Timer

Zeit-Abfragen in C:

- System-Makro in `time.h` :

  `#define CLOCKS_PER_SEC  1000`

  Frühere Makro-Bezeichnung (z.T. noch gebräuchlich):

  `#define CLK_TCK CLOCKS_PER_SEC``//aus:time.h`

- Zwei C-Anweisungen:

```
/*Rechenzeit seit Start (long int: 32 Bit)
                  [sec*CLOCKS_PER_SEC ]:*/
clock_t start; start=clock();

//Wartezeit seit 01.01.1970 00:00:00[sec]:
time_t start; time(&start); //64-bit value
```

Beispiel: Rechenzeit-Timer


```
/* Beispiel fuer Timer */

#include <conio.h> /*kbhit,getch*/
#include <stdio.h> /*printf     */
#include <stdlib.h>/*exit       */
#include <time.h>  /*clock_t    */
#define CR4SEC  /*<CR> statt sec*/
#define CR   13
#define ESC  27

int main (void)
{ int     ch=0, tRest=30, dt=1;
  clock_t tj=0, tTick=0, tBell=0,
          cps=CLOCKS_PER_SEC;

  tj = clock();
  tBell = tj + tRest*cps;
  tTick = tj ;
```

Timer.exe

```
 do
 { if (_kbhit())
   { ch=_getch();printf("%c\r",ch);
   } if (ch == ESC) exit(1);

#ifdef  CR4SEC
   if (ch==CR) {tj+=dt*cps; ch=0;}
#else //CR4SEC
   tj=clock();
#endif//CR4SEC


   while (tj >= tTick)
   { printf("%70s%5d\r"," ",tRest);
     tRest -= dt;  tTick += dt*cps;
   }
 } while (tj < tBell);
 printf("\a"); /*beep*/
 return 0 ;
}
```

TimePerCR.exe

Beispiel: Uhrzeit-Timer


```c
/* Beispiel fuer Timer */

#include <conio.h> /*kbhit,getch*/
#include <stdio.h> /*printf     */
#include <stdlib.h>/*exit       */
#include <time.h>  /*clock_t    */
#define CR4SEC  /*<CR> statt sec*/
#define CR   13
#define ESC  27

int main (void)
{ int     ch=0, tRest=30, dt=1;
  time_t  tj=0, tTick=0, tBell=0,
          cps=1;

  time(&tj);
  tBell = tj + tRest*cps;
  tTick = tj ;
```

Timer.exe

```c
 do
 { if (_kbhit())
   { ch=_getch();printf("%c\r",ch);
   } if (ch == ESC) exit(1);

#ifdef  CR4SEC
   if (ch==CR) {tj+=dt*cps; ch=0;}
#else //CR4SEC
   time(&tj);
#endif//CR4SEC


   while (tj >= tTick)
   { printf("%70s%5d\r"," ",tRest);
     tRest -= dt;  tTick += dt*cps;
   }
 } while (tj < tBell);
 printf("\a"); /*beep*/
 return 0 ;
}
```

TimePerCR.exe

Globale Variablen

Typische Speicherübersicht eines C-Programms:

Globale Variablen ermöglichen

- einfachere Programmierung
- schnellen Zugriff zur Laufzeit (keine Stack-Kopie nötig)

ABER:

- verleiten zu „spezifischen“ Modulen – z.B.:

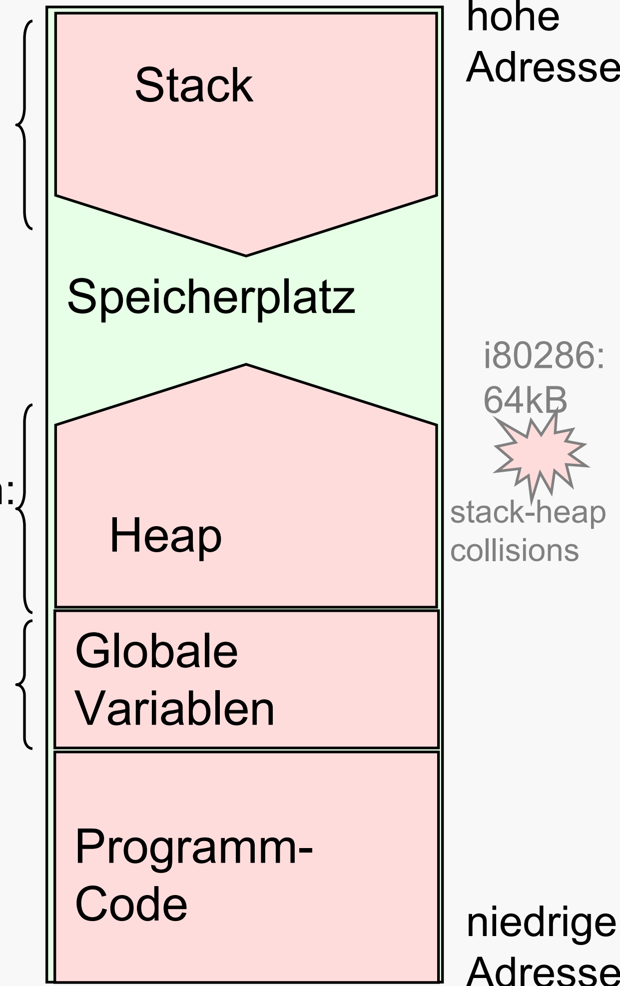
```
int x;
int squareL(int x)
{ return (x*x);
}

int squareG(void)
{ return (x*x);
}
```

Parameterliste, lokale Variablen, Rückgabewerte, CPU-Status

dynamische Daten: `malloc(),new()`

statische Daten (vgl.: `static`)



Globale Variablen

Weitere Nachteile des Einsatzes globaler Variablen:

- Keine Interrupt-Sperrung während d.Parameter-Übergabe
  - ↪ Gefahr der Daten-Inkonsistenz (durch andere Threads o. unübersichtliche Folge von Aufrufen)
- Kein Überblick über benötigte Variablen für eine Funktion
  - ↪ Erschwerung der Wiederverwendung von Code
- Bezeichner-Reservierung über das ganze Programm
  - ↪ Fehlerquelle: lokale Variablen setzen globale außer Kraft
- Verfügbarkeit von Daten über die gesamte Laufzeit
  - ↪ Erhöhung des Speicherbedarfs
- Ansprechbarkeit von Variablen über das ganze Programm
  - ↪ Aufhebung aller Kapselung, Gefahr der Datenmanipulation

➡ Eliminierung globaler Variablen (fast) immer zu empfehlen, manchmal aber mit Arbeit verbunden.

z.B.: Libraries!

Globale Variablen

Beispiel:

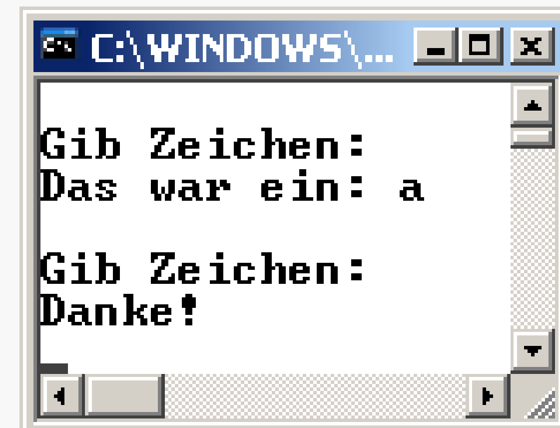
Zeichen eingeben und ausgeben:

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

#define ENDE   27
#define EWIG    1
```

```
int main(void)
{ char z=' ';
  while (EWIG)
  { printf ("\nGib Zeichen: ");
    z = _getch();
    if (z!=ENDE) printf("\nDas war ein: %c\n",z);
    else {printf("\nDanke!\n");_getch();exit(0);}
  }
  return 0;
}
```

(Globals\main\~\Test.exe)



Globale Variablen

Forderung: Modularisierung / Trennung von Ein- u. Ausgabe

```
char z=' ';
int Get(void)
{ printf ("\nGib Zeichen: ");  z = _getch();
  return 0;
}
int Put(void)
{ if (z!=ENDE) printf("\nDas war ein: %c\n", z);
  else { printf ("\nDanke!\n"); _getch(); exit (0); }
  return 0;
}

int main(void)
{ while (EWIG)
  { Get(); /* Zeichen eingeben */
    Put(); /* Zeichen ausgeben */
  }
  return 0;
}
```

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

#define ENDE   27
#define EWIG    1
```

(Globals\Glob\~\Test.*)

Globale Variablen

Forderung: Modularisierung / Trennung von Ein- u. Ausgabe
... ohne globale Variablen:

```
int common (char rw, char *zeichen)
{ static char _zeichen=' ';/*ohne static: wirkungslos!*/
  if (rw == READ) *zeichen = _zeichen;
  else             _zeichen = *zeichen;
  return 0;
}

int main(void)
{ while (EWIG)
  { Get(); /* Zeichen eingeben */
    Put(); /* Zeichen ausgeben */
  }
  return 0;
}
```

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

#define ENDE   27
#define EWIG    1
enum {READ, WRITE};
```

Globale Variablen

TECHNISCHE HOCHSCHULE MITTELHESSEN


```
int common (char rw, char *zeichen)
{ static char _zeichen=' '; /*ohne static: wirkungslos!*/
  if (rw == READ) *zeichen = _zeichen;
  else            _zeichen = *zeichen;  return 0;
}
int Get(void)
{ char zeichen=' '; /* static moeglich, nicht noetig */
  printf ("\nGib Zeichen: ");  zeichen = _getch();
  common (WRITE, &zeichen);    return 0;
}
int Put(void)
{ char zeichen=' ';
  common (READ, &zeichen);
  if (zeichen!=ENDE) printf ("\nDas war ein: %c \n", zeichen);
  else { printf ("\nDanke, das war's!\n");_getch(); exit (0);}
  return 0;
}
int main(void)
{ while (EWIG)  { Get(); /* Zeichen eingeben */
                  Put(); /* Zeichen ausgeben */
                } return 0;
}
```

(Globals\noGlob\~\Test.*)

Globale Variablen - und Callbacks


```
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#define ENDE   27
#define EWIG    1

int common (int (**pp2f)(void), char *zeichen);
int Get(void);
int Put(void);

int main(void)
{ while (EWIG)
  { Get(); /*Zeichen eingeben u. weiterschauen*/
  }
  return (0);
}
```

Globale Variablen - und Callbacks


```
int Get(void)
{ char zeichen=' ';  static int (*p2f)(void);  // Zeiger-Var.
  printf ("\nGib Zeichen: ");     zeichen = _getch();
  common (&p2f, &zeichen);        (*p2f)();         // Aufruf
  return (0);                    // p2f ();       // auch zulaessig
}
int common (int (**pp2f)(void), char *zeichen)
{ static char _erst=1, _zeichen=' ', _write=1;
  if (_erst) { *pp2f=Put;  _erst=0; }    /*Nur fuer Get()!*/
  if (_write) _zeichen=*zeichen; else *zeichen=_zeichen;
  _write = 1 - _write;
  if(*zeichen==ENDE){printf("\nDanke!\n");_getch();exit(0);}
  return (0);
}
int Put(void)
{ char  zeichen=' ';  int (*dummyP2f)();
  common (&dummyP2f, &zeichen);
  printf ("\nDas war ein: %c \n", zeichen);
  return (0);
}
```

(Globals\noGlobCall\~\Test.*)

für Hedonist/inn/en

Globale Variablen - und Callbacks

Einführung von `int(*common_(char **zeichen))(void);`

Die Funktion: `common_(char **zeichen)`

habe als Rückgabewert `R`:

… die Adresse einer anderen Funktion... `(*R)()`

…mit eigener leerer Parameter-Liste... `(*R)(void)`

… und eigenem Rückgabewert `int`: `int(*R)(void)`

- dann heißt der Prototyp von `common_()`:

```
int (*common_(char **zeichen))(void)
```

Globale Variablen - und Callbacks

für Hedonist/inn/en


```
int (*common_(char **zeichen))(void);         //Neuer Prototyp

int Get(void)
{ static char *zeichen=NULL;                  //NULL statt _erst
  if(!zeichen) common_(&zeichen);    //Speicherg in common_()
  printf("\nGib Zeichen: "); *zeichen = _getch();
 (*common_(&zeichen))();           //Aufruf ueber Rueckgabe-Pnt.
  return (0);
}
int (*common_ (char **zeichen))(void)
{ static char _zeichen=' ';            //Auch als Feld ausbaubar
  if(!*zeichen) { *zeichen=&_zeichen; }        //Initialisierg
  if(**zeichen==ENDE){printf("\nDanke!\n");_getch();exit(0);}
  return(Put);
}


int Put(void)
{ char  *zeichen=NULL;
  if (!zeichen) common_ (&zeichen);
  printf ("\nDas war ein: %c \n", *zeichen);
  return (0);
}
```

(Globals\05noGlobCallStore)