword    : 1;
      unsigned int is_extern     : 1;
      unsigned int is_static  : 1;
  } flags;
  ```
  - Variable flags hat drei Felder mit 1-bit Weite
  - Zugriff auf Strukturelemente wie üblich:
    ```
    flags.is_extern = flags.is_static = 1;     /* Einschalten von zwei Flaggen */
    flags.is_extern = flags.is_static = 0;     /* Ausschalten von zwei Flaggen */
    if (flags.is_extern == 0 && flags.is_static == 0)...     /* Test auf zwei Flaggen */
    ```

→ Code wird besser lesbar
  - Feldelemente müssen keinen Namen haben
    - Unbenannte Felder dienen dem „padding" – bspw. um genau bit 7 ansprechen zu können
  - Implementierung ist maschinenabhängig (endianness)

Abstrakte Datentypen

- Ein Datentyp ist eine Mengen von Daten zusammen mit einer Familie von Operationen
- abstrakter Datentyp:
  - Beschrieben wird lediglich die Menge und die Semantik der Operationen, nicht aber die interne Repräsentation der Daten oder die Implementierung der Operationen
  - In C: Trennung Deklaration (.h) und Implementierung (.c)
- Problem in C:
  - Viele Implementierungsdetails (Speicherlayout von Strukturen) offengelegt
  - Lösung: Zeiger auf echten Datentyp (opaque)
  - Verwendung von Zugriffsfunktionen
- Idee:
  - Daten und Zugriffsfunktionen bündeln → Objekte
  - Verwenden Funktionszeiger um Methoden an Objekte zu binden

Datenkapselung
i2stack.h

Objektorientierung - Begriffe

- Klasse:
  - Beschreibung von Strukturen mit gleichen Eigenschaften (vergleichbar einem Datentyp mit wesentlich komplexeren Möglichkeiten);
  - eine Klasse zeichnet sich durch zwei Merkmale aus:
    - sie definiert eine Menge von Daten und kann Handlungen in Form von Methoden/ Elementfunktionen (das sind Prozeduren und Funktionen, die der Klasse angehören) ausführen.
- Instanz: (einer Klasse)
  - anschaulich: eine Variable ,,vom Typ Klasse``
- Objekt:
  - Instanz einer Klasse; eine Menge von Daten (definiert in der Klasse) mit den Eigenschaften (Funktionen, Zugriffsrechte...) der Klasse, der es angehört
- Vererbung:
  - Weitergabe von Klasseneigenschaften an andere Klassen; ein Objekt ,,erbt`` zusätzlich zu seinen eigenen Eigenschaften die einer anderen Klasse

OOP in C

- Objekte sind Zeiger auf Strukturen
  - (void *) – generische Repräsentation für alle Objekte
  - Objekte besitzen Zeiger auf ihre Klassen (-objekte)
- Instanzenvariablen
  - Sind Elemente der Objektstrukturen
- Klassenobjekte
  - Enthalten Zeiger auf Methoden
    - Organisiert als Zeiger auf Methodentabelle (mit Funktionszeigern)
  - Zugriffsfunktionen für Objekten
    - Instanzenmethoden – funktionieren nur bei Existenz einer Objektinstanz
    - Klassenmethoden – zur Objekterzeugung

Beispiel: String-Objekte

```
/*    ( str_oo.h )      */

   #ifndef __STR_OO_H
   #define __STR_OO_H

   /* String-Definition: abstrakte Objekte */

   void * newString    (const char * text);
   void   deleteString (void * _string);
   void   setvalue     (void * _string, const char * text);
   char * getvalue     (const void * _string);
   void   cout         (const void * _string);

   #endif
```

Deklaration verrät keine Details über Implementierung

Implementierung String-Objekte

```
/* str_oo.c */
   #include <stdio.h>
   #include "str_oo.h"
   #include "error.h"

/* String-Implementation abstrakte Objekte */
   static const char STR_Rep;
/* die Repraesentation der Klasse STRING, einfach eine Adresse */
   static const void * STRING = & STR_Rep;

/* die Definition der Objektstruktur */
   struct String {
     void * class;
     char * text;
   };
```

Das Klassenobjekt soll alle Methodenzeiger enthalten

Implementierung Klasse STRING

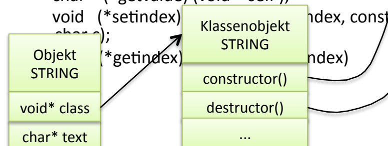
```
/* Definition der Klasse */
struct Class_Str {

/* die grundlegenden Methoden, fuer alle Klassen
     gleich */
     size_t size;
     void * (*constructor) (void * self, const int
      argnr, va_list * app);
     void * (*destructor)  (void *self);
     void * (*copy)       (const void * self);
     void   (*cout)       (const void *self);
/* ab hier stehen die benutzerdefinierten
      Methoden */

     void   (*setvalue) (void * self, char * chars);
     char * (*getvalue) (void * self );
     void   (*setindex)            ndex, const
            );
            (*getindex)            ndex)
```

```
/* Deklarationen der Methoden von STRING */
static void * String_constructor (void * self, const int argnr, va_list * app);
static void * String_destructor  (void * self);
static void * String_copy        (const void * self);
static void   String_setvalue    (void * self , char * chars);
static char * String_getvalue    (void * self);
static void   String_setindex    (void * self, int index, const char c);
static char   String_getindex    (void * self, int index);
static void   String_cout        (const void * self);

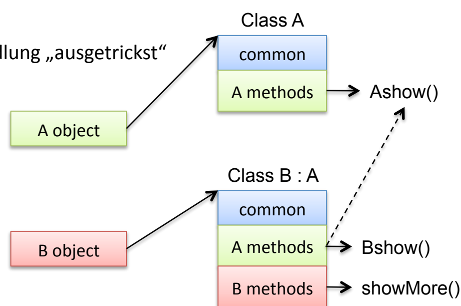
/* Initialisierung des Klassenobjekts */
static const struct Class_Str _String = {
      sizeof(struct String), String_constructor,
      String_destructor, String_copy,
      String_cout, String_setvalue,
      String_getvalue, String_setindex,
      String_getindex
};
 const void * STRING = & _String;
```



Klassenhierarchien

- Zugriff auf Methoden erfolgt über Klassenobjekte
- Idee:
  - Klassenobjekte beginnen immer gleich
    - (Konstruktor, Destruktor, etc....)
  - ...können aber durch weitere Methoden erweitert werden
    - Strukturen werden „vergrößert"
    - C Compiler wird durch Typumwandlung „ausgetrickst"

p->show()



- Viele Details:
  - Vererbung vs. Überschreiben
  - Kopieren von Objekten
  - Referenz-Typen
- All das führt letztlich zu C++

Zusammenfassung

- Grundlagen
  - Organisation von Daten; logisch zusammengehörende Variablen
- Strukturen und Funktionen
  - Strukturen fassen Elemente (verschiedenen) Typs zusammen
  - Können Argumente / Rückgabewerte von Funktionen sein
- Felder von Strukturen
  - Fassen mehrere strukturierte Datenobjekte zusammen
- Zeiger auf Strukturen
  - Spezielle Syntax für Zugriff auf Strukturelement: p->next ←→(*p).next;
- Selbstreferentielle Strukturen
  - Grundlage effizienter Datenstrukturen beliebiger Kardinalität
  - typedef – Deklarationen werden besser lesbar
  - unions, bit-fields – Spezialfälle (generische Datentypen, Kompakt)
- Objekte
  - Datenkapselung, Vererbung, Polymorphie – durch Zeiger auf Strukturen darstellbar