Teil 6: Strukturen und Unionen, Dynamische Speicherverwaltung

Gliederung

Strukturen

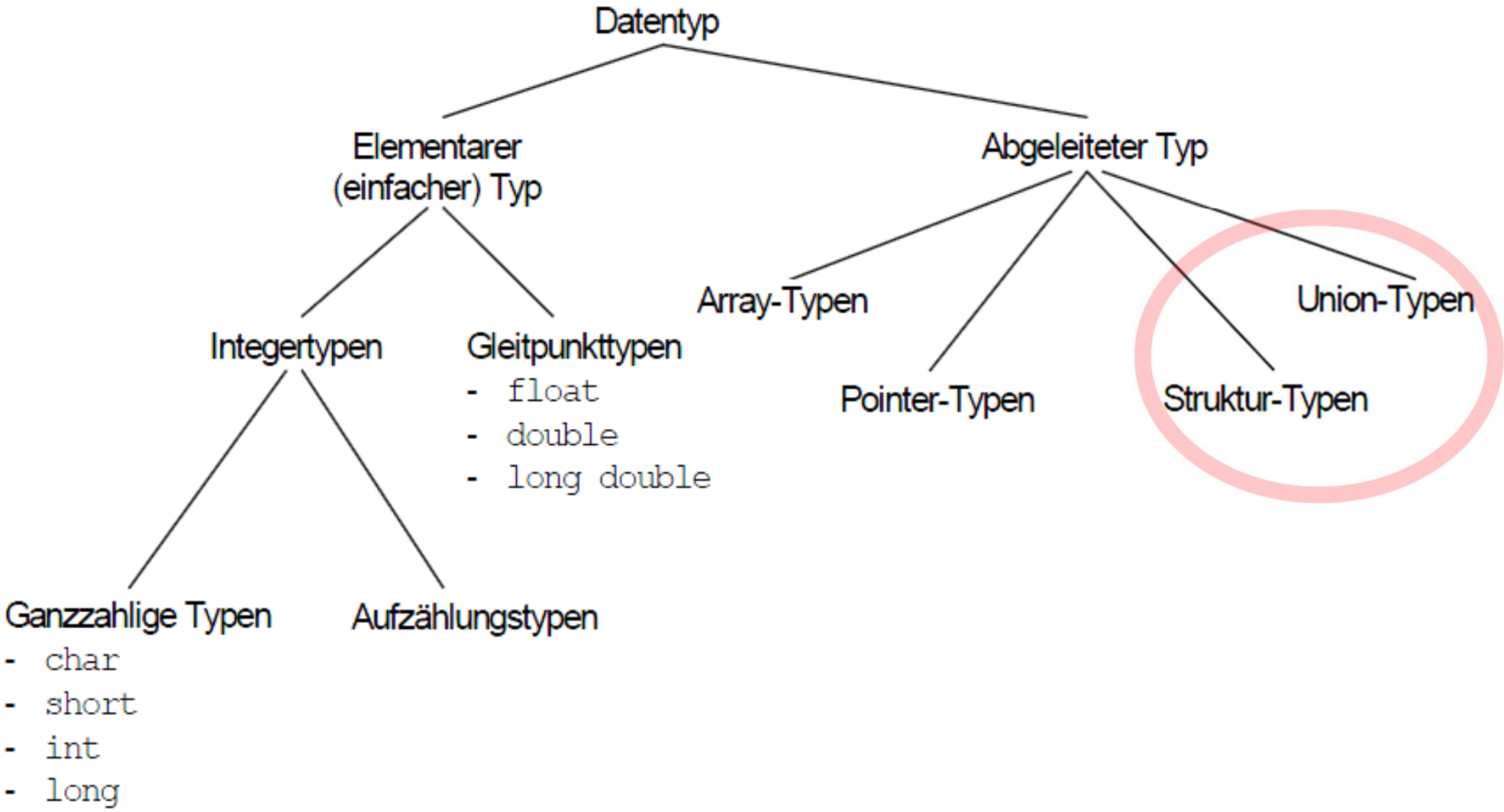
Typdefinitionen

Unionen

Dynamische Speicherverwaltung

Strukturen

Typ-Klassifikation



Strukturen

- Ursprung in Pascal: **record, Verbunddatentyp**
- In C: **struct , Strukturtyp**
- kann Komponenten (Variablen) verschiedener Typen enthalten:

Personal-nummer	Nachname	Vorname	Straße	Haus-nummer	Postleit-zahl	Wohnort	Gehalt
`int`	`char[20]`	`char[20]`	`char[20]`	`int`	`int`	`char[20]`	`float`

- Anwendung: Gruppierung logisch zusammenhängender Daten
  - z. B. Datum, Adresse, Personendaten, geometrische Objekte, …
- Behandlung beim Lesen und Schreiben immer als als Einheit

Strukturtyp-Definition

- Anzahl und Typ der Komponenten sind frei definierbar:

```
struct Name
{
    typ1 komponente_1;
    typ2 komponente_2;
    ...
    typN komponente_n;
};
```

- der Typname ist

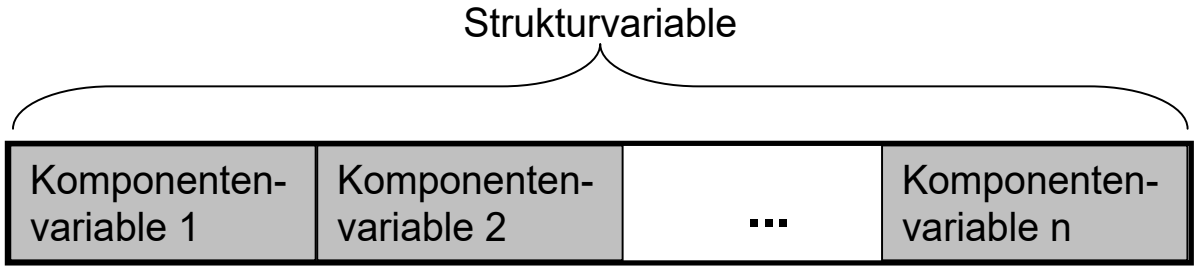
```
struct Name
```

- Erstellung einer Variablen

```
struct Name variablenname;
```

Strukturvariablen

- Variable eines Strukturtyps = aus Komponentenvariablen des Strukturtyps zusammengesetzte Variable



Beispiel-Definition

```
struct Adresse
{
   char  strasse[20];
   int   hausnummer;
   int   postleitzahl;
   char  stadt[20];
};

struct Student
{
   int             matrikelnummer;
   char            name[20];
   char            vorname[20];
   struct Adresse  wohnort;
};

struct Student meyer, mueller;  // 2 Studenten
struct Student kurs[30];        // Feld mit 30 Studenten
```

Definition mit Initialisierung

```
struct Student meyer = {
                           66202,
                           "Meyer",
                           "Herbert",
                           {
                              "Schillerplatz",
                              20,
                              73730,
                              "Esslingen"
                           }
                       };
```

Übergabe an Funktionen

- *call-by-value* (wie bei elementaren Datentypen **`int`**, **`float`**, `etc.`)

Zugriff auf Komponentenvariablen

```
meyer.matrikelnummer = 716347;
strcpy(meyer.name, "Meyer");

meyer.wohnort.postleitzahl = 73733;
strcpy(meyer.wohnort.stadt, "Esslingen");

printf("%6d %5d %s\n", meyer.matrikelnummer,
                       meyer.wohnort.postleitzahl,
                       meyer.name);
```

Kombination von Typ- und Variablendefinition

- ohne Strukturbezeichner/Etikett (**unüblich**)

```
struct
{
   float x;
   float y;
} punkt1, punkt2, punkt3;
```

keine weiteren Variablendefinitionen möglich

- mit Strukturbezeichner/Etikett (**üblich**)

```
struct Koordinaten
{
   float x;
   float y;
} punkt1, punkt2, punkt3;

struct Koordinaten punkt4, punkt5, punkt6;
```

weitere Variablendefinitionen möglich

Struktur-Zuweisungen, Vergleich, Größe

- Direkte Zuweisung möglich, alle Komponentenwerte der Variablen `punktA` werden denen der Variablen `punktB` zugewiesen

```
struct Koordinaten punktA = {1.5, 3.0}, punktB;
punktB = punktA;
```

- **ABER**: Prüfung auf Gleichheit `if (punktA == punktB)` ist **nicht** möglich!

- Bestimmung der Größe in Bytes einer Strukturvariablen:

```
sizeof(struct Koordinaten)
```

Punktoperator . und Pfeiloperator ->

```
struct Koordinaten
{
   float x;
   float y;
};

struct Koordinaten punktA = {1.5, 3.0};
struct Koordinaten *pA;


pA = &punktA;
```

- Wie greife ich auf Komponenten von `punktA` über den Zeiger `pA` zu?
- Komponenten-Selektion über Zeiger auf 2 Arten möglich:

  `(*pA).x` bzw. `pA->x`

- Beispiel:

  `(*pA).x = 3;` bzw. `pA->x = 3;`

Zeiger und geschachtelte Strukturen

- Strukturvariablen als Komponenten

```
struct Vektor
{
   struct Koordinaten *start;
   struct Koordinaten *ende;
};
```

- Zeiger vom Typ `struct Vektor`

```
struct Koordinaten punktA = {1.5, 3.0}, punktB = {2.5, 6.0};
struct Vektor vec;

vec.start = &punktA;
vec.ende = &punktB;

struct Vektor *pvec;
pvec = &vec;

vec.start->x = 9.0   bzw.
```

Strings in Strukturen

- String als Komponentenvariable

```
struct Name
{
   char nachname[20];
   char *vorname;
};

struct Name person1 = {"Maier", "Herbert"};
```

- `char nachname[20];` struktur**intern**, String ist Teil der Strukturvariable
- `char *vorname;` struktur**extern**, Zeiger auf einen String außerhalb (Hier auf eine Stringkonstante)

Typdefinition mit typedef

Was ist typedef?

- Vergabe eines neuen Namens (**Alias**) *neuername* für einen
  - schon bekannten oder
  - soeben definierten Datentyp

```
typedef bekanntertyp neuername;
```

- Beispiele:
  - Typ definieren

```
typedef unsigned long int UINT;
typedef float REAL;
```

  - Variable definieren

```
UINT a;
REAL b;
```

Anwendung

- spart Schreibarbeit
- Erlaubt Definition generischer Datentypen
- Portierbarkeit von Programmen mit maschinenabhängigen Datentypen (maschinenabhängige Datentypen treten dann NUR in der **typedef**-Zeile auf)
- Beispiel: Windows-API
  `https://learn.microsoft.com/en-us/windows/win32/winprog/windows-data-types`

Anwendungsbeispiel

64-bit data models

Data model	short (integer)	int	long (integer)	long long	pointers, size_t	Sample operating systems
LLP64, IL32P64	16	32	32	64	64	Microsoft Windows (x86-64 and IA-64) using Visual C++; and MinGW
LP64, I32LP64	16	32	64	64	64	Most Unix and Unix-like systems, e.g., Solaris, Linux, BSD, macOS. Windows when using Cygwin; z/OS
ILP64	16	64	64	64	64	HAL Computer Systems port of Solaris to the SPARC64
SILP64	64	64	64	64	64	*Classic* UNICOS[41] (versus UNICOS/mp, etc.)

- Beispiel: einheitlicher Long-Integer `INT64`

```
typedef long long int INT64;  // Anpassung auf LLP64-System
typedef long int INT64;       // Anpassung auf LP64-System
```

typedef und Strukturen

```
struct Datum
{
  short int tag;
  char monat[10];
  short int jahr;
};


struct Datum heute;
```

```
typedef struct
{
  short int tag;
  char monat[10];
  short int jahr;
} DATUM;


 DATUM heute;
```

Empfohlene Variante

Unionen

Union-Definition

- Varianten-Datentyp
- Benutzung ähnlich wie Struktur-Strukturtypen
- ABER: alle Komponenten (**Alternativen**) beginnen bei **der selben Adresse**
- speichert jeweils **nur eine einzige** Komponente (aus Reihe von Alternativen)
- Programmierer ggf. muss verfolgen, welcher Typ momentan in der Union gespeichert ist
- Beispiel:

```
union Variant
{
   int      intAlternative;
   float    floatAlternative;
};
```

Zugriff auf Alternativen

```
union Variant
{
   int      intAlternative;
   float    floatAlternative;
};

union Variant vario;
```

- Alternative schreiben / auslesen

```
vario.floatAlternative = 123.0;

int x;
x = vario.intAlternative;
```

Beispiel: Prozessorregister

```
struct WORDREGS
{
    unsigned int ax;
    unsigned int bx;
    unsigned int cx;
    unsigned int dx;
};

struct BYTEREGS
{
    unsigned char al, ah;
    unsigned char bl, bh;
    unsigned char cl, ch;
    unsigned char dl, dh;
};

union REGS
{
    struct WORDREGS w;
    struct BYTEREGS b;
};

union REGS myCPU;

myCPU.w.ax = 0x0815;  // AX Register auf Hex 0815 setzen
myCPU.b.al = 0x1A;    // AL Register auf Hex 1A setzen, AH bleibt auf Hex 08
```

Beispiel: Farbwerte mittels Union

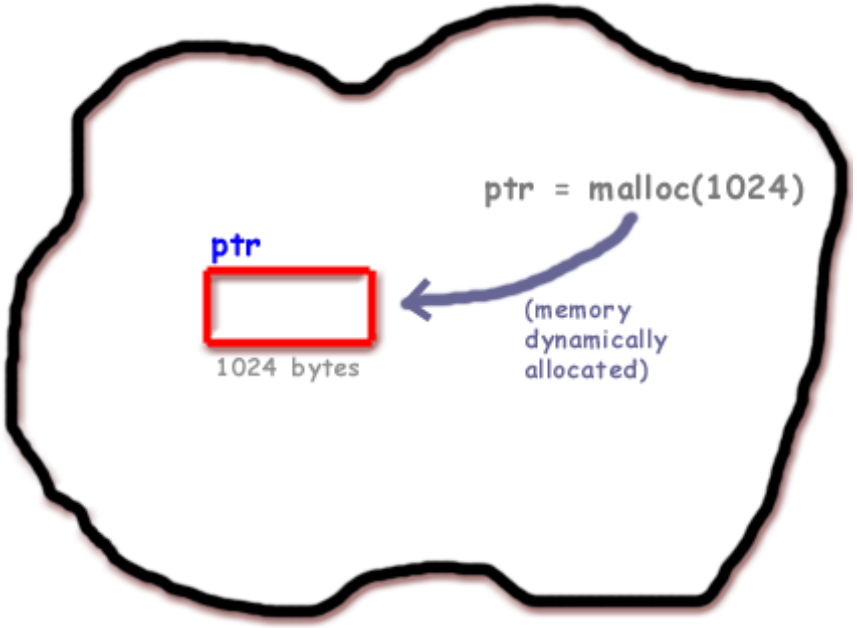
```
typedef union
{
  long int   farbwert;
  char       kanal[4];
} PIXEL;


PIXEL pix;
char einKanal;

pix.farbwert = 0xABCD1234;


einKanal = pix.kanal[3];    // liefert Farbkanal
                            // (höchstwertigste Byte)
pix.kanal[0] = 0xFF;        // setzt Farbkanal
                            // (niederwertigstes Byte)
```

Dynamische Speicherverwaltung



Motivation

Problem: Der Speicherbedarf des Programms ist während verschiedener Programmausführungen unterschiedlich hoch

Lösung: Variablen (insb. Arrays) maximal dimensionieren:

```
char eingabetext[1000];
```

→ Nachteile?

Dynamische Speicherverwaltung

Problem:	Speicherbedarf ist während verschiedener Programmausführungen unterschiedlich
Lösung 1:	Felder maximal dimensionieren → Nachteil: "verschenkter" Speicher
Lösung 2:	Felder als lokale Variablen anlegen (C99) → Nachteil: Lebensdauer der Feldvariable ist auf die Funktion beschränkt
Lösung **3:**	Speicherbedarf dynamisch zur Programmlaufzeit reservieren

stdlib.h bietet Funktionen zum Speicheranforderung und -freigabe:

```
void* malloc(int Groesse);
void* calloc(int AnzahlElemente, int ElementGroesse);
void* realloc(void *memBlock, int Groesse);

void free(void *memBlock);
```

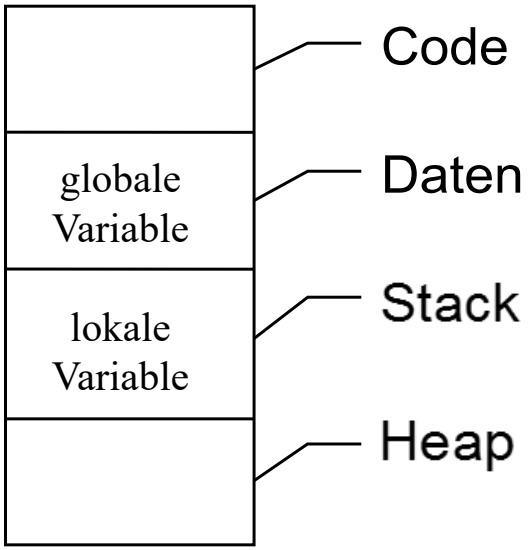
Statische vs. dynamische Variablen

- Lokale und globale Variable sind immer statisch:

```
int x;
```

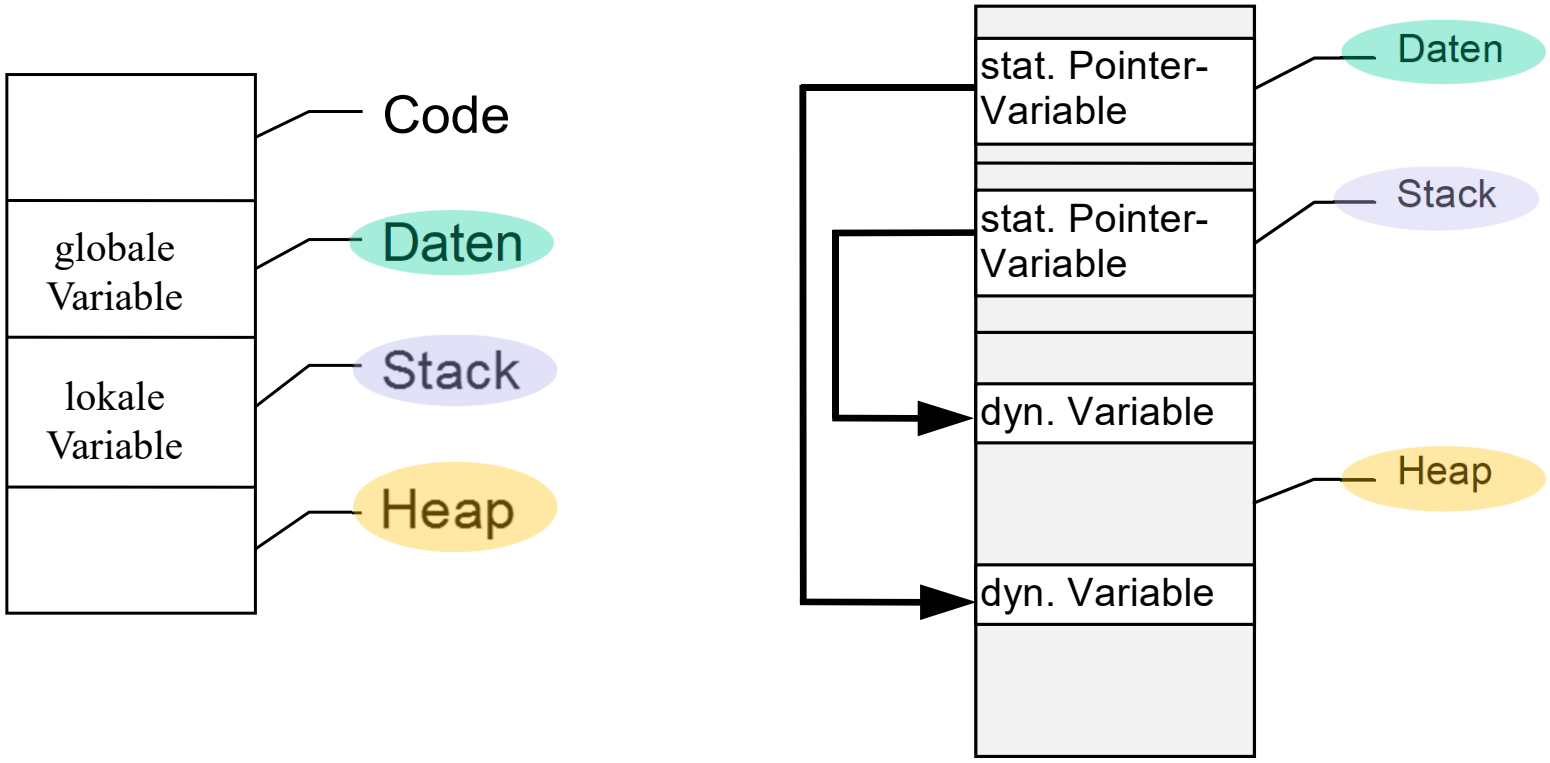
Ein dynamische Allokation lässt sich nur mittels Zeigern erreichen.

- Speichersegmente



Statische vs. dynamische Variablen

- eine dynamische Variable wird über einen Zeiger angelegt
- Anlegen erfolgt dann bei Bedarf auf dem **Heap**
- Zugriff und Freigabe erfolgen ebenfalls durch diesen Zeiger



Bibliotheksfunktionen zur dynamischen Speicherverwaltung

stdlib.h bietet folgende Funktionen zur Speicheranforderung und -freigabe:

```
void* malloc(int Groesse);

void* calloc(int AnzahlElemente, int ElementGroesse);

void* realloc(void *memBlock, int Groesse);

void free(void *memBlock);
```

Speicherreservierung mit malloc()

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int *pMem;

   pMem = (int *) malloc(sizeof(int));
   if (pMem == NULL)
   {
      printf("Speicheranforderung fehlgeschlagen.\n");
      return EXIT_FAILURE;
   }
   else
   {
      *pMem = 3;
      printf("pointer auf int-Zahl mit Wert: %d\n", *pMem);
      free(pMem);
   }
   return EXIT_SUCCESS;
}
```

Speicherreservierung mit realloc() I

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int *pMem;

   pMem = (int *) malloc(sizeof(int));
   if (pMem == NULL)
   {
       printf("Speicheranforderung fehlgeschlagen.\n");
       return EXIT_FAILURE;
   }

   *pMem = 3;
   printf("Integer Variable im dynamischen Speicher: %d\n", *pMem);

   ...
```

Speicherreservierung mit realloc() II

```
    ...

    pMem = (int *) realloc((void *)pMem, 2 * (sizeof(int)));
    if (pMem == NULL)
    {
       printf ("Speicheranforderung fehlgeschlagen.\n");
       return EXIT_FAILURE;
    }

    pMem[0] = 123;
    pMem[1] = 456;

    printf("1. Element von pMem hat den Wert: %d\n", pMem[0]);
    printf("2. Element von pMem hat den Wert: %d\n", pMem[1]);

    free(pMem);
    return EXIT_SUCCESS;
}
```

Beispiel: Feld variabler Größe

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int *pFeld = NULL;
   int i, anzahl = 0;

   printf "Wie viele Feldelemente sollen angelegt werden? ");
   scanf("%d", &anzahl);

   pFeld = (int *) malloc(anzahl * sizeof(int));
   if (pFeld == NULL)
   {
      printf("Programmabbruch: Speicheranforderung fehlgeschlagen.\n");
      return EXIT_FAILURE;
   }

   // Nutzung des Speicherbereichs, Belegung mit Quadratzahlen
   for (i = 0; i < anzahl; i++)
       pFeld[i] = i * i;

   free(pArray);
   return EXIT_SUCCESS;
}
```

malloc & free: 10 Typische Fehler

	Fehler	Folge
1)	kein Speicher mehr frei	keine (bei korrekter Prüfung)
2)	Freigabe einer falschen Adresse	u.U. Absturz
3)	Freigabe bereits freigegeben Speichers	u.U. Absturz
4)	Freigabe eines regulären Feldes / einer regulären Variablen mit free()	u.U. Absturz
5)	Freigabe eines nicht-initialisierten Speichers	u.U. Absturz
6)	Zugriff auf ungültigen Speicher vor der Speicheranforderung	u.U. Absturz
7)	Zugriff auf bereits freigegebenen Speicher	u.U. Absturz
8)	Zugriff auf Speicher mittels ungültiger Indizes	u.U. Absturz
9)	Verlust des Speichers durch Überschreiben des Zeigers	"memory leak"
10)	Verlust des Zeigers durch Rücksprung aus einer Funktion (z.B. return-value verworfen)	"memory leak"