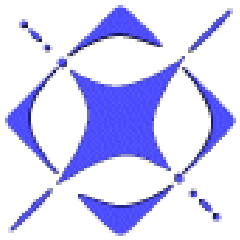


Programmieren in C

Speicher anfordern, Unions und Bitfelder

Prof. Dr. Nikolaus Wulff

Vergleich: Felder und Strukturen

- Felder müssen Elemente vom selben Typ enthalten.
- Strukturen können Elemente unterschiedlichen Typs enthalten.
- Felder werden an Methoden immer als Adressen übergeben, um das Kopieren zu vermeiden.
- Strukturen können per Werte- oder Zeigersemantik an Methoden übergeben werden:

```
void callByValue(struct color c);
void callByReference(struct color *c);

struct color farbe;
callByValue(farbe);
callByReference(&farbe);
```

Zugriff auf die Strukturelemente

```
struct color farbe, *ptrFarbe;
```

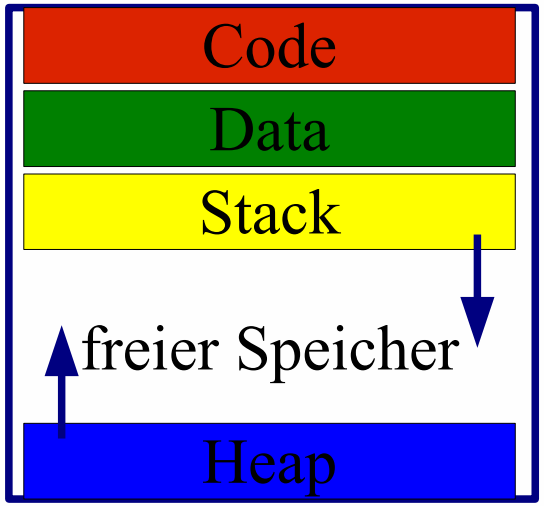
- Der Zugriff auf die Elemente einer Struktur erfolgt mit dem „Punkt“-Operator:
  - farbe.b = 128
- Ausser es liegt ein Zeiger auf die Struktur vor, dann muss dieser dereferenziert werden:
  - (*ptrFarbe).b = 128
- alternativ kennt C den „Pfeil“-Operator:
  - ptrFarbe->b = 128

Dynamische Speicherverwaltung

- Bisher wurden lokale Variablen verwendet, die C Umgebung sorgte sich um die Speicherverwaltung.
- C bietet die Möglichkeit Speicher direkt vom Betriebsystem anzufordern.
- Der Programmierer muss sich dann selbständig um die Größe, die Initialisierung und die Freigabe des Speichers kümmern.
- Hierzu dienen die Funktionen ***malloc*** (Memory Allocation) und ***free***, sowie der **`sizeof`** Operator.
- Diese Funktionen sind in `<stdlib.h>` deklariert.

Aufteilung des Hauptspeichers

- Das Betriebsystem teilt den Speicher in vier Bereiche auf:
  - Code: Maschinencode
  - Data: Globale und konstante Daten
  - Stack: Lokale Variablen, Rücksprungadressen
  - Heap: Dynamisch reservierter Speicher



Speicher anfordern mit malloc

- malloc hat die Signatur:

```
void* malloc(size_t numbytes);
```

- size_t ist ein per typedef definierter eigener Datentyp (meist unsigned int) der die Größe des Speicherbereichs angibt.
- Es wird ein Zeiger auf void geliefert, da das Betriebsystem nicht „wissen“ kann was für Daten verwaltet werden sollen.
- Dieser Zeiger muss dann auf den passenden Datentypen „gecasted“ werden.

Malloc anwenden

- Es soll ein z.B. ein Feld von 6 double Zahlen alloziiert werden => 6*8 Byte = 48 Byte.
- Um die Größe eines Datentypen zu ermitteln wird der `sizeof` Operator verwendet, dieser ermittelt auch bei Strukturen die richtige Größe.

```
double *array;
size_t bytes = 6*sizeof(double);

array = (double *)malloc(bytes);
array[0] = 3.14;
  ....

free(array);
```

Speicher für Strukturen

- Ein Feld von vier Farben soll angelegt werden:

```
typedef struct color {
    char r,g,b;
} *Color;

size_t bytes = 4*sizeof(struct color);
Color c, feld;

feld = (Color)malloc(bytes);

c = feld;
c->b = 255;
c++;
 ....

free(feld);
```

C Union

- C kennt einen weiteren selbstdefinierten Datentyp union, der leicht mit C struct's zu verwechseln ist.
- Die Deklaration sieht sehr ähnlich aus:

```
union <identifier> {
   type1 <identifier1>;
                ...       ;
   typeN <identifierN>;
};
```

- Im Gegensatz zu einem struct wird jedoch nur Speicherplatz für einen der Typen `type1 ...typeN` bereitgestellt.

Union Beispiel

```
union numeric {
   int    i;
   long   l;
   float  f;
   double d;
};

union numeric X;
X.f = 3.14;
printf("X.f = %f \n",X.f);
printf("X.i = %d \n",X.i);
printf("X.d = %lf\n",X.d);
printf("X.l = %ld\n",X.l);
```

- `numeric` kann zur Laufzeit genau einen der Datentypen int, long, float oder double enthalten.
- Es liegt am Programmierer den *„richtigen Wert“* passend zum Typ ein- und auszulesen...

Color ADT per Union

- Der Color ADT wurde entweder als ein int

```
typedef int Color;
```

- oder als ein (rgb)-Triple mit Hilfe einer Struktur definiert:

```
struct color_struct {
    unsigned char r;
    unsigned char g;
    unsigned char b;
};
typedef struct color_struct Color;
```

- Im erstem Fall wurde der Zugriff auf eine Farbe aufwendig mit Bitshift Operationen realisiert.

Color ADT per Union

```
struct color_struct {
    char bw,r,g,b;
};

typedef union color_union {
   int value;
   struct color_struct comp;
} Color;
```

- Eine Union ermöglicht die Verwendung beider Möglichkeiten.

```
Color col;

col.value = 127;
printf("Color red = %d\n",col.comp.r);
```

struct und union

- Häufig wird zur Unterscheidung des Union Typs dieser in eine Struktur eingebettet:

```
typedef enum {RECT, ELIPSE,
              TRIANGLE} ShapeType;

typedef struct shape_struct {
      ShapeType type;
      union {
        struct Rectangle r;
        struct Elipse    e;
        struct Triangle  t;
      } shape;
} Shape;
```

Shape Rotation

```
void rotateShape(Shape *aShape, int deg) {
    switch(aShape->type) {
    case RECT:
       rotateRect(&aShape->shape.r,deg);
       break;
    case ELIPSE:
       rotateElipse(&aShape->shape.e,deg);
       break;
        ...
```

- Mit Hilfe des Typ-Feldes lässt sich in die entsprechenden Unterroutinen für den jeweiligen geometrischen Typen verzweigen...

Frohe Weihnachten und einen guten Start in das neue Jahr 2011!

Bit-Masken

- Häufig werden mehrere einzelne Werte innerhalb einer Variablen gespeichert, meist mit Bit-Masken:

```
#define KEYWORD  01
#define EXTERNAL 02
#define STATIC   04
```

- oder per Enumeration:

```
enum {KEYWORD=01, EXTERNAL=02,
      STATIC=04};
```

- Setzen einzelner Flags:

```
    int flag;
    flag |= STATIC | EXTERNAL;
```

Anwendung von Bit-Masken

- Entsprechend das Löschen einzelner Felder:

```
flag &= ~(STATIC | KEYWORD);
```

- und die Bedingung

```
if((flag & (KEYWORD | STATIC)) == 0) {
```

- ist genau dann erfüllt wenn beide Felder gelöscht sind.
- Wesentlich schöner wäre es, wenn auf die Felder innerhalb der Variablen `flag` direkt zugegriffen werden kann...

Bitfields

- Eine spezielle Form von Strukturen sind Bitfelder.
- Bei Ihnen wird hinter der Deklaration der einzelnen Elemente explizit der Speicherbedarf der Elemente in Bit angegeben:

```
struct {
    unsigned int is_keyword : 1;
    unsigned int is_extern  : 1;
    unsigned int is_static  : 1;
} flags;
```

- `flags` ist eine Variable, die drei Einzel-Bit-Felder enthält.

Programmsteuerung per Bitfeld

- Die Bitfelder lassen sich jetzt direkt per Namen ansprechen, lesen und setzen:

```
flags.is_keyword = flags.is_static = 1;

if ( flags.is_keyword == 0 &&
     flags.is_static == 0) {
```

- Bitfelder haben keine Adresse, daher ist auch der Adress-Operator **&** nicht erlaubt, und es gibt somit keine Zeiger und Arrays von ihnen.
- Die Elemente müssen immer vom Typ int sein.
- Bitfelder sind Maschinen- und Compilerabhängig, der Code ist meist schwieriger zu portieren.

Hardwarenahe Programmierung

- Die Register der CPU lassen sich gut mit Hilfe von Union-Struct Kombinationen ein- und auslesen.

```
struct register_8bit {
    short al,ah;
};

struct register_16bit {
    union {
        int ax;
        struct register_8bit r;
    };
} reg;

reg.ax   = 0;
reg.r.ah = 1;
reg.r.al = 64;
```

Hardwarenahe Programmierung

- Meist werden die Typen int und short per `typedef` durch die Bezeichner Byte, Word, DWord etc. ersetzt...
- Mit Bitfeldern lassen sich die einzelnen 1-Bit Felder des Statusregisters als boolsche Variablen verwenden:

```
struct status_register {
    int overflow  : 1;
    int direction : 1;
    int interrupt : 1;
        ....
    int parity    : 1;
    int carry_flag: 1;
};
```