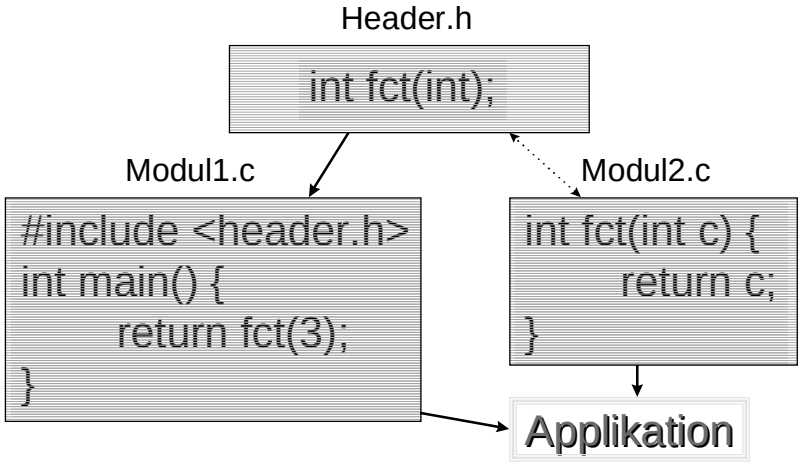
Modulkonzept und Speicherklassen

- ein C-Programm verfügt über eine Reihe von Funktionen, unter denen genau eine ausgezeichnet ist (die **`main`**-Funktion)
- Funktionen können in voneinander verschiedenen Modulen definiert werden

Modulkonzept 1



Modulkonzept 2

- Module werden mit Hilfe des Linkers zur Applikation gebunden
- der Linker wird meist direkt vom Compiler gerufen cc Modul1.c Modul2.c
- die Module können aber auch erst einzeln übersetzt werden cc -c Modul1.c
- auch gemischte Übersetzung/Bindung ist möglich cc Modul1.o Modul2.c

Die main-Funktion 1

- Eintrittspunkt in das Programm

```
int main(int argc ,char *argv[]);
```

- die optionalen Argumente bezeichnen die, auf der Kommandozeile übergebenen Argumente (als Strings) und deren Anzahl

Die main-Funktion 2

```
int main(int argc ,char *argv[]);
```

- argv[0] ist immer der Name des Programmes (mit dem vollen Pfad)
- argv[argc]==0 ist garantiert

Die main-Funktion 3

- Beispiel:

```
#include <stdio.h>
int main(int argc,char *argv[]) {
  int i;
  printf("Programm: %s\n",argv[0]);
  for (i=1;i<=argc;i++)
    printf("argv[%d]=´%s´\n",argv[i]);
  return 0;
}
```

M. und Speicherklassen 1

- gibt es bzgl. eines Moduls globale Variablen, so wird für diese beim Start des Programms Speicherplatz bereitgestellt
- globale Variablen behalten ihren Wert, auch wenn im Moment keine, im Modul erklärte Funktion abgearbeitet wird

M. und Speicherklassen 2

- auf eine globale Variable eines anderen Moduls kann nach der Deklaration

  extern *Typname Variablenname* ;

  zugegriffen werden

M. und Speicherklassen 3

- wird eine globale Variable als

  static *Typname Variablenname* ;

  definiert, so ist kein Zugriff von außen möglich
- static kann auch auf Funktionen angewandt werden

Speicherklassen 4

- normalerweise wird Platz für die lokalen Variablen einer Funktion auf dem Stack reserviert, d.h. lokale Variablen werden nach dem Verlassen der Funktion vernichtet
- wird eine lokale Variable als static definiert, so bleibt sie erhalten und behält ihren Wert

Speicherklassen 5

- C kennt darüber hinaus die Speicherklasse “register”

  register *Typname Variablenname* ;

  - “register” kann nur auf lokale Variablen und Funktionsparameter angewandt werden
  - “register” veranlaßt die Speicherung in einem Register des Prozessors

Speicherklassen 6

- Beispiel:

```
extern int xe;
extern void f3();

void f1() {
  static int sf;
}

int main() {
  int lm;
}
```

```
int xe;
static int s2;

static void f2() {
}

void fe() {
}
```

Module und Bibliotheken 1

- ein hoher Grad an Wiederverwendung von Code ist durch die Nutzung vorübersetzter Module gegeben
- ein oder mehrere Module werden oft zu Bibliotheken zusammengefaßt
  - unter UNIX gibt es dazu das Programm “ar”
    ar cr libLIBNAME.a Modul1.o ... Moduln.o
  - Funktionsprototypen und Deklarationen werden in Header-Dateien bereitgestellt

Module und Bibliotheken 2

- um den Code aus einer Bibliothek zu binden, wird der Bibliotheksname an den Compiler/Linker weitergeleitet

  cc My_Modul.c -l*My_Lib*

  der Linker sucht dann automatisch nach der Datei “lib*My_Lib*.a”

Besonderes

- C Pre-Processor
- Ein-/Ausgabe
- Standardbibliotheken
- dynamische Speicherverwaltung

Pre-Processor

- der Pre-Processor wird durch Direktiven gesteuert

  #directive dir_parameters

- Standarddirektiven: #define, #undef, #if, #ifdef, #ifndef, #else, #elif, #endif, #include

Pre-Processor 2

- Symboldefinition und Ersetzung

  #define SYMBOL Ersetzung

  ersetz an jeder Stelle im Code, in der SYMBOL auftritt SYMBOL durch die angegebene Ersetzung
- #undef SYMBOL löscht die Definition

Pre-Processor 3

- Beispiel:

```
#include <stdio.h>
#define PI 3.1415926

int main() {
        printf("Umfang=%f\n",10*PI);
        return 0;
}
```

Pre-Processor 4

- Beispiel 2:

```
#define MAX(X,Y) ( (X) > (Y) ? X : Y)
```

ersetzt

```
z=MAX(a+x,b*y) ;
```

durch

```
z=( (a+x) > (b*y) ? a+x : b*y ) ;
```


Pre-Processor 5

- bedingte Übersetzung

```
#ifdef SYMBOL1      oder    #if S0==S1
                    oder
   Code1
#elif defined SYMBOL2 oder #elif
S0==S2
   Code2
#else
   DefaultCode
#endif
```

Pre-Processor 6

- die Include-Direktive

```
#include <StandardHeader>
#include "/home/alf/project/header.h"
```

  ersetzt die Include-Zeile durch den Inhalt des Header-Files
- <> sucht Dateien im Standardsuchpfad

Pre-Processor 7

- der Pre-Processor kann i.A. auch durch Compileroptionen gesteuert werden
  - Definition von Symbolen ("#define SYMBOL")
    cc -c Modul.c -D*SYMBOL*
  - Include-Suchpfad erweitern
    cc -c Modul.c -I*PFAD*

Pre-Processor 8

- um doppelte Deklarationen zu vermeiden faßt man den Inhalt von Header-Dateien oft in

```
#ifndef HEADERNAME_
#define HEADERNAME_
  ...
#endif
```


Ein- und Ausgabe

- jedes Programm verlangt Ein- und Ausgabedaten
- I/O bedeutet allgemein Daten werden von einem bel. Gerät gelesen bzw. auf ein bel. Gerät geschrieben
- Geräte sind z.B.:
  - die Konsole, Dateien, Drucker etc.

Standard Input/Output 1

- für die interaktive Ein- und Ausgabe steht die “Konsole” zur Verfügung
- das Eingabegerät ist die Tastatur, die Ausgabe erfolgt standardmäßig über den Bildschirm
- Standardfunktionen (Dekl. in stdio.h):
  - `printf` für die formatierte Ausgabe
  - `scanf` für die formatierte Eingabe

Standard Input/Output 2

- printf - formatierte Ausgabe

```
int printf(char *format , ... );
```

- format: String mit “Steuerbefehlen”
- ein Steuerbefehl besteht immer aus

%[[ ]Länge[.NK]]Format

- Beispiel: `"%2d", "%-6.4f"`

Standard Input/Output 3

d,i	ganze Zahlen
o	natürliche Zahlen (octadezimal) ohne führende 0
x,X	natürliche Zahlen (hexadezimal) ohne “0x” bzw. “0X” am Anfang
u	unsigned
c	Zeichen (char)
s	0-terminierte Strings
f,lf	Gleitkommazahl (float, double)
e,E	Gleitkommazahl (wiss. Notation)
g,G	%e falls Exp. < -4, %f sonst
p	Pointer

Standard Input/Output 4

- Beispiel:

```
printf(”%d”,123)                  123
printf(”%5d”,123)                   123
printf(”%f”,-123.456)             -123.456
printf(”%-10.2f”,-123.456)        -123.46
printf(”%s”,”hello”)              hello
printf(”%5s”,”hello”)              hello
printf(”%.2s”,”hello”)            he
```

Standard Input/Output 5

- scanf - formatierte Eingabe

```
int scanf(char *format , ... );
```

- format: String mit “Steuerbefehlen”
- scanf überträgt die eingelesenen Daten in die, durch die Argumente referenzierten Variablen

Standard Input/Output 6

- Beispiel:

```
int i; float f; char s[256];
scanf(”%d,%f;%s”,&i,&f,&s);
```

liest und interpretiert die eingegebene Zeichenkette ”`123,45.345;abcd_bla_bla`” als:

```
i==123
f==45.345
s==”abcd_bla_bla”
```

Datei Ein-/Ausgabe 1

- aus der UNIX-Welt kann man zitieren: “Alles ist Datei.”
- E/A auf Dateien läuft analog zu E/A auf der Konsole
- da beim Dateizugriff Informationen über Dateiname, Zugriffsmodus etc. benötigt werden, führt stdio.h einen speziellen Typ “FILE” ein

Datei Ein-/Ausgabe 2

- um eine Variable vom FILE-Typ mit einer Datei auf dem Filesystem zu verknüpfen wird die Funktion

```
FILE *fopen(char *name,char *mode);
```

verwendet

- `name` ist der Dateiname
- `mode` ist der Zugriffsmodus

Datei Ein-/Ausgabe 3

- Zugriffsmodi:
  - ”r” Lesezugriff
  - ”w” Schreibzugriff (überschreibend)
  - ”a” Schreibzugriff (anhängend)

```
FILE *f1,*f2,*f3;
f1=fopen(”README”,”r”);
f2=fopen(”WRITEME”,”w”);
f3=fopen(”APPENDME”,”a”);
```


Datei Ein-/Ausgabe 4

- Öffnen einer nicht existierende Datei mit “r” produziert einen Fehler (NULL)
- beim Öffnen mit “w” oder “a” wird die Datei neu angelegt

Datei Ein-/Ausgabe 5

- Lesen eines Zeichens

```
int getc(FILE *fp);
```

- Schreiben eines Zeichens

```
int putc(int c,FILE *fp);
```

- im Fehlerfall liefern **`getc`** und **`putc`** EOF (Dateiende)

Datei Ein-/Ausgabe 6

- für die formatierte Ein- und Ausgabe gibt es die Funktionen

```
int fprintf(FILE *fp,char *format,...);
```

und

```
int fscanf(FILE *fp,char *format,...);
```

(erklärt in stdio.h)

Datei Ein-/Ausgabe 7

- ist die Arbeit an einer Datei beendet, so muß sie geschlossen werden
- Schließen einer Datei bewirkt insbesondere das Update des (gepufferten) Filesystems, Änderungen sind erst nach dem Schließen garantiert
- stdio.h erklärt hierfür die Funktion

```
int fclose(FILE *fp);
```


Datei Ein-/Ausgabe 8

```
#include <stdio.h>
main() {
  int i,j;
  FILE *f;
  f=fopen("Einmal-Eins.txt","w");
  for (i=1;i<=10;i++) {
    for (j=1;j<=i;j++)
      fprintf(f,"%3d ",i*j);
    fprintf(f,"\n");
  }
  fclose(f);
}
```

Datei Ein-/Ausgabe 9

- stdio.h definiert zwei spezielle Konstanten vom Typ FILE*
  - `FILE *stdin`, die Standardeingabe
  - `FILE *stdout`, die Standardausgabe
- `printf("Hallo\n");`

  ist äquivalent zu

  `fprintf(stdout,"Hallo\n");`

UNIX System Calls 1

- low-level file I/O:
  - unter UNIX erhält jede Datei einen "filedescriptor" vom Typ int, der mit

    `int open(char *name,int flags,int perm);`

    zugewiesen wird
    - » flags: O_RDONLY, O_WRONLY, O_RDWR
    - » perm: permissions gemäß den UNIX filesystem permissions
  - Deklarationen in: stdio.h, syscalls.h, fcntl.h

UNIX System Calls 2

- low-level file I/O:
  - nicht existierende Dateien müssen explizit mit

    ```
    int create(char *name,int perm);
    ```

    erzeugt werden
  - gelesen n Byte in den mit

    ```
    int read(int fd,char *buffer,int n);
    ```
  - geschrieben werden n Byte mit

    ```
    int write(int fd,char *buffer,int n);
    ```


UNIX System Calls 3

- low-level file I/O:
  - Schließen mit: `int close(int fd);`
  - Löschen mit: `int unlink(char *name);`
  - UNIX unterstützt den wahlfreien Zugriff auf Dateien:

    ```
    int lseek(int fd,long offset,int origin);
    ```

    » offset ist die Anzahl der Bytes relativ zur Position

    » origin (Position) kann sein: 0 - Dateianfang
    1 - aktuelle Pos.
    2 - Dateiende

mathematische Funktionen

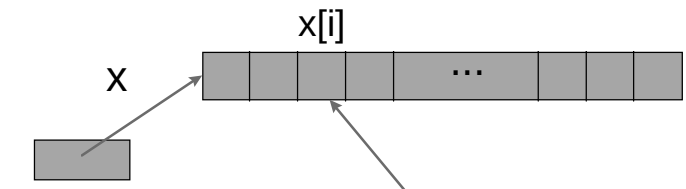
- deklariert in math.h (Argumente und Funktionswerte sind vom Typ double):
  - `sin(x), cos(x), atan2(x)`
  - `exp(x)`
  - `log(x), log10(x)`
  - `pow(x,y)`
  - `sqrt(x)`
  - `fabs(x)`
- libm.a muß gebunden werden (-lm)

String Funktionen

- deklariert in string.h:
  - char *strcat(s,t); hängt t an s an
  - int strcmp(s,t); vergleicht s und t (*s<*t → <0, *s==*t → =0, *s>*t → >0)
  - char *strcpy(s,t); kopiert t nach s (der Inhalt, der durch t referenzierten Stelle wird an die, durch s ref. Stelle übertragen
  - int strlen(s) gibt die Anzahl der Zeichen zurück (ohne “Terminator-0”)

Zeigerarithmetik

Typ x[...]; Typ *xptr;



xptr=&(x[0]);
xptr=x;

*(xptr+i)==x[i]

dynamische Speicherverwaltung 1

- zur effizienten Verwaltung vorhandener Ressourcen gehört auch eine effektive Speicherauslastung
- Code sollte keine logischen Grenzen in der Problemgröße kennen, etwa bei Gleichungssystemlösern für A*x=b;

```
float A[50][50],x[50],b[50];
```

dynamische Speicherverwaltung 2

- durch die Verwendung von Referenzen (Zeigern) können diese Probleme gelöst werden
  - das System oder das Laufzeitsystem reserviert hierzu auf Anforderung einen Platz im Freispeicher (Heap) und gibt eine Referenz auf diesen Bereich zurück
  - der Typ der Referenz bestimmt den Typ des bezeichneten Datums

dynamische Speicherverwaltung 3

- jedes Datum belegt Platz im Speicher
- der Platz wird automatisch bei allen nicht-dynamischen Variablen reserviert

```
float x; double A[10][10];
const char text[]="Hello world.\n";
```

- dieser Platz kann auch vom Nutzer *allokiert* werden

dynamische Speicherverwaltung 4

- angeforderter Speicherplatz wird grundsätzlich linear und zusammenhängend bereitgestellt
- wird Platz für mehr ein Datum des selben Typs reserviert, so repräsentiert der allokierte Speicher ein Feld, entsprechend kann auf die Elemente mittels Indizierung zugegriffen werden

dynamische Speicherverwaltung 5

- Beispiel: es wird ein Vektor mit n float Elementen benötigt (n*sizeof(float) Byte)
- malloc fordert den Speicher an

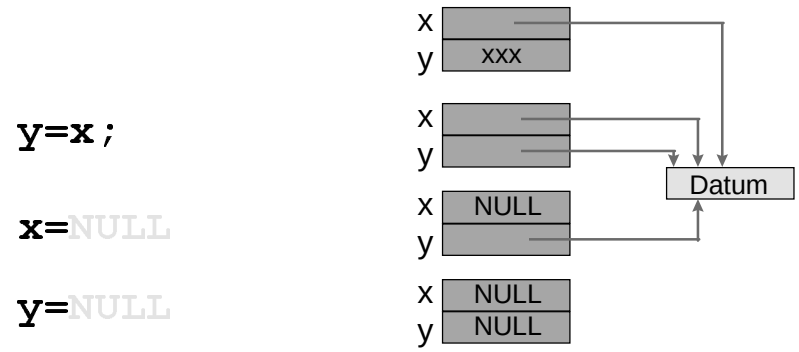
```
float *x
x=(float*)malloc(n*sizeof(float))
for (i=0;i<n;i++)
     x[i]=0;
```

dynamische Speicherverwaltung 6

- wird der Code der vorherigen Folie in einer Funktion ausgeführt, so wird der reservierte Bereich beim Verlassen der Funktion nicht freigegeben, d.h. die darin gespeicherten Daten bleiben erhalten
- die Freigabe des Speichers muß explizit erfolgen: `free(x);`

dynamische Speicherverwaltung 7

```
Typ *x,*y;
x=(Typ*)malloc(sizeof(Typ));
```



dynamische Speicherverwaltung 8

- malloc und free sind Funktionen der Standard-Bibliothek
- die Prototypen für malloc und free sind in “stdlib.h” deklariert

```
void *malloc(size_t size);
void free(void *ptr);
```


void-Zeiger

- void* als Referenz symbolisiert den untypisierten Zeiger und kann damit jedes Datum referenzieren
- die Konversion von typisierten Zeigern in void-Zeiger und umgekehrt erfolgt ohne Informationsverlust

```
void *to_any_date;
```