Grafische Benutzungsschnittstellen

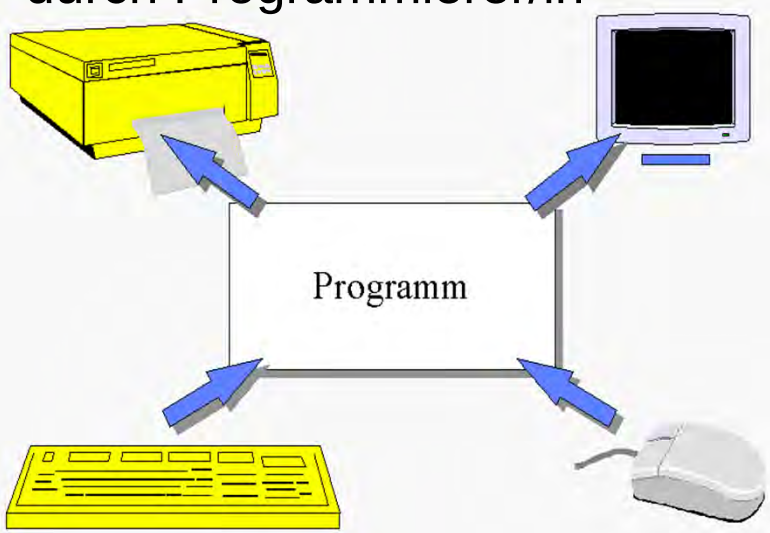
- Ziel der GUI: Verlagerung des Schreibtisches in den PC
  „Desktop“: Rechner auf dem Tisch – eher: Tisch-Oberfläche!
  Vorstellung: Papier-Stapel („Batch“) vs. Lose Blätter (Fenster)
- Einige Forderungen an GUI und ihre Auswirkungen
  - Intuitive Handhabung ⇨ Minimierung d. Voraussetzungen
    (z.B.: Brief-/Mail-Schreiben nach Klick statt Befehl-Eingabe)
    ➥ Design / Ergonomie
  - Quasi-parallele Bearbeitung ⇨ Präemptives Multitasking
    (z.B.: Erstellung v. Präsentation während Internet-Suche)
  - Arbeits-Abläufe einer Person ⇨ Prozeß-Kommunikation
    (z.B.: Datei-Löschen i. allen betroffenen Fenstern anzeigen)
    ➥ Plattform / Betriebssystem
  - Einheitliches „Look & Feel“ ⇨ Kapselung d. Ein-/Ausgabe
    (z.B.: „Drag&Drop“ programm-übergreifend realisiert)
    ➥ Applikationen / Bibliotheken

Grafische Benutzungsschnittstellen

⇨ Steigerung d. Hw-Abstraktion durch zusätzliche Sw

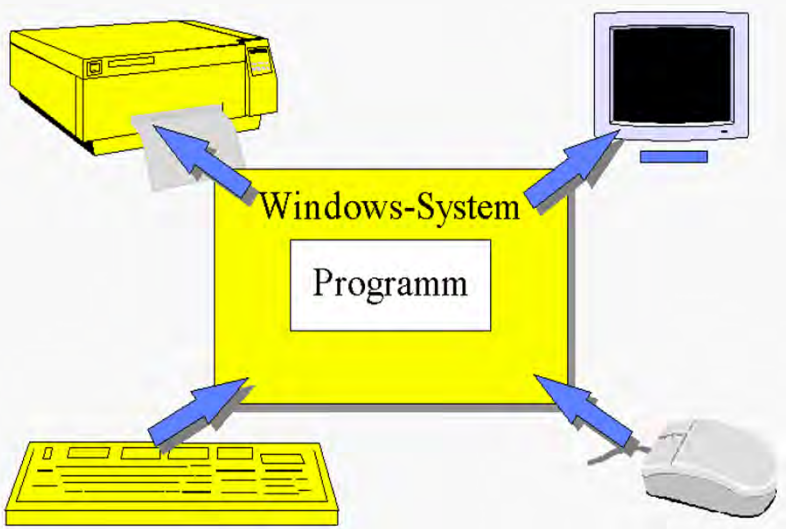
Anwendungsprogrammierung unter **MS-DOS**:

“Standard-I/O” u.standardisierte Rechenzeit-Zuweisung; abweichende Handhabung durch Programmierer/in



Windows-Programmierung:

Kommunikation mit Peripherie ins System integriert: Geräte werden “wie unter Windows” angesprochen.



Bilder: W. Doberenz, Th. Kowalski: “Programmieren lernen in Visual Basic 5”, Hanser 1997

Zur Verdeutlichung: Word for DOS auf ≥20 Disketten ausgeliefert; darunter: 1-2 für das Programm selbst, Rest für diverse Treiber

- Dialog-Komponenten immer komplexer / differenzierter
  (z.B.: gezielte Warn-Meldungen mit Fallunterscheidungen)
- Dialog (quasi-)unabhängig vom Programm
  (z.B.: Menüleisten mit: „Datei - Bearbeiten - Ansicht -...“)

⇨ Loslösung der eigentlichen Applikation von Ein-/Ausgabe

⇨ Abgabe der Ablaufkontrolle an GUI-Umgebung/-Plattform

Konsequenzen für die Applikations-Entwicklung:

- für die Ergonomie: sehr groß
- für das DV-Ergebnis: keine
- für die Sw-Konzeption: gering
- für die Codierung: deutlich (v.a. für kleine Sw-Projekte)

Zusätzlicher Codierungsaufwand durch Berücksichtigung

- der Plattform-Aufrufe
  (z.B.: Ausgabe im Fenster) - aber auch
- der plattform-eigenen Erfordernisse (Plattform-‘Eigenleben‘)
  (z.B.: Änderung der Fenstergröße, Löschung des Fensters)

Beispiel / Übung:

Beispiel-Projekt:

Sw-Umgebung SP (‚SuperProgramming') soll I/O übernehmen.

Anwendungsprogramm:

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

int main (void)
{ int  num=0, ch=' ';
  while ((ch = _getch()) != 27)
  { num++;  printf ("%d\n", num);
  }
  return (num);
}
```

(App0th.exe)

Dringend empfohlen: Trennung & Modularisierung ⇨

Beispiel / Übung:

`main()` entbehrlich halten:

```
int data=0;
/* Zu Testzwecken: */
int main (void)
{
  printf("\n\r A.Christidis:");
  printf("\n\r AnschlgZaehlg");
  printf("\n\r Ende: <Esc>");
  printf("\n\r(bel. Taste)");
  _getch();
  /*Besser–z.B.: App4Win.h mit
  #define CLS "cls" */
  system("cls"); /*system(CLS);*/

  example ();
}
```

(App1st.exe)

```
/*Beispielhafte Anwendung:*/
int example (void)
{ int ch=' ';

  while ((ch=_getch()) != 27)
  { data = calc (data);
    printf ("%5d\n", data);
  }
  return (data);
}
```

I/O

```
/*Fachlich relevant:*/
int calc (int num)
{ return (++num);
}
```

Algorithmus bleibt unberührt!

Neue I/O-Aufrufe:

```
int SPgetch(void);
int SPprintf(char *format, int data);
```

Beispiel / Übung:


```
int   data=0;     /*Daten ...*/
char *form="%5d"; /*... und ihr Ausgabe-Format*/
/* Beispielhafte Anwendung: */
int example (void)
{ int  ch=' ';
  SPinit (redisplay);
  while ((ch = SPgetch()) != 27)
  { data = calc (data);
    SPprintf (form, data);
  } return (data);
}
```

Wichtig: Funktion als Maßnahme bei Löschung des Fensters:

```
/* Anpassung an Plattform-Vorgaben: */
int redisplay (void)              /* Callback-Funktion*/
{ int j1=MAX((data-SPLINES),0);/*#define SPLINES 24*/
  while (j1 < data)
  { j1= calc (j1);  SPprintf (form, j1);
  }  return (j1);
} /* Bekanntgabe bei der Initialisierung!! */
```

Callback

Zentraler Begriff bei Errichtung o. Nutzung v. Sw-Plattformen:

➡ **Callback** [-Funktion]: Funktion d. Anwendungsprogramms, die in vorgegebenen Situationen von der Sw-Umgebung aufgerufen wird (z.B. zum Auffrischen des Fensterinhalts)

Hintergrund:

- Multitasking ⇔ Teilen von Ressourcen mit anderen (I/O, Rechenzeit etc.)
- Koordination nur durch Sw-Umgebung / -Plattform möglich (Info über laufende Programme u. Ressourcen-Bedarf)

Callback

⇨ Maßnahmen nach Wiederzuweisung von Ressourcen sind nur durch die Plattform einzuleiten – aber:

➡ Code ist nur als Bestandteil der Applikation sinnvoll: nur dort ist Information über Wiederherstellungs-Maßnahmen vorhanden – z.B.:

Uhr: . . . . . Abfrage (ggf. vorzeitig)

Bild: . . . . . Neuladen

Live-TV: . . . . . (keine Maßnahme)

Wie kann die (Standard-) Applikation der (Standard-) Plattform mitteilen, welche Funktion aufzurufen ist?

- Einrichtung einer Plattform-Routine zur Aufnahme eines Zeigers auf eine C-Funktion

(„Callback“ = Rückruf)

Callback

Zur Erinnerung - Zeiger auf C-Funktionen:

Passend (Typ, Parameterliste) zu einer Funktion – z.B.:

```
int myfunc (void)
```

kann eine Zeiger-Variable **p2f** deklariert werden,...

```
int (*p2f)(void)   //bis C99: int (*p2f)();
```

Ohne Klammer: Variable **int *p2f**

Ohne Klammer: Return-Wert **int***

Ohne **void**: bel. Parameter-Liste

... die zur Laufzeit die physikalische (Einsprungs-)Adresse der Funktion zugewiesen bekommt:

```
p2f = myfunc;
```

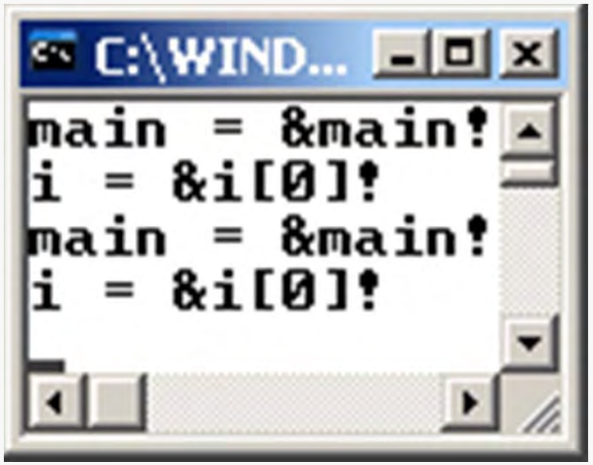
(vgl. Felder: Adreßoperator **&** unnötig)

Die Funktion kann nun über ihre Adresse aufgerufen werden:

```
(*p2f)();              //seit C99 auch: p2f();
```

Callback

Beispiel / Experiment:




```
#include <conio.h>  //wg. _getch()
#include <stdio.h>  //wg. printf()

int main(void)
{ static int i[1]={0};
  /*Funktionsname = Funktionsadresse...*/
  if (main == &main) printf ("main = &main!\n");

  /*...wie Feldname = Feldadresse:*/
  if (i == &i[0]) printf ("i = &i[0]!\n");

  /*Wiedereintritt ohne Endlos-Schleife:*/
  if (i[0]++ < 1) (*main)();     /*wie: main();*/
  _getch();
  return 0;
}
```

(Callback0\Exc\Test.exe)

Callback

Zur Erinnerung - Zeiger auf C/C++-Funktionen (2):

```
#include <stdio.h>  /*printf(), getchar()*/
```

Plattform

```
void (*funcPointer)(int)=NULL;
/* Callback-Registrierung: */
void init (void (*callback)(int))
{ printf ("In init()!\n");
  callback (0);     /*  untypisch, aber OK  */
  funcPointer = callback;
 (*funcPointer)(1); /*i.d.R. in anderen Fktn*/
  return;
}
```

Applikation

```
/* Callback: */
void myfunc(int param)
{ printf ("In myfunc() mit %d!\n", param);
  getchar();  return;
}
int main (void)
{ init (myfunc);
  return 0 ;
}
```

(Callback1\Exc\Test.exe)

Ereignisse

Verstärkter Einsatz von Callbacks in modernen Plattformen und Umgebungen führt zu weiterem Paradigmenwechsel:

Klassisch:

Programm fordert vom BS Ressourcen u. Aktionen an

(prozedurorientierte Arbeitsweise - z.B.: Single Task)

Nunmehr meist:

Betriebssystem gibt Programm Bescheid, wann es und mit welcher Funktion „an der Reihe ist”

(ereignisgesteuerte Arbeitsweise)

Beispiel:

I/O-Kontrolle ermöglicht bei gleichem **`example`** und **`calc`** veränderte Ausgabe: `(App2nd.exe)`

Ereignisse

- Definition: Als **Ereignis** *(engl. event)* bezeichnen wir jedes Vorkommnis *(occurence)*, das eine nicht-sequentielle Bearbeitung eines Programms bewirkt.

  Ein Vorkommnis kann eine Zustandsänderung (Meßwert) oder die Erfüllung einer Bedingung (`x == y`) sein.

  Ein dazugehöriges Ereignis kann die Abarbeitung einer `if`-Abfrage im Programm sein.

- **Synchrone** Ereignisse treten zu vorhersagbaren Zeitpunkten, **asynchrone** zu nicht-vorhersagbaren ein.

  Synchrone Ereignisse sind z.B. `if`-Abfragen im prozedur-orientierten Programm: sie treten vorhersagbar ein (z.B.: ab Programmbeginn).

  Asynchrone Ereignisse sind z.B. Tasten- u. Maus-Aktionen (Zeit oft nicht-vorhersagbar: „OK“ vs. „Abbrechen“).

d.h.: Programm-stelle!

Ereignisse

Ereignisgesteuerte Arbeitsweise bedient sich asynchroner Ereignisse.

Klassische Auslöser asynchroner Ereignisse:

- die Systemzeit (C-Anweisungen **`clock()`**, **`time()`**)

```
clock_t start;  start=clock();
// Rechenzeit seit Start [sec*CLOCKS_PER_SEC]
time_t  start;  time(&start);
// Wartezeit seit 01.01.1970 00:00:00 [sec]
```

- die Tastatur (Nicht-ANSI-Anweisung **`_kbhit()`**) ,
- die Maus (nicht über C ansprechbar)

und

- Kombinationen daraus.

Übung:

Aufgaben:

1. Konzipieren und codieren Sie die neue Sw-Umgebung SP so, daß Sie **`int _getch(void)`** durch **`int SPgetch(void)`**, **`int printf (const char* format, int data)`** durch **`int SPprintf (char *format, int data)`** ersetzen können.

   Zur Initialisierung erstellen Sie
   `void SPinit (int (*callback)(void))` .
2. Verändern Sie die Umgebung so, daß die Ausgabe durch Betätigung der Tasten ‚s' und ‚d' verschoben wird.
3. Verändern Sie die Umgebung so (bedingte Kompilierung), daß die Applikation nur 30 sec aktiv bleibt - bei eingeblendetem Countdown.
4. Beseitigen Sie alle globalen Variablen, die Sie evtl. verwendet haben.

`App3rd.exe`

Timer

Zeit-Abfragen in C:

- System-Konstante in `time.h` :

  `#define CLOCKS_PER_SEC  1000`

  Frühere Bezeichnung (z.T. noch gebräuchlich):

  `#define CLK_TCK  CLOCKS_PER_SEC``//in: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


```
/* 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

```
 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