

Speicherverwaltung in C

27.05.2004

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 1

Dynamische Speicherverwaltung in C

- In diesem Kapitel wollen wir zeigen:

- ➔ welche Speicherbereiche C verwendet
- ➔ welche Daten in welchen Bereichen gespeichert werden
- ➔ was dynamische Speicherverwaltung ist
- ➔ wie man dynamischen Speicher vereinbart
- ➔ wozu man dynamischen Speicher verwendet
- ➔ welche Funktionen zur Verwaltung von dynamischen Speicher zur Verfügung stehen.
- ➔ wie man Programme auf mehrere C-Dateien verteilt (erweitertes modulares Design).

27.05.2004

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 2

Speicheraufteilung in C

- C verwendet **getrennte, in sich zusammenhängende Speicherbereiche**, um
 - ➔ **den Programmcode**
 - ➔ **globale (externe) Variable**
 - ➔ **lokale Variable**
 - ➔ **dynamische Variable** zu speichern.
- Dies ist vorteilhaft, weil
 - ➔ sich der **Programmcode** während der Laufzeit des Programms normalerweise **nicht ändert**.
 - ➔ **externe Variable** während der **Laufzeit des gesamten Programms** gültig sind.
 - ➔ **dynamische und lokale Variable** nur in dem Programmabschnitt gültig sind, in dem sie vereinbart werden.

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 3

27.05.2004

Speichersegmente in C

- Die unterschiedlichen Speicherbereiche heißen auch
 - ➔ **Segmente**
- Man kann mit C-Anweisungen festlegen, in welchen Segmenten die Daten abgelegt werden.
 - **Ohne besondere Festlegungen** enthält

das Code segment	➔ den Programmcode
das Daten segment	➔ globale Variable
der stack (Stapel)	➔ lokale Variable
der heap (Haufen)	➔ dynamische Variable

- **Funktionsparameter und -rücksprungadressen** werden ebenfalls auf dem Stack abgelegt.

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 4

27.05.2004

Statische und dynamische Variable in C

- **Globale Variable** sind statisch und speicherineffizient.
 - Sie haben den Nachteil, dass sie im Datensegment
➡ auch dann **Speicherplatz belegen**, wenn sie
noch nicht oder **nicht mehr** gebraucht werden.
- **Lokale Variable** sind dynamisch und speichereffizient.
 - Sie werden von C **zu Beginn des Blocks**, in dem sie
vereinbart werden, **automatisch auf dem Stack angelegt**.
- **Am Ende des Blocks** werden die Speicher dieser lokalen
Variablen **wieder aufgegeben**.
- **Derselbe** Speicherbereich kann nun vorteilhaft von
einem folgenden Block **erneut verwendet** werden.

27.05.2004

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 5

Dynamische Speicher in C

- C stellt Bibliotheksfunktionen zur Verfügung, mit
denen der **Programmentwickler selbst**
dynamischen Speicherplatz auf dem Heap anlegen
und **wieder aufgeben kann**.
- Dies kann beispielsweise für Zwischenablagen
(vgl. *MS Windows*) benutzt werden:
 - Wenn man z.B. einen Textbereich oder ein Grafik
markiert und in die Zwischenablage kopiert, so wird der
Speicherplatz für die Zwischenablage genau so groß
angefordert, dass das gesamte Objekt hineinpasst.
- Ist der Kopiervorgang beendet, so kann der Speicher
wieder freigegeben und neu verwendet werden.

27.05.2004

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 6

Eigenschaften des Heap (1)

- **Dynamische Variable** auf dem *Heap* können in C **nicht mit Namen**, sondern **nur mit Zeigeroperatoren** angesprochen werden.
- **Die Lebensdauer** von dynamischen Variablen auf dem *Heap* ist **nicht an Blockgrenzen gebunden**:
Der Entwickler kann dynamischen Speicher zu jeder Zeit reservieren und wieder frei geben.
- Speicher auf dem *Heap* wird immer **zusammenhängend reserviert**.
 - Die **Größe des Heap** kann eingestellt werden.
Sie ist aber mindestens durch den Speicher des Rechners begrenzt.

27.05.2004

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 7

Eigenschaften des Heap (2)

- Ein **Heap-Überlauf** kann auftreten, wenn nur **Speicher angefordert, aber nicht wieder freigegeben** wird.
 - C besitzt im Unterschied z.B. zur *Internet-Programmiersprache JAVA* keinen
➡ **garbage collector** ('Müllsammler').
Ein **garbage collector** ordnet den **Speicher** von Zeit zu Zeit **neu**, so dass statt vieler kleiner wieder ein großer zusammenhängender Speicherbereich entsteht.
 - Bei **häufiger Reservierung** und **Freigabe** von dynamischem Speicher wird der Heap mehr und mehr **zerstückelt**:
Es kann zu einem **Heap-Überlauf** kommen, obwohl **in Summe** noch genügend Speicher vorhanden ist.

27.05.2004

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 8

Funktionen zur dynamischen Speicherverwaltung

- In C existieren u.a. 2 Bibliotheksfunktionen in den Bibliotheken `stdlib.h` und `alloc.h` zur
 - ➔ **Reservierung von dynamischem Speicher:**
`(void *) malloc (size_t Anzahl_Bytes)`
 - ➔ **Freigabe von dynamischem Speicher:**
`void free (void *pointer)`
- Zusätzlich gibt es in C ein Schlüsselwort zur
 - ➔ **Ermittlung des Speicherbedarfs:**
`size_t size = sizeof Variable`
oder
`size_t size = sizeof ( Typ )`

27.05.2004



Dynamische Speicherreservierung: `malloc (..)`

- Die Funktion `void* malloc (size_t size)` erwartet als **Eingangsparameter**
 - ➔ **size** ist die Anzahl der **zu reservierenden Bytes** im compilerspezifischen Format `size_t` (im Allgemeinen = `unsigned int`)
 - `malloc (..)` liefert als **Funktionsrückgabewert**
 - ➔ einen **Zeiger auf void**, der die **Anfangsadresse** des dynamisch vereinbarten Speicherbereichs enthält.
Kann kein Speicher alloziert (*allocated*) werden, so wird der Wert `NULL` (`'\0'`) zurückgegeben.
- Ein Zeiger auf void kann als einziger **ohne explizite Typkonversion** jedem anderen Zeigertyp zugewiesen werden.

27.05.2004



Dynamische Speicherreservierung:

free (..), sizeof (..)

- Die Funktion **void free (void *pointer)**

erwartet als **Eingangsparameter**

➔ **pointer** den Zeiger auf den zuvor reservierten Speicherbereich, den **malloc (..)** dem Zeiger **pointer** zugewiesen hat.

- Die Funktion **sizeof (Typ)** oder **sizeof Variable** erwartet als **Eingangsparameter** entweder

➔ **Typ** einen Datentyp in Klammern
oder

➔ **Variable** einen Variablennamen ohne Klammern

27.05.2004

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 11

Beispiel: dynamische Speicherverwaltung (1)

- Das folgende Programm **TEST24.C**

➔ stellt fest, wie viele Bytes die Struktur **DATUM** belegt,
➔ reserviert den benötigten Speicherplatz dynamisch,
➔ liest die Komponenten von **DATUM** ein
➔ druckt die Komponenten von **DATUM** aus.

```
/* Datei: TEST24.C */
#include <stdio.h>    /* Bibliothek stdio.h einbinden */
#include <stdlib.h>   /* Bibliothek stdlib.h einbinden */
int main (void)      /* Hauptprogramm main (void) */
{
    typedef struct    /* Strukturtyp DATUM vereinbaren*/
    {
        int tag, monat, jahr;
    } DATUM;

    ***
```

27.05.2004

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 12

Beispiel: dynamische Speicherverwaltung (2)

```

***
DATUM *p_DATUM ;    /* Zeiger auf Struktur DATUM vereinbaren */
unsigned int Anzahl_Bytes ;
Anzahl_Bytes = sizeof (DATUM) ;    /* Funktion sizeof (..) aufrufen */
printf ("\\n Die Struktur DATUM belegt %u Bytes.\\n", Anzahl_Bytes) ;
p_DATUM = (DATUM*) malloc (Anzahl_Bytes) ;    /* Speicher
                                              reservieren */

if ( p_DATUM == NULL)    /* Fehlerabfrage bei malloc (..) */
{
    printf ("\\n Kein Speicher alloziert: Programmabbruch! \\n") ;
    /* Fehlermeldung */

    return 1 ;    /* Fehlerrücksprung ins Betriebssystem */
}
***

```

27.05.2004

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 13

Beispiel: dynamische Speicherverwaltung (3)

```

***
printf (" Bitte im unsigned int-Format eingeben: Tag.Monat.Jahr: ") ;
fflush (stdin) ;    /* Eingabepuffer löschen */
scanf ( "%u.%u.%u", &(p_DATUM->tag), &(p_DATUM->monat),
        &(p_DATUM->jahr) ) ; /* Struktur des Typs DATUM einlesen */
printf ( "\\n Struktur des Typs DATUM im dynamisch belegten"
        "Speicher: %u.%u.%u\\n\\n Programmende!",
        p_DATUM->tag, p_DATUM->monat, p_DATUM->jahr) ;
    /* Struktur des Typs DATUM ausgeben */

free (p_DATUM);    /* allozierten Speicher wieder freigeben */
return 0 ;    /* Rücksprung ins Betriebssystem */
}

```

27.05.2004

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 14



Aufgabe: dynamische Speicherverwaltung (1)

1. Erläutern Sie die Zugriffe auf die Komponenten der Struktur des Typs **DATUM** bei der **scanf (..)**- und **printf (..)**-Anweisung des vorhergehenden Beispiel:

```
typedef struct          /* Strukturtyp DATUM vereinbaren */
{
    int tag, monat, jahr;
} DATUM;
DATUM *p_DATUM ; /* Zeiger auf Struktur DATUM vereinbaren */
...
scanf ( "%u.%u.%u", &(p_DATUM->tag), &(p_DATUM->monat),
        &(p_DATUM->jahr) ); /* Struktur des Typs DATUM einlesen */
printf ( "In Struktur des Typs DATUM im dynamisch belegten"
        "Speicher: %u.%u.%u\n\n Programmende!",
        p_DATUM->tag, p_DATUM->monat, p_DATUM->jahr) ;
        /* Struktur des Typs DATUM ausgeben */
```

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 15

27.05.2004



Aufgabe: dynamische Speicherverwaltung (1) (1)

1. Zugriffe auf die Komponenten der Struktur **DATUM** in **scanf (..)**:

- Auf den dynamischen Speicher im Heap kann nur mit Zeigerkonstrukten zugegriffen werden:
- Es gibt keinen Variablennamen, sondern nur den Zeiger **p_DATUM** auf die Struktur des Typs **DATUM**.

```
DATUM *p_DATUM ;      ↓ ↓ ↓          /* Zeiger auf DATUM */
...
scanf ( "%u.%u.%u", &(p_DATUM->tag), &(p_DATUM->monat),
        &(p_DATUM->jahr) ); /* Struktur des Typs DATUM einlesen */
```

➡ **scanf (..)** verlangt auf dieser Position einen Zeiger auf die Komponente **tag** von **DATUM** (die Adresse von **tag** auf dem Heap)

➡ Auf **DATUM** kann man nur mit dem Zeiger **p_DATUM** zugreifen. **p_DATUM->tag** ist die Abkürzung für **(*p_DATUM).tag**: Es wird auf die **Komponente tag** des **Objekts** von **p_DATUM** zugegriffen.

➡ Der Adressoperator **&** erzeugt die **Adresse der Komponente tag** des Objekts von **p_DATUM**.

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 16

27.05.2004

✓ Aufgabe: dynamische Speicherverwaltung (1) (2)

1. Zugriffe auf die Komponenten der Struktur *DATUM* in *printf* (..):

- Auf den dynamischen Speicher im Heap kann nur mit Zeigerkonstrukten zugegriffen werden:
- Es gibt keinen Variablennamen, sondern nur den Zeiger **p_DATUM** auf die Struktur des Typs **DATUM**.

```
DATUM *p_DATUM ;
...
printf ( ...
           p_DATUM->tag, p_DATUM->monat, p_DATUM->jahr) ;
```

/* Zeiger auf DATUM */

➡ **printf** (..) verlangt auf diesen Positionen die Komponenten von **DATUM**.

➡ Auf **DATUM** kann man nur mit dem Zeiger **p_DATUM** zugreifen.
p_DATUM->tag ist die Abkürzung für **(*p_DATUM).tag**:
 Es wird auf die **Komponente tag** des **Objekts von p_DATUM** zugegriffen.



✎ Aufgabe: dynamische Speicherverwaltung (2)

2. Schreiben Sie ein Programm, das mit Hilfe von **malloc** (..) feststellt, wie viele KByte an Speicher auf dem *Heap* noch zur Verfügung steht.

Das Programm soll den Speicherplatz mit einem Fehler von maximal 1 KByte ermitteln!



✓ Aufgabe: dynamische Speicherverwaltung (2)

2. C-Quelltext zur Ermittlung des freien Heap-Speichers:

```

/* Datei: test25.c */
#include <stdio.h>      /* Bibliothek stdio.h einbinden */
#include <stdlib.h>     /* Bibliothek stdlib.h einbinden */
int main (void)        /* Hauptprogramm main (void) */
{
    int *p_mem;
    unsigned long Kbytes = 1024; /* 1 Kbyte = 1024 Byte ! */
    while (1)
    {
        p_mem = malloc(Kbytes); /* Kbytes Speicher allozieren */
        if (p_mem) Kbytes += 1024; /* Fehler?; nein: 1 Kbyte mehr */
        else break; /* Fehler: Schleifenende */
        free (p_mem); /* Kbytes Speicher wieder freigeben */
    }
    printf ( "\n Auf dem Heap sind ca. %ld Kbyte
zusammenhaengender Speicher frei.\n\n "
" Programmende!", Kbytes / 1024 ); /* Terminalausgabe */
    free (p_mem); /* Speicher freigeben */
    return 0; /* Rücksprung ins Betriebssystem */
}

```

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 19

27.05.2004

Funktion des Stack

- **Stacks** sind **Stapel zur Zwischenspeicherung von Daten**
- **Variable** können dort **zwischengespeichert** werden.
 - In Stacks wird - ebenso wie im *Heap* - der **Speicher dynamisch verwaltet**.

Dabei wird nicht mehr benötigter Speicher wieder
freigegeben. Dieser kann dann von anderen
Programmteilen erneut verwendet werden.

➡ Vorteil:

* **effizientere Speichernutzung**

➡ Nachteil:

* **Fehler sind fatal**
* **Variable können nur über
Zeigeroperationen
angesprochen werden**

27.05.2004

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 20

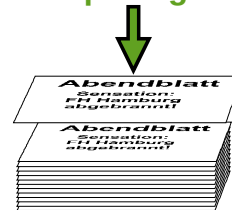
Stack: Datenstapel füllen

- Stacks sind LIFO-Speicher (*last in first out*).
- Stacks funktionieren nach dem Prinzip des **Zeitungsstapels**.

- **Stack (LIFO) füllen**
(*stacking*):

letzte (neueste) Zeitung auf
den Stapel legen

letzte Zeitung
auf den
Stapel legen



27.05.2004

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 21

Stack: Datenstapel leeren

- **Stack leeren**

letzte Zeitung als erste
vom Stapel abholen



27.05.2004

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 22

Stack und Funktionsaufrufe

- **Normalerweise** wird der **Stack** bei **Funktionsaufrufen** benötigt.
 - Bei jedem Funktionsaufruf werden
 - ➔ die **Rücksprungsadresse**
 - ➔ **alle Eingangsparameter** der Funktion
 auf den Stack kopiert.
- Ferner werden **alle lokalen Variablen der Funktion** auf dem Stack abgelegt.
- Deshalb können Funktionen **geschachtelt** und auch **rekursiv** aufgerufen werden.

27.05.2004

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 23

Stack bei geschachtelten Funktionsaufrufen (1)

- Das Programm zeigt ➔ **mehrere Funktionsaufrufe**

```
int main (void)          /* Hauptprogramm main (void) */
{ char x1, x2, x3 ;      /* Variable von main vereinbaren */
  int y0, y1 ;

  ...

  func1 (x1,x2) ;        /* Aufruf der Funktion func1 (..) */
  ...                    /* kein weiterer Funktionsaufruf! */
  func2 (x3) ;           /* Aufruf der Funktion func2 (..) */
}

...

void func1 (char a1,char a2) /* Funktion func1 (a1,a2) */
{ char f1 ;              /* lokale Variable von func1 vereinbaren */
  ...                    /* keine weiteren Variablen in func1! */
  return;                /* Rücksprung nach main */
}

...

void func2 (char b3)       /* Funktion func2 (b3) */
{ ...                     /* Rücksprung nach main */
  return ;
}
```

27.05.2004

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 24

Stack bei Funktionsaufrufen (2)

- Das Beispiel zeigt **die Anordnung des Stacks** und **den Stackpointer (SP)**

```

int main (void)
{ char x1, x2, x3 ;
  ...
  func1 (x1, x2) ;
  ...
  ...
  func2 (x3) ;
  ...
}

void func1 (char a1, char a2)
{ char f1 ;
  ...
  return ;
}

void func2 (char b3)
{ ..
  ...

```

STACK	
SP →	f1
SP →	Rücksprungadresse von func1
	Kopie von x2
	Kopie von x1
	XXX

27/05/2004

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd

Programmieren in ANSI C:
 K13_ Speicherverwaltung.ppt, Seite 25

Stack bei Funktionsaufrufen (3)

- Das Beispiel zeigt **die Anordnung des Stacks** und **den Stackpointer (SP)**

```

int main (void)
{ char x1, x2, x3 ;
  ...
  func1 (x1, x2) ;
  ...
  ...
  func2 (x3) ;
  ...
}

void func1 (char a1, char a2)
{ char f1 ;
  ...
  return ;
}

void func2 (char b3)
{ ..
  ...

```

STACK	
SP →	f1
SP →	Rücksprungadresse von func1
	Kopie von x2
	Kopie von x1
	XXX

27/05/2004

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd

Programmieren in ANSI C:
 K13_ Speicherverwaltung.ppt, Seite 26

Stack bei Funktionsaufrufen (4)

- Das Beispiel zeigt ➡ die **Anordnung des Stacks** und ➡ den **Stackpointer (SP)**

```
int main (void)
{ char x1, x2, x3 ;
  ...
  func1 (x1, x2) ;
  ...
  ...
  func2 (x3) ;
  ...
}
...
void func1 (char a1, char a2)
{ char f1 ;
  ...
  return ;
}
...
void func2 (char b3)
{ ..
  ...
}
```



SP ➡

SP ➡

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 27

27.05.2004

Speicherklassen in C

- C verwendet ➡ **Speicherklassen** zur
 - ➡ Steuerung des **Speicherorts**
 - ➡ Steuerung des **Gültigkeitsbereichs** von Objekten.
- Grundsätzlich unterscheidet man die 2 Speicherklassen:
 - ➡ **automatisch**
 - ➡ **statisch**
- Mit den Schlüsselwörtern:

auto register static extern typedef

kann man festlegen, in **welchem Speicherbereich** eine Variable abgelegt wird.

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 28

27.05.2004

Automatische Speicherklasse: **auto** Variable

- Die Schlüsselwörter:
auto und **register**

definieren Objekte der **automatischen Speicherklasse**.

- Automatische Objekte existieren **lokal in einem Block**. Sie sind **nach Verlassen des Blocks nicht mehr gültig**.
- Definitionen in einem Block erzeugen **automatische Objekte**, wenn **keine Speicherklasse angegeben ist**.
 - auto-Objekte** werden **auf dem Stack** abgelegt.

Beispiel:

```
int main (void)
{
    auto char c ; ...    /* char-Variablen c definieren ≡ char c */
    register int *p_int; /* Zeiger p_int auf int soll im Register liegen */
}
```

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 29

27/05/2004

Automatische Speicherklasse: **register** Variable

- Der Compiler versucht, eine **register**-Variable in einem schnellen CPU-Register zu halten. Dadurch kann die **Programmablaufzeit** gegebenenfalls erheblich **verkürzt werden**.
- Es gibt aber **keine Garantie** dafür, dass eine **register**-Variable tatsächlich in einem Register abgelegt ist.
- Auf **register**-Variable darf der Adressoperator **&** **weder implizit noch explizit angewendet** werden.
- Falls gewünscht darf für formale Funktionsparameter nur die Speicherklasse **register** angegeben werden.

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 30

27/05/2004

Statische Speicherklasse: **extern**

- **extern**

- wird benötigt, wenn man aus **verschiedenen Programmen** auf **dieselben Variablen** zugreifen möchte.
- wird **vor dem 1. Aufruf** einer Variablen benötigt, die erst **später im gleichen Programm** definiert wird.

Eine Variable darf **nur in einem Programm definiert** und **initialisiert** werden.

Vor der Verwendung in einem anderen Programm muss sie dem Compiler mit **extern** bekannt gemacht werden.

Beim Linken der Programme setzt der Linker die Adresse der extern-Variablen ein.

27.05.2004



Beispiel für die Verwendung von **extern** (1)

```
/* Datei c1.c */
#include <stdio.h>          /* Bibliothek stdio.h einbinden */

void func1 (void) ;         /* Prototyp: func1 (..) ist in c1.c definiert */
extern void func2 (void) ;  /* func2 (..) ist in einer anderen Datei */
extern int i1 ;             /*- hier in c2.c - und i1 ist erst später definiert */

int main (void)             /* Hauptprogramm main (..) */
{ printf (" \n Programm zur Erläuterung von 'extern':\n"
  " \n In 'main': i1 = %d\n", i1); /* Ausgabe von i1 in main */
  func1 ();                 /* func1 ( ) aufrufen */
  func2 ();                 /* func2 ( ) aufrufen */
  printf (" \n beliebige Taste = Programmende !");
  getch (stdin);            /* Benutzerbildschirm offen halten */
  return 0; }               /* Rücksprung ins Betriebssystem */

int i1 = 10;                /* Definition und Initialisierung von i1 */

void func1 (void)           /* Definition von func1 */
{ printf (" \n In 'func1' : i1 = %d\n", i1); } /* Ausgabe von i1 in func1 */
```

27.05.2004



Beispiel für die Verwendung von **extern** (2)

```
/* Datei c2.c */
#include <stdio.h>          /* Bibliothek stdio.h einbinden */

extern int i1;              /* i1 ist in c1.c definiert */

void func2 (void)          /* Definition von func2 (..) */
{ printf ("\n In 'func2' : i1 = %d\n", i1); } /* Ausgabe von i1 in func2 */
```

- Bei der Prototypenvereinbarung
extern void func2 (void) ;
darf **extern entfallen**, weil der C-Compiler **bei Funktionen** die **externe Speicherklasse** als **Voreinstellung** benutzt.
- Bei **externen Funktionen** macht die explizite Angabe von **extern** die Programme leichter lesbar.
- Mehrere Programme werden in **MSVC++** am einfachsten im Menü ➡ **Projekt** zusammengebunden.

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd

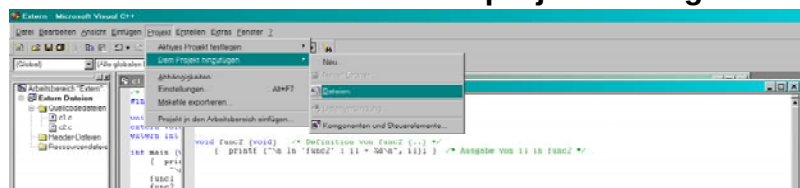


Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 33

27.05.2004

Beispiel für die Verwendung von **extern** (3)

- Die Programme werden in **MSVC++** im Menü
➡ **Projekt** zusammengebunden,
nachdem man ein neues Konsolprojekt erzeugt hat



- In diesem Fall heißt das Projekt **Extern** und besteht aus den beiden C-Quelldateien **c1.c** und **c2.c**.
- Beim Kompilieren fragt MSVC, ob es die Datei **Extern.exe** erstellen soll. Die Antwort lautet **ja**.
- Die **gesamte exe-Datei** erhält also den **Namen des Projekts**.

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 34

27.05.2004

Beispiel für die Verwendung von **extern** (4)

- BildschirmAusgabe von **Extern.exe**

```
"I:\MS Visual C++\PR2\Extern\Debug\Extern.exe"

Programm zur Erläuterung von 'extern':

In 'main': i1 = 10

In 'func1' : i1 = 10

In 'func2' : i1 = 10

beliebige Taste = Programmende !_

```

27.05.2004



Statische Speicherklasse: **extern, static, typedef**

- Die Schlüsselwörter:

static extern und **typedef**

definieren Objekte der **statischen Speicherklasse**.

- typedef**

definiert einen neuen Variablentyp und **reserviert keinen Speicherplatz**.

- static**

- **lokale** Variable werden nicht auf dem Stack, sondern im Speicherbereich der globalen Variablen angelegt.
- kann auch für **externe Variable** und **Funktionen** verwendet werden.

Wenn ein **externes** Objekt als **static** definiert wird, so ist es **außerhalb der Datei, in der es definiert wird, unsichtbar**.

27.05.2004



Schlüsselwörter: **const**, **volatile**

- **const**

definiert Variable als konstant.

const Variable können nur **einmal initialisiert** und danach nicht mehr überschrieben werden.

Ein C-Compiler **muss** diese Bedingung **nicht überprüfen**!

Beispiel:

```
const long double PI = 3.1415926535897932384626433832795
```

```
/*  $\pi$  als long double Konstante definieren */
```

- **volatile**

definiert Variable als veränderlich.

- Sie werden z.B. keiner **Schleifenoptimierung** unterzogen.

27.05.2004

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 37

Schlüsselwort: **volatile**

Beispiel:

```
...
volatile unsigned char flag; /* globale Variable in externem Register */
flag = 0;
while (flag == 0) /* flag in externem Register abfragen */
{
    ... /* In dieser Schleife wird flag nicht verändert !!! */
}
...
```

- Ein optimierender C-Compiler kann nicht **automatisch** erkennen, dass **flag** in der Schleife von einer **externen Hardware** verändert wird.
- Er erkennt normalerweise, dass **flag** in der **while-Schleife** **nicht verändert** wird und **lässt die Schleife** eventuell mit einer Warnung **weg**.
- Die Angabe **volatile** verhindert diese Optimierung.

27.05.2004

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 38

Speicherklassen: tabellarische Übersicht

- Die Tabelle zeigt die **Eigenschaften** der **unterschiedlichen Speicherklassen**.

Speicher- klasse	Variable	Gültig- keit	Lebens- dauer	Automat. initialisiert	Ort: Segment
auto	lokal	Block	Block	nein	Stack
register	lokal	Block	Block	nein	Register / Stack
static	lokal	Block	Programm	mit 0	Daten
static	global	In Datei ab De- finition	Programm	mit 0	Daten
extern	global	In Pro- gramm ab De- finition	Programm	mit 0	Daten

27.05.2004

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 39



Zeiger als Rückgabewert von Funktionen (1)

Ein Programmentwickler hat das folgende Programm geschrieben:

- Hauptprogramm **main (...)**

```
#include <stdio.h>
#define max 6

char *semester6 ( short );

int main (void)
{
    short idx ;
    for (idx = 0; idx < max +1 ; idx++)
        printf ("%s\n", semester6 ( idx ) ) ;
    return 0;
}
```

27.05.2004

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 40



Zeiger als Rückgabewert von Funktionen (2)

- Funktion `semester6 (...)`

/* Achtung: Die folgende Funktion enthaelt Fehler !!! */

```
char *semester6 ( short index )
{
    char semester [ ] [13] =
        { "A6", "I6", "IE6", "N6", "TI6", "ST6", "SB6", "Indexfehler!" };

    if ( index < 0 || index > max-1 )
        return &semester [7] [0] ;
    else
        return &semester [index] [0] ;
}
```

1. Kommentieren Sie jede Zeile des Programms!
2. Welchen Fehler hat der Programmierer gemacht?
3. Korrigieren Sie das Programm!

27.05.2004

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 41



✓ Zeiger als Rückgabewert von Funktionen (1)

1. Programm kommentieren

- Hauptprogramm `main (...)`

```
#include <stdio.h>      /* Bibliotheksfunktion für printf einbinden */
#define max 6           /* Konstante max statisch definieren */

char *semester6 ( short ) ;    /* Prototyp von semester6 definieren */

int main (void)          /* Beginn des Hauptprogramms main */
{
    short idx ;           /* Variable idx des Typs short definieren */
    for (idx = 0; idx < max +1 ; idx++) /* Schleife von idx=0 ... idx<7 */
        printf ("%s\n", semester6 ( idx ) ) ; /* semester (idx) ausgeben */
    return 0 ;            /* Ruecksprung ins Betriebssystem */
}                          /* Ende des Hauptprogramms main */
```

27.05.2004

Hochschule für angewandte Wissenschaften Hamburg
Prof. Dr.-Ing. U. Vogelsang, Prof. Dr.-U. Sauvagerd



Programmieren in ANSI C:
K13_ Speicherverwaltung.ppt, Seite 42

✓ Zeiger als Rückgabewert von Funktionen (2)

1. Programm kommentieren

- Funktion `semester6 (...)`

/ Achtung: Die folgende Funktion enthaelt Fehler !!! */*

```
char *semester6 ( short index )      /* Beginn der Funktion
semester6 */
{ /* semester6 erwartet die short-Eingangsvariable index und
   gibt einen Zeiger auf char zurueck */
char semester [ ] [13] =
    { "A6", "I6", "IE6", "N6", "TI6", "ST6", "SB6", "Indexfehler!" };
/* Definition und Initialisierung des 2-D char-Feldes semester */
if ( index < 0 || index > max-1 )
    return &semester [7] [0] ; /* zulaessiger Indexbereich verlassen:
                                Rückgabe: Zeiger auf "Indexfehler!" */
else
    return &semester [index] [0] ; /* zulaessiger Indexbereich:
                                Rückgabe: Zeiger auf das dem Index zugehörige Semester */
}
```



✓ Zeiger als Rückgabewert von Funktionen (3)

2. Programmfehler

Der Programmierer hat nicht beachtet, dass lokale Variable einer Funktion auf dem Stack gespeichert werden.

Am Ende der Funktion wird der Speicherbereich der lokalen Variablen auf dem Stack freigegeben und u.U. neu belegt!

Der zurückgegebene Zeiger des Typs *char* zeigt auf nichts!

☛ **Es treten unvorhersehbare Fehler auf!**



✓ Zeiger als Rückgabewert von Funktionen (4)

3. Programm korrigieren (1)

1. Möglichkeit: Definition des Textfeldes als **static**.

Dies hat zur Folge, dass das Feld **nicht dynamisch** auf dem Stack, sondern **statisch im Datensegment** gespeichert wird.

```
char *semester6 ( short index )
static char semester [ ] [13] =
    { "A6", "I6", "IE6", "N6", "TI6", "ST6", "SB6", "Indexfehler!" } ;
/* Definition und Initialisierung des
   2-D char-Feldes semester als static!*/
if ( index < 0 || index > max-1 )
    return &semester [7] [0] ;
else
    return &semester [index] [0] ;
}
```

Es muss beachtet werden, dass eine Funktion mit **static** definierten lokalen Variablen nicht mehr rekursiv aufgerufen werden darf!



✓ Zeiger als Rückgabewert von Funktionen (5)

3. Programm korrigieren (2)

2. Möglichkeit: externe Definition des Textfeldes.

Dieses gilt - falls man keinen besonderen Grund hat - als unprofessionell. Das Programm wird unübersichtlich. Fehlerhafte Zugriffe aus anderen Programmteilen kann nicht mehr verhindert werden!

```
#include <stdio.h>
#include <conio.h>
#define max 6
char *semester6 ( short );
char semester [ ] [13] =
    { "A6", "I6", "IE6", "N6", "TI6", "ST6", "SB6", "Indexfehler!" } ;
/* semester extern ( als globale Variable ) definieren */
int main (void)
{
    ...
}
```



✓ Zeiger als Rückgabewert von Funktionen (6)

3. Programm korrigieren (3)

2. Möglichkeit: externe Definition des Textfeldes. Dieses gilt - falls man keinen besonderen Grund hat - als unprofessionell. Das Programm wird unübersichtlich. Fehlerhafte Zugriffe aus anderen Programmteilen kann nicht mehr verhindert werden!

/* Achtung: Hier ist das Feld *semester* extern definiert! */

```
char *semester6 ( short index )
{
    if ( index < 0 || index > max-1 )
        return &semester [7] [0] ;
    else
        return &semester [index] [0] ;
}
```

27.05.2004

