

# **C-Kurs**

# **Dynamische**

# **Speicherverwaltung**

# Speicher – Abstraktion

## □ Speicher:

**Virtueller Speicher: Ein Bytearray**

## □ Programmsicht:

- Jedes Programm hat seinen eigenen Speicher
- Es hat eine „unbegrenzte Speichermenge“
- Der Zugriff auf alle Speicherbereiche ist gleich schnell, ...

# Speicher – Realität

## □ Speicher:

**Virtueller Speicher: Ein Bytearray**

## □ Realität:

### ➤ Kein unbegrenzter physikalischer Speicher

- Alle Programme teilen sich den selben physikalischen Speicher
- Speicher wird durch das Betriebssystem allokiert und verwaltet
- Viele Anwendungen sind speicherdominiert
- Es gibt eine Speicherhierarchie: Cache, RAM, Platte

## □ Speicherzugriffsfehler sind besonders problematisch

### ➤ Effekte sind weit von der Ursache entfernt

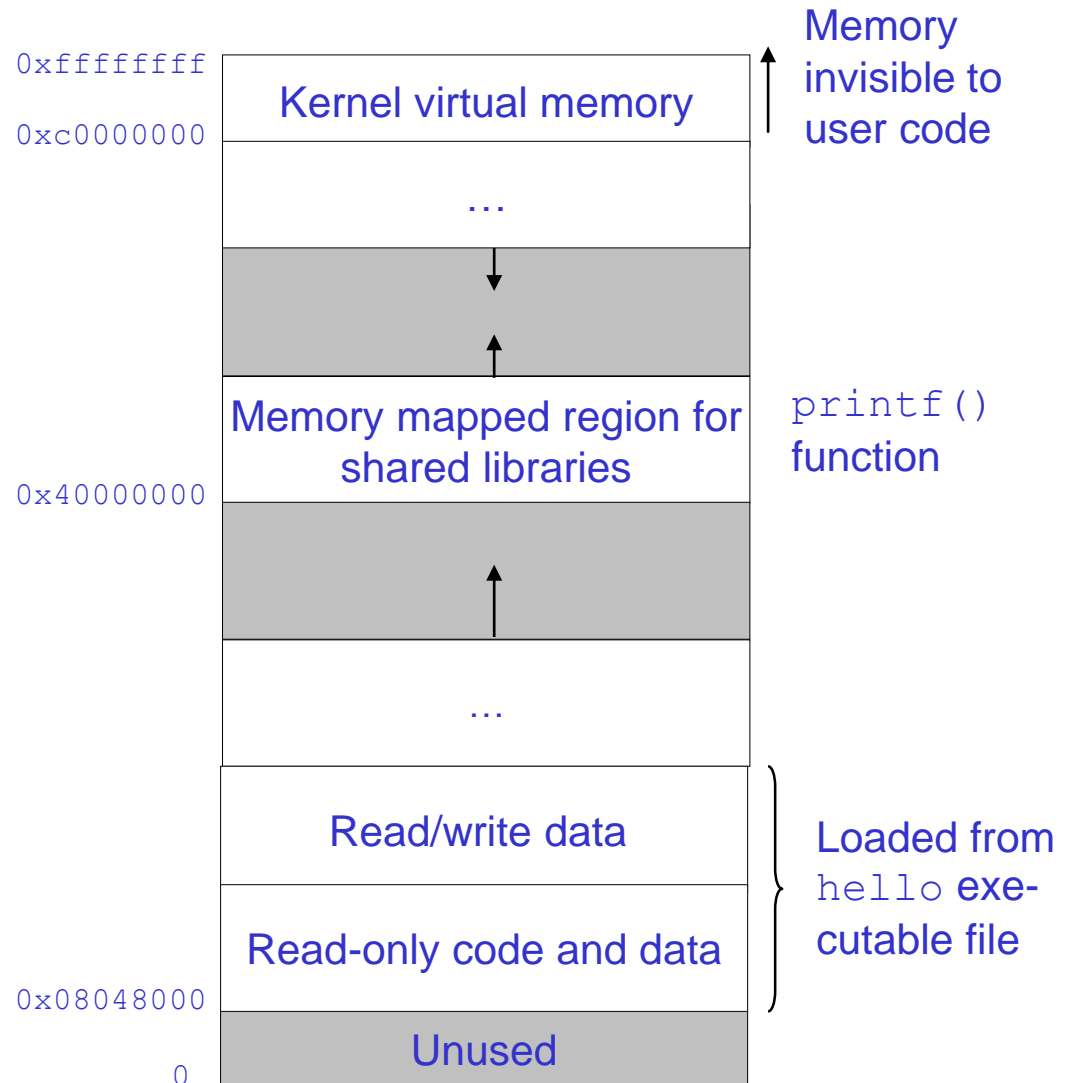
# Speicher und C-Programme

# Speicher und C-Programme

- ❑ Prinzip: Zuteilung von Speicher nach Bedarf, da begrenzte Ressource
- ❑ Programmkomponenten die Speicher brauchen
  - Der ausführbare C-Code – das Programm
  - C-Bibliotheken und externe Funktionen, z.B. printf
  - ...

# Speicher und C-Programme

## Physikalischer Speicher

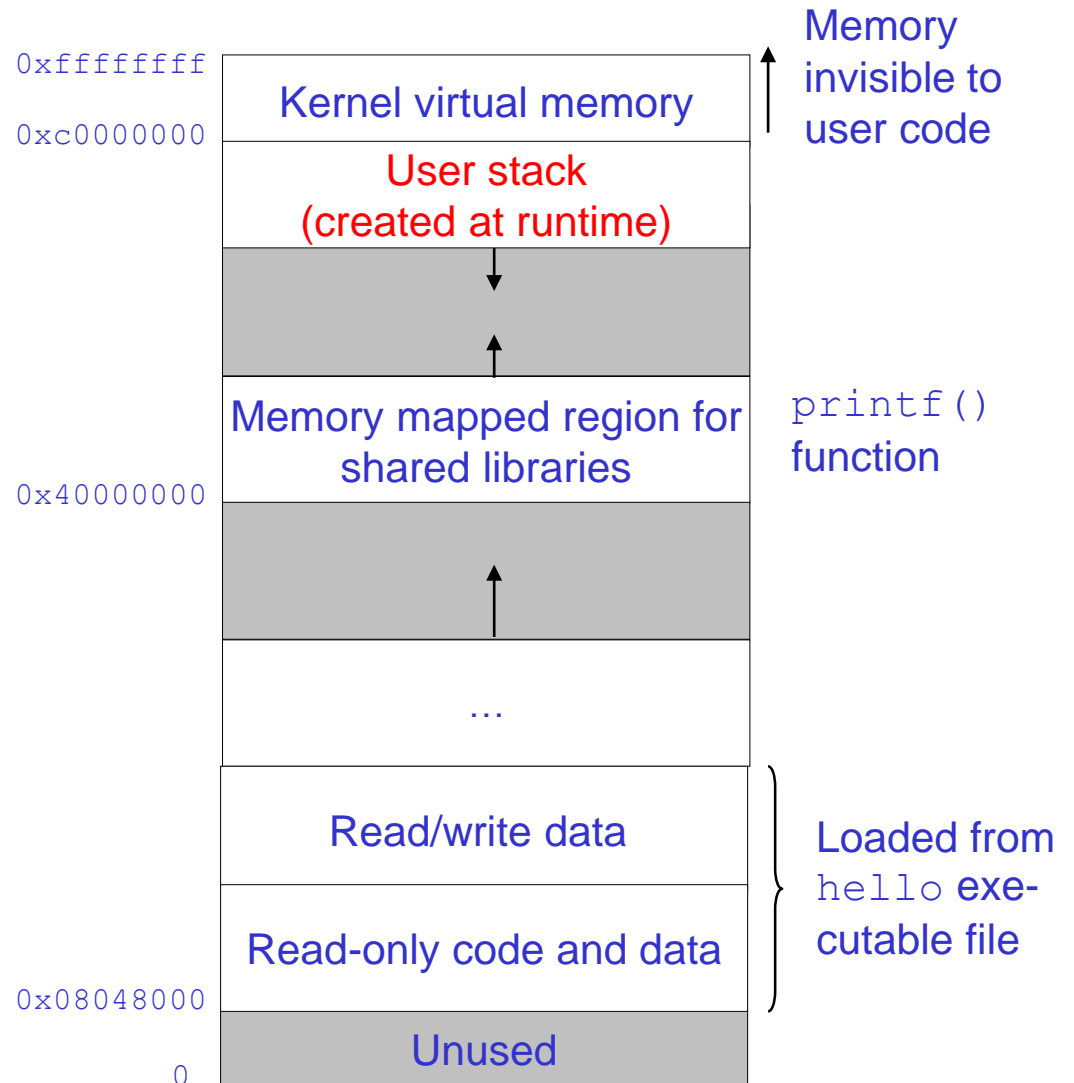


# C-Speicher – implizit

- ❑ Prinzip: Zuteilung von Speicher nach Bedarf, da begrenzte Ressource
- ❑ Programmkomponenten die Speicher brauchen
  - C-Code selber
  - C-Bibliotheken und externe Funktionen, z.B. printf
  - Implizit für C-Variablen, C-Arrays und C-Funktionen
  - ...
- ❑ Problematik
  - Speicher für C-Funktionen unbekannt vor Programmaufruf
  - Warum: Funktionsaufrufreihenfolge unbekannt
- ❑ Konzept
  - Speichern der Variablen, etc in einem dynamisch wachsenden Datenstruktur (Hier Stack – mehr zu der Datenstruktur Stack später)

# Speicher und C-Programme

## Physikalischer Speicher



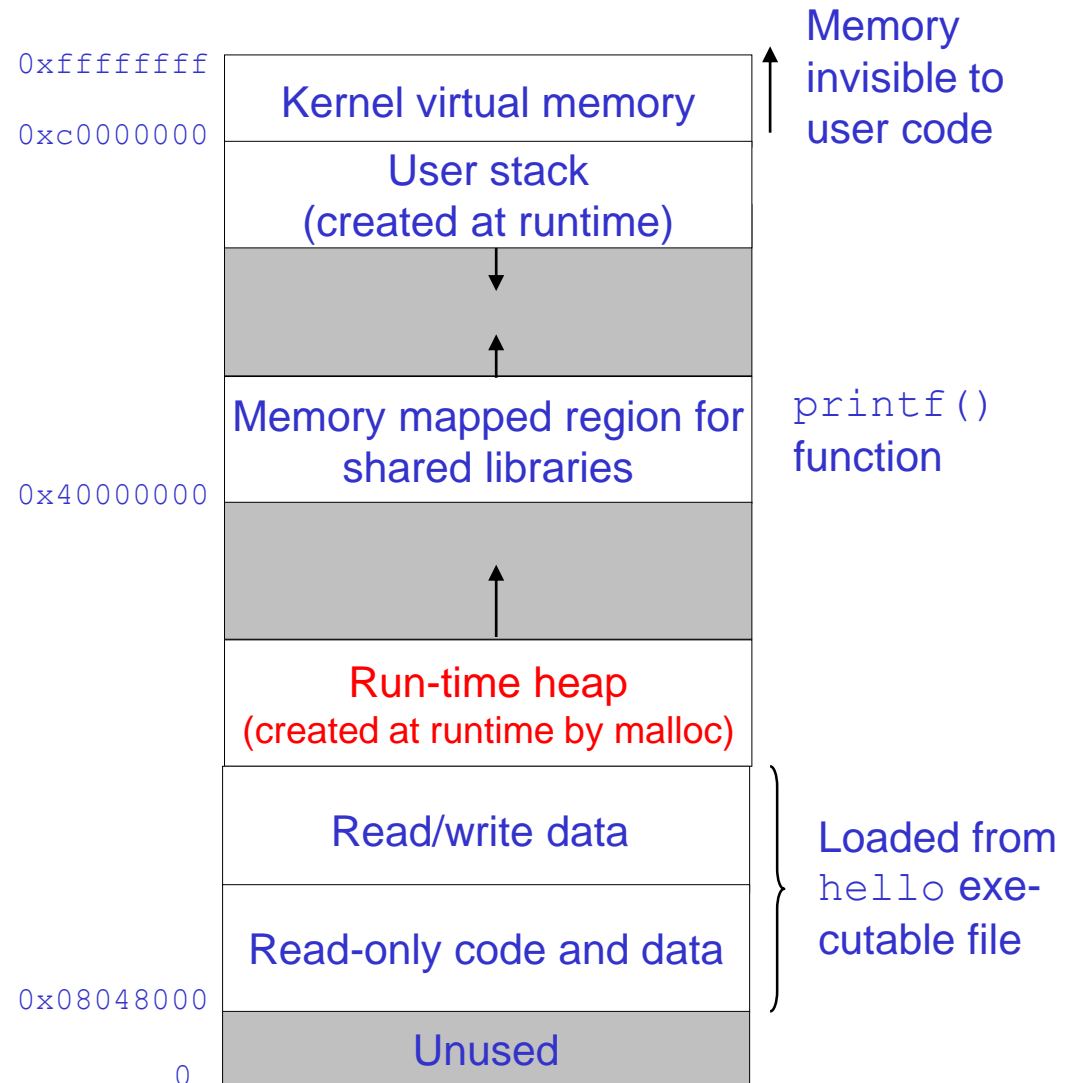


# C-Speicher – explizit

- ❑ Prinzip: Zuteilung von Speicher nach Bedarf, da begrenzte Ressource
- ❑ Programmkomponenten die Speicher brauchen
  - C-Code selber
  - C-Bibliotheken und externe Funktionen, z.B. printf
  - Implizit für C-Variablen und C-Funktionen
  - **Explizit für C-Variablen, wenn die benötigte Menge Speicher von Parametern abhängig ist**
- ❑ Problematik
  - Speicherbedarf für C-Variablen unbekannt vor Programmaufruf
  - Warum: Anforderungen unbekannt
- ❑ Konzept
  - **Speichern der Variablen, etc. in einer weiteren dynamisch wachsenden Datenstruktur** (einem Heap – mehr zu der Datenstruktur Heap später)

# Speicher und C-Programme

## Physikalischer Speicher



# Laufzeit vs. Compilezeit

## □ Compilezeit

- Während des Compilieren d.h.: Übersetzen des C-Codes in Assemblercode

## □ Laufzeit

- Während der Ausführung eines compilierten Programms

## □ Beispiele:

- |  |                           |
|--|---------------------------|
| ➤ Welche Funktionen existieren:        | Bekannt zur Compilezeit   |
| ➤ Wie häufig eine Funktion ausgeführt: | idR: Bekannt zur Laufzeit |
| ➤ Mit welchen Funktionsparametern:     | idR: Bekannt zur Laufzeit |

# Dynamische Speicherallokation in C

# Dynamische Speicherallokation

- ❑ Prinzip: Zuteilung von Speicher nach Bedarf, da begrenzte Ressource
- ❑ Zwei Varianten der Speicherverwaltung
  - Implizit
    - C Variablen

```
int a[100]; // Die benötigte Menge Speicher  
           // steht zur Compilezeit fest
```
  - **Explizit**, z.B: für ein Array mit zur Compilezeit unbekannter Länge,  
welches abhängig von den Eingabedaten ist
    - C Speicherverwaltung: **malloc** und **free**

# Memory ALLOCation

C: `malloc`

# Dynamische Speicherverwaltung: malloc

- ❑ `#include <stdlib.h>`

- ❑ `void *malloc(size_t size)`

- Rückgabe:

Zeiger auf Speicherblock der wenigstens die Größe `size` Bytes hat. Alignment (typischerweise) auf 8-Byte Grenzen

- Speicher wird nicht (mit 0) initialisiert

# Einschub: Rückgabewerte im Kontext der Fehlerbehandlung

## ❑ Motivation:

- In jeder Funktion können Fehler auftreten
- Wie werden diese an die aufrufende Funktion zurückgemeldet?
- Z.B: Speicher ist endlich: malloc ist nicht in der Lage die gewünschte Speichermenge zu allokalieren

## ❑ Idee – Nutzen von Rückgabewerten

## ❑ Falls Fehler in der Funktion auftritt:

- Explizite Rückgabe eines bestimmten Wertes
- Setzen eines Fehlercodes in der globalen Variable `errno`
- Nutzen einer Hilfsfunktion `perror`. Um diesen Fehlercode und die Fehlermeldung auf der Konsole (genauer `stderr`) auszugeben



# Einschub: Fehlerbehandlung

## ❑ Motivation:

- In jeder Funktion können Fehler auftreten
- Gewisse Fehler sollen zum Abbruch des Programms führen

## ❑ Idee – Nutzen der Abbruchfunktion `int exit()`

## ❑ Falls Fehler in einer Unterfunktion auftritt:

- Erst Fehleranalyse
- Dann Abbruch des Programms mittels `exit`
  - Argumentwert  $> 0$
- Bricht das Programm vollständig ab
- Erfolgreiche Ausführung eines Programms gibt den Wert 0 zurück (Beachte: `int main()` )

# Malloc: Beispiel ohne und mit Fehlerbehandlung

```
char *foo(int n){                                     // not to be used!
    char *p;
    // allocate a block of n bytes
    p=(char *) malloc(n);
    return p;
}
```

```
char *foo-with-error-handling(int n){ // to be used
    char *p;
    // allocate a block of n bytes
    if ((p=(char *) malloc(n)) == NULL){
        perror("malloc failed while allocating n chars");
        exit(1);
    }
    return p;
}
```

# Malloc: Beispiel mit Fehlerbehandlung

```
char *foo-with-error-handling(int n){ // to be used
    char *p;
    // allocate a block of n bytes
    // if ((p=(char *) malloc(n)) == NULL){

    p=(char *) malloc(n);
    if (p == NULL){

        perror("malloc failed while allocating n char");
        exit(1);
    }
    return p;
}
```

# Beispiel: möglicher Speicherzugriffsfehler

```
char *foo(int n){                                     // not to be used!
    char *p;
    // allocate a block of n bytes
    p=(char *) malloc(n);
    return p;
}
```

- ❑ Fehlerbehandlung wichtig, weil sonst Speicherzugriffsfehler möglich sind
- ❑ Hier ein Zugriff auf ein nicht allokiertes Array
- ❑ Zugriff auf „Nullpointer“ => Core dump

# Einschub: Rückgabewerte

## Hier zwei C-Konventionen

### ❑ Funktion hat eigentlich keinen Rückgabewert

- Z.B.: `int add (int *sum, int a, int b)`
- Alles OK           => Rückgabe des Wertes 0
- Fehler             => Rückgabe eines Wertes != 0

### ❑ Funktion hat einen Rückgabewert

- Z.B.: `void *malloc(size_t size)`
- Alles OK           => Rückgabe eines Wertes != 0
- Fehler             => Rückgabe des Wertes `NULL` == 0

### ❑ Zusätzlich: Setzen des Fehlercodes in `errno`.

- Nutzen einer Hilfsfunktion `perror`. Um diesen Fehlercode und die Fehlermeldung auf der Konsole (genauer `stderr`) auszugeben

# Dynamische Speicherverwaltung: malloc

❑ `#include <stdlib.h>`

❑ `void *malloc(size_t size)`

➤ Falls erfolgreich:

- Rückgabe: Zeiger auf Speicherblock der wenigstens die Größe `size` Bytes hat. Alignment (typischerweise) auf 8-Byte Grenzen
- Falls `size == 0`, Rückgabewert `NULL`

➤ Falls nicht erfolgreich: Rückgabe `NULL (0)` und setzen von `errno`.

➤ Speicher wird nicht (mit 0) initialisiert

❑ `void perror(msg)`

➤ Gibt die letzte Systemfehlermeldung auf der Konsole (genauer `stderr`) aus

# Einschub: Was ist void?

## □ Motivation:

- Viele Funktionen geben nichts zurück.
- Z.B: `void print_hello () {  
    printf(„hello world\n“);  
}`

□ Problem – C-Syntax verlangt, dass jede Funktion einen Rückgabewert hat

□ Idee – Nutzen eines „generischen Typ“: `void`

# Einschub: Was ist void \*?

## □ Motivation:

- Viele Funktionen interessiert es nicht, ob sie einen Pointer auf int oder auf double, oder sonst einen Datentyp bekommen oder zurückgeben.
- Wichtig ist, dass es ein Pointer ist.

## □ Idee – Nutzen eines „generischen Pointertyps“: `void *`

## □ Beispiel: `void *malloc(size_t size)`

## □ Kann dann in den gewünschten Typ umgewandelt („casting“) werden:

- `char *p-char = (char *) p = *malloc(12);`
- `int *p-int = (int *) p = *malloc(12);`
- `float *p-float = (float *) p = *malloc(12);`



# Dynamische Speicherverwaltung: free

- ❑ `#include <stdlib.h>`

- ❑ `void free(void *p)`

- Der Block auf den `p` zeigt wird an den verfügbaren Speicherpool gegeben
- `p` Resultat eines vorherigen Aufrufes von `malloc` oder `realloc`

- ❑ **Hinweise: Es gibt in C keine Garbagecollection!**

- Speicher muss explizit freigegeben werden!
- Speicher wird nicht automatisch auf 0 gesetzt

# Bestimmung der Speichergrößen

- ❑ Operator: `sizeof`
- ❑ Ermittelt Größe von Typ / Variablen in Bytes
- ❑ Beispiel:  
`long l;`  
`sl = sizeof(l);`  
`sd = sizeof(double);`
- ❑ Beispiel: Sun Sparc 32 Bit

```
char 1 Byte  
short 2, int 4, long 4, long long 8 Bytes  
float 4, double 8, long double 16 Bytes  
pointer 4 Bytes
```

## □ Größen von C Objekten (in Bytes)

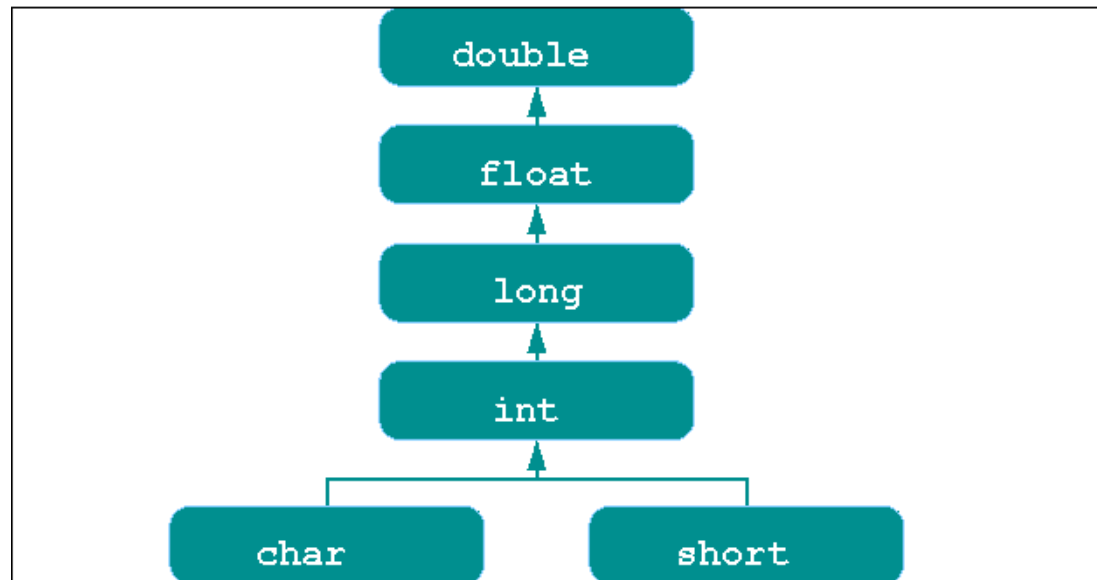
C Data Typ	Typical 64-bit	Typical 32-bit	Intel IA32
▪ int	4	4	4
▪ long int	8	4	4
▪ char	1	1	1
▪ short	2	2	2
▪ float	4	4	4
▪ double	8	8	8
▪ long double	8	8	10/12
▪ char *	8	4	4
▪ Oder jeder andere Pointer	8	4	4

## □ Operator `sizeof()`

Ermittelt Größe vom Typ / Variablen in Bytes

# Typumwandlung in Ausdrücken

- Sind unterschiedliche Typen in Ausdrücken enthalten wird eine implizite Typumwandlung gemacht



- Explizite Typumwandlung mittels Casting:  
`(type) expr`

# Malloc/free: Beispiel

```
void foo(int n){
    int i, *p;

    /* allocate a block of n ints */
    if ((p=(int *) malloc(n*sizeof(int))) == NULL){
        perror("malloc failed when allocating n ints");
        exit(1);
    }
    for (i = 0; i < n; i++){
        p[i] = i;
    }
    free(p); /* return p to available memory pool */
}
```

# Dynamische Speicherverwaltung: calloc

❑ `#include <stdlib.h>`

❑ `void *calloc(size_t n, size_t size)`

➤ Falls erfolgreich:

- Rückgabe: Zeiger auf Speicherblock der wenigstens die Größe `n * size` Bytes hat. Alignment (typischerweise) auf 8-Byte Grenzen

➤ Falls nicht erfolgreich: Rückgabe `NULL (0)` und setzen von `errno`

➤ Speicher wird mit 0 initialisiert

# Arrays und Pointer

# Arrays und Pointer

## □ Bekannt:

Arrays und Pointer werden in C ganz ähnlich behandelt

## □ Wesentlichster Unterschied:

➤ Arrays haben eine feste Dimension

⇒ Ihnen ist ein fester Speicherort zugeordnet

⇒ Für die zu speichernden Objekte / Arrayelemente ist Platz reserviert

➤ Zeiger/Pointer weisen erst nach Zuweisung oder dyn. Allokation auf den Speicherort ihrer Objekte

□ Arrayvariable      ⇒      Adresse des 1. Elements

□ Arrayindizes      ⇒      Offset im Array



# Arrays und Pointer

- ❑ Konstante Dimension/Größeangabe von Arrays

```
double df[100]; /* Array mit 100 Elementen */
```

- ❑ Variable Dimensionierung von Arrays nur für **lokale/automatische** Arrayvariable zulässig

```
void fun(int n) {  
    double df[n]; /* Array mit n Elementen */  
    ...  
}
```

- ❑ Grund: Arraygröße muss beim Anlegen / bei Speicherzuweisung des Arrays bekannt sein

➤ Statisch / global	⇒	Compile-Zeit
➤ Automatisch / lokal	⇒	Eintritt in Funktion / Block

# Arrays und Pointer

- ❑ Variable Dimensionierung von Arrays wird häufig benötigt
- ❑ Lösung  $\Rightarrow$  dynamische Arrayallokation
- ❑ Beispiel: **double**-Array dynamisch duplizieren

```
double *dbldup(double d[], int n) {  
    double *df;  
    int i;  
  
    df = malloc(n * sizeof(double));  
    for(i = 0; i < n; i++) {  
        df[i] = d[i];  
    }  
    return(df);  
}
```

# Arrays und Pointer

- Arraynamen sind eigentlich Pointer, zeigen auf das erste Element im Array

```
int i, *ip, ia[4] = {11, 22, 33, 44};  
ip = ia;
```

# Arrays und Pointer

- Arraynamen sind eigentlich Pointer, zeigen auf das erste Element im Array

```
int i, *ip, ia[4] = {11, 22, 33, 44};  
ip = ia;  
i = *++ip;
```

- Ähnlichkeit von Arrays und Zeigern
  - ⇒ macht die Pointerarithmetik möglich
  - ⇒ Pointerarithmetik mächtig, aber oft unübersichtlich

# Arrays und Pointer

- Arraynamen sind eigentlich Pointer, zeigen auf das erste Element im Array

```
int i, *ip, ia[4] = {11, 22, 33, 44};  
ip = ia;  
i = *++ip;
```

- Ähnlichkeit von Arrays und Zeigern
  - ⇒ macht die Pointerarithmetik möglich
  - ⇒ Pointerarithmetik mächtig, aber oft unübersichtlich
- Arrayindizes sind Offsets == Abstand zum Arrayanfang  

```
ia[3]    ⇒    *(&ia[0] + 3)
```
- Hinweis: Pointerarithmetik ist mit Vorsicht zu genießen und hier nur der Vollständigkeit halber erwähnt.