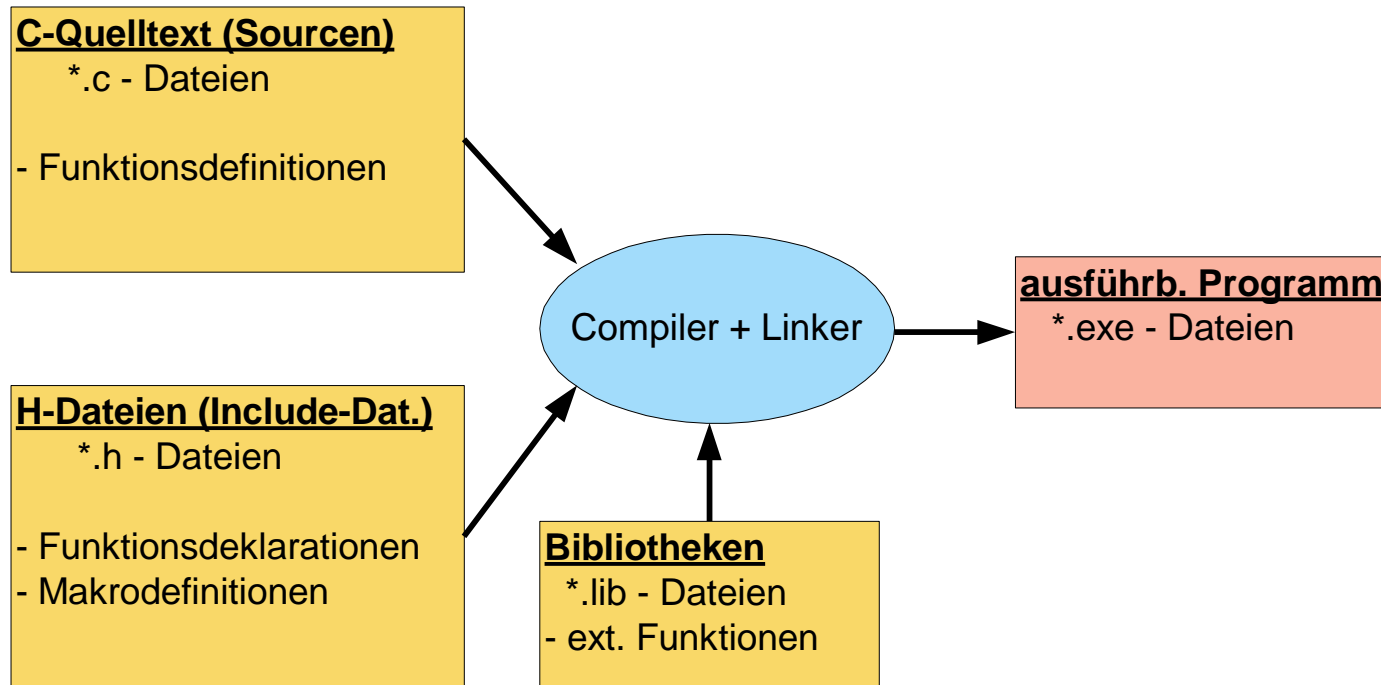




13.24 Softwareerstellung unter C

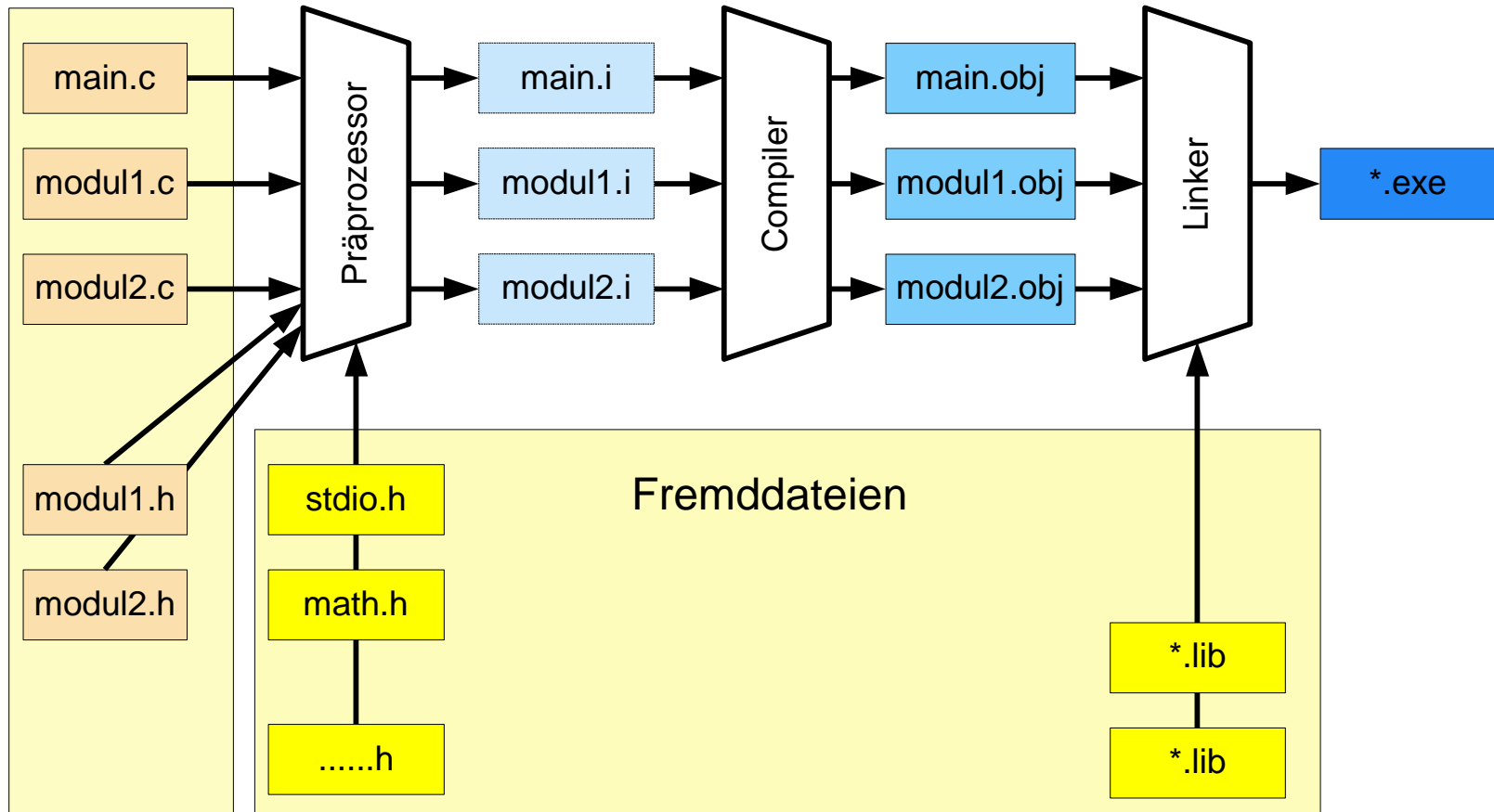
13.24.1 Grundlegende Überlegungen



Ziel: Erzeugung eines ausführbaren Programms für das Zielsystem aus dem Sourcecode.



13.24.2 Schritte zur Erzeugung eines ausführbaren Programms





13.24.3 Präprozessor

Aufgabe 1: Dateien einfügen

```
#include "filename"
```

bewirkt, dass der Inhalt der Datei <filename> geladen wird.

Anm.: <filename>
"filename"

Suche im Comp.-Defaultpfad
Suche in eigenen Verzeichnissen

Aufgabe 2: Text-Ersatz

```
#define TRUE 1
```

bewirkt, dass im Sourcecode das Wort TRUE durch 1 ersetzt wird.

Aufgabe 3: Bedingte Compilierung

```
#ifdef DOS  
.....  
#endif
```

bewirkt, dass der Sourcecode dazwischen nur compiliert wird, wenn DOS definiert ist (#define DOS).



13.24.3 Präprozessor (Fortsetzung)

Nützliche #define's:

```
#define PI      3.1415926
#define TRUE   1
#define FALSE  0
#define AND    &&
#define OR     ||
#define NOT    !
```

```
/* Macros */
```

```
#define MAX(A,B) ((A) > (B) ? (A) : (B))
#define MIN(A,B) ((A) < (B) ? (A) : (B))
```

ÜBUNG: → Programmbeispiel



ÜBUNG: Präprozessor

Wie sieht der Zwischencode (*.i - Datei) nach Ablauf des Präprozessors aus?

```
/* module_1.h */  
#define VERSION_A  
#define PRINT_1(A) printf("%lf\n", (A))  
  
double funktion1(int, int);  
double funktion2(double, double);
```

```
/* main.c */  
#include "module_1.h"  
  
int main(){  
    int    Var1=15, Var2=5;  
    double b, d1=1.0, d2=3.3;  
  
    #ifndef VERSION_A  
        b = funktion1(Var1, Var2);  
    #else  
        b = funktion2(d1, d2);  
    #endif  
  
    PRINT_1(b);  
}
```



ÜBUNG: Präprozessor, Macros

1. Geben Sie den Wert von y nach folgender Programmsequenz an.
Wie lautet das Programm nach der Textersetzung durch den Präprozessor?

```
#define SQUARE (x)    (x*x)
...
int a=3, b=4;
y = SQUARE (a+b);
```

Was kann man besser machen?

2. Geben Sie ein Macro an, welches zwei Zahlen miteinander vergleicht:

```
#define COMPARE (A, B) ???           /* = 0, wenn A=B */
                                     /* = +1, wenn A>B */
                                     /* = -1, wenn A<B */
```

Was sind die Vor- und Nachteile eines Macros?



Bedingte Compilierung

Mit den Präprozessordirektiven `#ifdef`, `#ifndef`, `#else`, `#endif` kann man steuern, dass der zwischen ihnen stehende Code nur unter der Bedingung compiliert wird, dass ein Name *name* definiert ist (**#define** *name*).

```
#ifdef name
    Programmcode 1
#else
    Programmcode 2
#endif
```

wird compiliert, wenn *name* definiert

wird compiliert, wenn *name* nicht definiert

Mit den Präprozessordirektiven `#if`, `#elif`, `#else`, `#endif` kann man steuern, dass der zwischen ihnen stehende Code nur unter der Bedingung compiliert wird, dass ein Konstantenausdruck *cexpr* ungleich 0 ist.

```
#if cexpr1
    Programmcode 1
#elif cexpr2
    Programmcode 2
#else
    Programmcode 3
#endif
```

Wird compiliert, wenn *cexpr1* ungleich 0 ist.
Alle weiteren Ausdrücke werden dann übersprungen.

Wird compiliert, wenn *cexpr2* ungleich 0 ist.

Wird compiliert, wenn *cexpr1* und *cexpr2* gleich 0 sind.



Mit dem Operator `defined(name)` kann beim Compilieren geprüft werden, ob der Name `name` definiert ist (**#define** `name`).

`defined(name)` gibt 1 zurück, wenn der Name definiert ist.
gibt 0 zurück, wenn der Name nicht definiert ist.

Der `defined`-Operator darf nur in `#if`- und `#elif`-Ausdrücken verwendet werden.

```
#if defined(name1)  
    Programmcode 1
```

Wird compiliert, wenn `name1` definiert ist.
Alle weiteren Ausdrücke werden dann übersprungen.

```
#elif defined(name2)  
    Programmcode 2
```

Wird compiliert, wenn `name2` definiert ist.
Alle weiteren Ausdrücke werden dann übersprungen.

```
#else  
    Programmcode 3
```

Wird compiliert, wenn weder `name1` noch `name2` definiert sind.

```
#endif
```



Beispiel 1: Präprozessor - Bedingte Compilierung

Beispiel: Alternative Einbindung verschiedener IO-Karten.

```
#define IO_KARTE_1    0
```

```
#define IO_KARTE_2    1
```

```
#if IO_KARTE_1
```

```
    IOVals = GetIOValues_Karte_1(); /* Implement. Karte 1 */
```

```
#elif IO_KARTE_2
```

```
    IOVals = GetIOValues_Karte_2(); /* Implement. Karte 2 */
```

```
#else
```

```
    IOVals = 0;
```

```
#endif
```



Beispiel 1: andere Möglichkeit

Beispiel: Alternative Einbindung verschiedener IO-Karten.

```
#define IO_KARTE_2

#if defined(IO_KARTE_1)
    IOVals = GetIOValues_Karte_1(); /* Implement. Karte 1 */
#elif defined(IO_KARTE_2)
    IOVals = GetIOValues_Karte_2(); /* Implement. Karte 2 */
#else
    IOVals = 0;
#endif
```



Beispiel 2: Präprozessor - Bedingte Compilierung

Beispiel: Berücksichtigung von Zielsystem-Abhängigkeiten

```
#define VAX    /* VAX, CRAY, PDP11, .... */  
#define UNIX  /* UNIX, VMS, Solaris, ....*/
```

```
#if defined(VAX) && defined(UNIX)  
    ... VAX-Code for UNIX-OS
```

```
#elif defined(VAX) && !defined(UNIX)  
    ... VAX-Code for Non-UNIX-OS
```

```
#else  
    ... usw. ...;
```

```
#endif
```



13.25 Allokierung von Speicher zur Laufzeit

Bisher wurde der benötigte Speicher immer zur Compilierzeit (statisch) festgelegt, z.B.

```
int    i, j, k;  
double Mat[3][4];
```

In vielen Aufgabenstellungen ist aber die Größe des benötigten Speichers zur Compilierzeit gar nicht bekannt.

Beispiele:

- Unterprogrammammlung zur Matrizenberechnung,
- Texteditor.

In diesen Fällen muß zur Laufzeit Speicherplatz vom Betriebssystem angefordert (*allokiert*) werden.



13.25 Allokierung von Speicher zur Laufzeit (Fortsetzung)

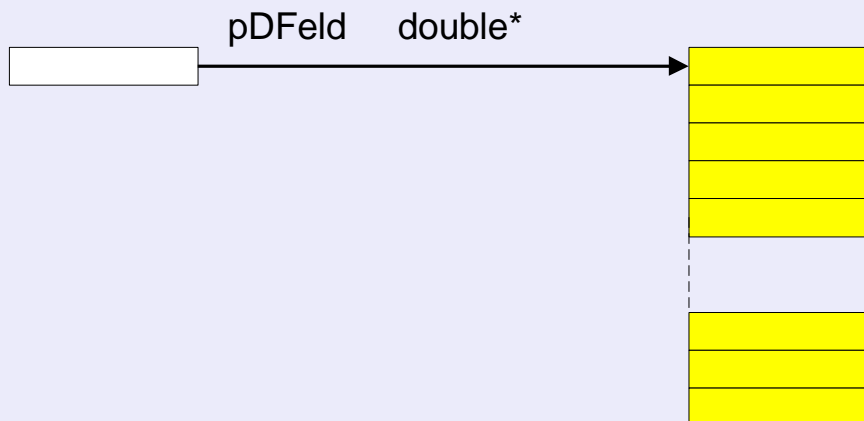
Für die Allokierung des Speichers hat C den Befehl:

```
void * malloc(size_t n);
```

- allokiert n Bytes und gibt einen Zeiger auf den allokierten Speicherbereich zurück.
- Ist die Allokierung nicht möglich, wird NULL (*Nullpointer*) zurückgegeben.
- *size_t* entspricht auf dem ARM7 dem Datentyp *unsigned int*

Beispiel:

```
double *pDFeld ;  
pDFeld = (double *) malloc (100 * sizeof(double)) ;
```





13.25 Allokierung von Speicher zur Laufzeit (Fortsetzung)

Zur Freigabe von nicht mehr benötigtem Speicherplatz gibt es den Befehl

```
void free(void *);
```

Wichtig ist, dass es sich bei dem übergebenen Zeiger tatsächlich um einen Zeiger handelt, der auf allokierten Speicher zeigt. Ansonsten stürzt die Anwendung ab !!

Für die ordnungsgemäße Speicherfreigabe hat der Programmierer zu sorgen (Ursache vieler Softwarefehler).



ÜBUNG: Dynamische Datenstrukturen → einfach verkettete Liste

Auf der Basis der folgenden Strukturen soll eine Studentenkartei aufgebaut werden.

```
struct datum {  
    int tag;  
    int monat;  
    int jahr;  
};
```

```
struct Student {  
    char name[20];  
    char vorname[20];  
    int Matrikelnummer;  
    struct datum eingeschrieben;  
    struct Student *next;  
};
```

Folgende Funktionen werden benötigt:

```
void AddStudent(char *nm, char *vrnm, int Mtnr,  
                int etag, int emon, int ejahr);
```

```
void DelStudent(int Mtnr);
```

```
void PrintStudenten();
```



ÜBUNG: (Wiederholung) Funktionsaufrufe Stringfunktionen

Zu definieren, deklarieren und aufzurufen ist eine Funktion `CharCount (...)`, welche die Zeichen eines Strings zählt:

Weiter sind folgende Daten-Deklarationen gegeben:

```
char String[]          = "einfach ein String";  
char *pString         = "String mit Pointer";  
char *Stringliste[]  = { "Will", "Gunter", "Bernhardus" };  
-----
```

Geben Sie die Definition der Funktion `CharCount` an.

Schreiben Sie ein Hauptprogramm, welches `CharCount` aufruft und die Zeichen zählt

- von `String`
- von `pString`
- vom 2. String der `Stringliste`
- vom String `"noch ein String"`.

Geben Sie einen alternativen Prototypen für `CharCount` an.



ÜBUNG: (Wiederholung) Funktionsaufrufe Stringfunktionen

Zu definieren, deklarieren und aufzurufen ist eine Funktion `StrCount`, welche die Anzahl der Strings zählt in Listen des Typs:

```
char *Stringliste[] = { "Will",  
                        "Gunter",  
                        "Bernd",  
                        "" /* Listenende */  
                      };
```

Geben Sie die Definition der Funktion `StrCount` an.

Schreiben Sie ein Hauptprogramm, welches `StrCount` aufruft.

Geben Sie einen alternativen Funktionsprototyp an.



ÜBUNG: (Wiederholung) Funktionsaufrufe Vektoren

Zu definieren, deklarieren und aufzurufen ist eine Funktion `Vektorsumme`, welche die Summe der Vektorkomponenten berechnet:

Weiter sind folgende Daten-Deklarationen gegeben:

```
int Vektor[] = {1,2,3,4};  
int *pVektor = Vektor;  
int Matrix[][3] = {1,2,3,  
                  4,5,6};
```

Geben Sie die Definition der Funktion `Vektorsumme` an.

Schreiben Sie ein Hauptprogramm, welches die Summe der Elemente berechnet:

- von `Vektor`
- von dem Vektor, auf den `pVektor` zeigt
- von der 2. Zeile von `Matrix`

Geben Sie einen alternativen Prototypen für `Vektorsumme` an.



ÜBUNG: (Wiederholung) Funktionsaufrufe Matrizen

Zu definieren, deklarieren und aufzurufen ist eine Funktion `Matrixsumme`, welche die Summe der Matrixelemente berechnet:

Weiter sind folgende Daten-Deklarationen gegeben:

```
int Matrix[][3] = {1,2,3,
                  4,5,6};
int *pMatrix[] = {Matrix[1],Matrix[0]};
```

Geben Sie die Definition der Funktion `Matrixsumme` an.

Schreiben Sie ein Hauptprogramm, welches die Summe der Elemente von `Matrix` berechnet.

Geben Sie alternative Prototypen für `Matrixsumme` an, welche die Summe der Matrix über `pMatrix` berechnen.



13.26 Zeigerarithmetik (Fortsetzung)

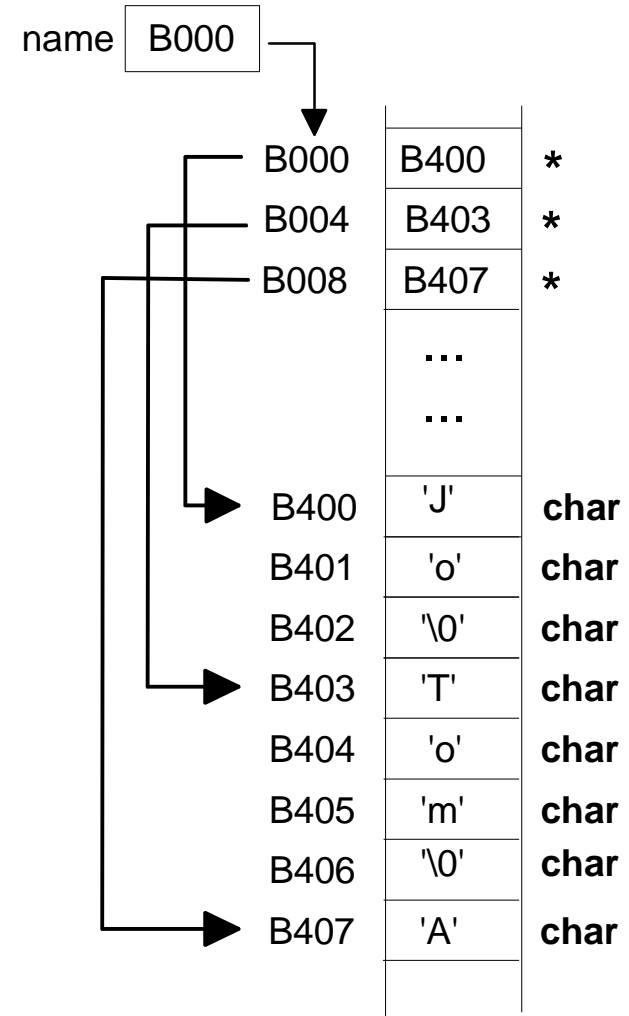
Beispiel

```
char *name[] = { "Jo"      , "Tom",
                 "Andreas", "Lou"
                };
```

```
printf ("1.Name: %s, 3.Name: %s",
        *(name),      *(name+2) );
```

/ ist äquivalent zu */*

```
printf ("1.Name: %s, 3.Name: %s",
        name[0], name[2] );
```





ÜBUNG: Zeigerarithmetik (C-Puzzle, kein ernsthafter Code)

Was gibt das folgende Programm aus ?

Zeichnen Sie zuvor die Memory-Map (Adressen s. Kommentar).

```
int a[] = {0,1,2,3,4};           /* a beginne bei 0x1000 */
int *p[] = {a,a+1,a+2,a+3,a+4}; /* p beginne bei 0x2000 */
int **pp = p;                   /* pp stehe hinter p */

main() {
    printf("%d %d %d \n",      a,    a[1],    &a[2]    );
    printf("%d %d %d \n",      *p[0],  p[0],    &p[0]    );
    printf("%d %d %d \n",      *(p[1]+2), p+3,    *p      );
    printf("%d %d %d \n",      *pp,    pp,    pp[0][4] );
    printf("%d %d %d \n",      *(*pp+1)+2), &pp,    **pp    );
}
```