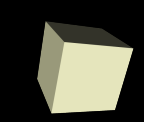
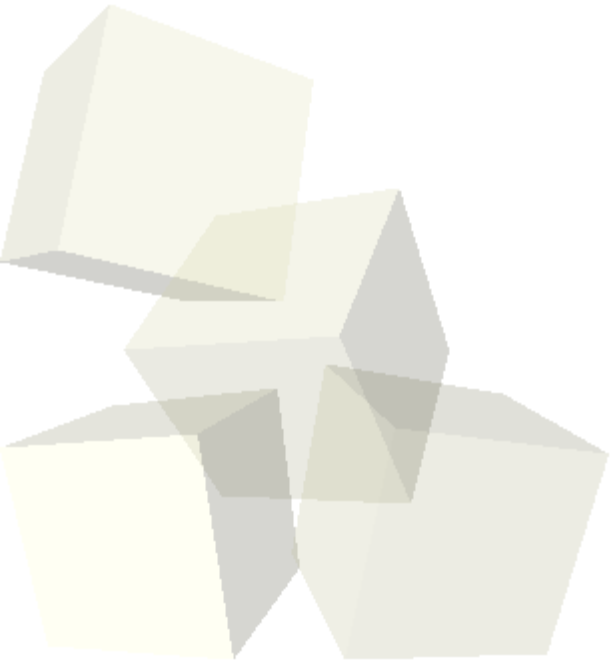


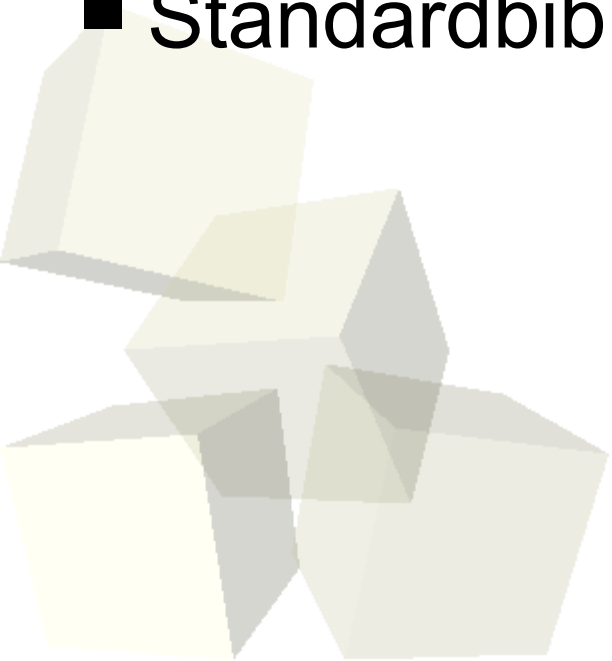
C-Schulung Tag 3

Herzlich willkommen



Agenda

- Primitive Datentypen
- Operatoren
- Kontrollstrukturen
- Funktionsdeklarationen
- Linker
- Header-Dateien
- Präprozessor
- Standardbibliothek



Primitive Datentypen

typename	minimum	maximum
signed char	SCHAR_MIN	SCHAR_MAX
unsigned char	0	UCHAR_MAX
char	CHAR_MIN	CHAR_MAX
signed short	SHRT_MIN	SHRT_MAX
unsigned short	0	USHRT_MAX
signed int	INT_MIN	INT_MAX
unsigned int	0	UINT_MAX
signed long	LONG_MIN	LONG_MAX
unsigned long	0	ULONG_MAX
float	FLT_MIN	FLT_MAX
double	DBL_MIN	DBL_MAX
long double	LDBL_MIN	LDBL_MAX

Beispielarchitektur

typename	minimum	maximum	
signed char	-128	+127	
unsigned char	0	255	
char	-128	+127	
signed short	-32768	+32767	
unsigned short	0	65535	
signed int	-2147483648	+2147483647	
unsigned int	0	4294967295	
signed long	-2147483648	+2147483647	
unsigned long	0	4294967295	
float	10^-38	10^+38	7 Ziffern
double	10^-308	10^+308	16 Ziffern
long double	10^-4932	10^+4932	34 Ziffern

Wo ist boolean?

- Zahlen werden als Wahrheitswerte interpretiert:
  - 0 ist false.
  - alles andere ist true.
- Dadurch vereinfachen sich manche Bedingungen:
  - if (x != 0) ...
  - if (x) ...
- Es schleichen sich aber auch subtile Fehler ein:
  - if (a = b)
  - if (a == b)
- Logische Operatoren liefern nur 0 oder 1.

- In C++ gibt es den Datentyp bool.
- In C99 gibt es den Datentyp _Bool.

Operatoren

- C bietet 47 Operatoren in 15 Präzedenzstufen.
- Wir besprechen hier nur ein paar interessante.
- Eine komplette Übersicht gibt es als Handout.

Fallunterscheidung

- Neben der klassischen Fallunterscheidung auf Anweisungsebene (führe dies oder das aus) gibt es auch eine Fallunterscheidung auf Ausdrucksebene (werte dies oder das aus).

- c ? x : y

- Falls c zu wahr ausgewertet wird, liefert x das Ergebnis, ansonsten y.

- vorzeichen = (i < 0) ? -1 : +1;

Boolesche Operatoren

- x && y
  - Wenn x zu false ausgewertet wird, ist das Ergebnis false, und y wird nicht mehr ausgewertet.
  - Sonst wird y ausgewertet und als Ergebnis geliefert.
  - x ? y : 0
- x || y
  - Wenn x zu true ausgewertet wird, ist das Ergebnis true, und y wird nicht mehr ausgewertet.
  - Sonst wird y ausgewertet und als Ergebnis geliefert.
  - x ? 1 : y

Sequenz

- Sequenz auf Anweisungsebene:
  - s; t;
- Sequenz auf Ausdrucksebene:
  - x, y
- x wird ausgewertet, dann wird das Ergebnis verworfen (man interessiert sich also nur für die Seiteneffekte von x), dann liefert y das Ergebnis.
- for (i = 0, k = n - 1; i < k; i = i + 1, k = k - 1)
- while (c = getchar(), c >= '0' && c <= '9')

Inkrement

- Die folgenden Ausdrücke sind äquivalent:
  - i = i + 1
  - i += 1
  - ++i ← “Präfix-Inkrement”
- Alle drei Ausdrücke erhöhen die Variable i und liefern den neuen Wert zurück.
- Will man stattdessen den alten Wert von i haben, verwendet man das “Postfix-Inkrement”:
  - i = 0; printf("%d\n", i++);
- Interessiert man sich nicht für den Wert, gibt es keinen Unterschied zwischen ++i und i++.

Kontrollstrukturen

- Java hat die Kontrollstrukturen fast 1:1 von C übernommen, daher nur eine knappe Aufzählung.
- Sequenz
  - a; b; c;
- Fallunterscheidung
  - if
  - if/else
- Auswahl
  - switch/case
- Wiederholung
  - while
  - for
  - do/while

Funktionsdeklarationen

- Bisher haben wir Funktionen *definiert*, d.h. sowohl Signatur als auch Implementation vorgegeben:
  - `int min(int a, int b) { return (a < b) ? a : b; }`
- Es ist aber auch möglich, lediglich die Signatur einer Funktion ohne Implementation festzulegen. Man spricht dann von einer *Funktionsdeklaration*:
  - `int min(int a, int b);`
  - `int max(int, int);  /* Parameternamen optional */`
- Funktionsdeklarationen ermöglichen das Aufrufen von Funktionen, die:
  - unterhalb des Aufrufs definiert sind.
  - in einer anderen Datei definiert sind.

Single-Pass-Compiler

- Der eigentliche Kompiliervorgang geschieht stur von oben nach unten: jeder verwendete Name sollte bereits deklariert oder definiert sein.

```
float absolut(float x)
{
    return (x < 0) ? -x : x;
}

int main(void)
{
    printf("%f\n", absolut(-4.25f));
    return 0;
}
```

Single-Pass-Compiler

- Der eigentliche Kompiliervorgang geschieht stur von oben nach unten: jeder verwendete Name sollte bereits deklariert oder definiert sein.

```
int main(void)
{
    printf("%f\n", absolut(-4.25f));
    return 0;
}

float absolut(float x)
{
    return (x < 0) ? -x : x;
}
```

```
error: conflicting types for 'absolut'

error: previous implicit declaration of
'absolut' was here
```

Single-Pass-Compiler

- Der eigentliche Kompiliervorgang geschieht stur von oben nach unten: jeder verwendete Name sollte bereits deklariert oder definiert sein.

```
/* Funktionsdeklaration */
float absolut(float);

int main(void)
{
    printf("%f\n", absolut(-4.25f));
    return 0;
}

float absolut(float x)
{
    return (x < 0) ? -x : x;
}
```

Single-Pass-Compiler

- Deklarierte aber nicht definierte Namen führen zu einem Linker-Fehler.

```
/* Funktionsdeklaration */
float absolut(float);

int main(void)
{
    printf("%f\n", absolut(-4.25f));
    return 0;
}
```

```
undefined reference to 'absolut'
ld returned 1 exit status
```

Single-Pass-Compiler

- Deklaration und Definition einer Funktion können in verschiedenen Dateien stehen, der Linker sorgt dann für die korrekte Verbindung.

haupt.c

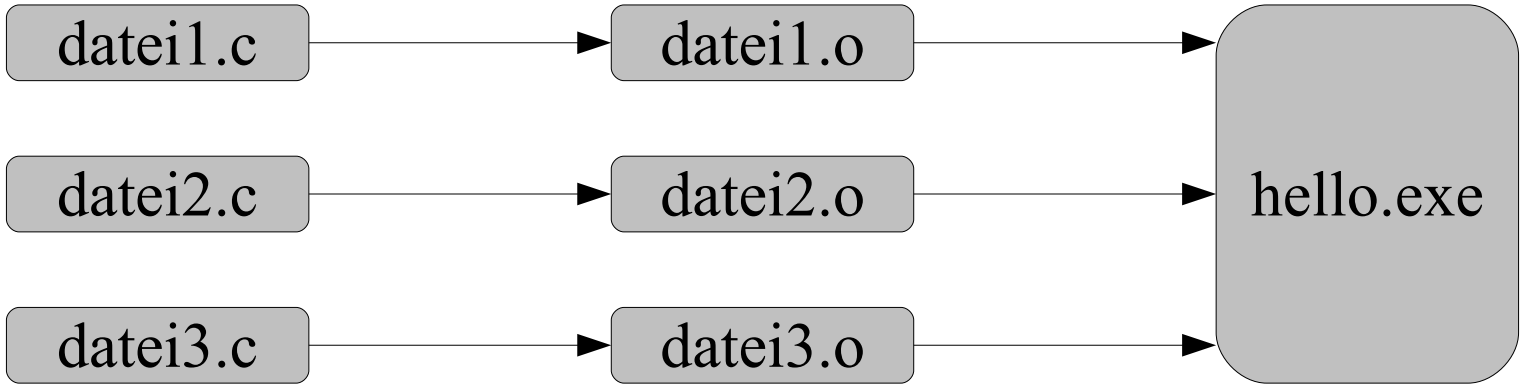
```
/* Funktionsdeklaration */
float absolut(float);

int main(void)
{
    printf("%f\n", absolut(-4.25f));
    return 0;
}
```

unter.c

```
/* Funktionsdefinition */
float absolut(float x)
{
    return (x < 0) ? -x : x;
}
```

Kompilierungsmodell

- Der Compiler erzeugt aus Textdateien mit der Endung .c Binärdateien mit sog. *Objektcode*.
- Der Linker verbindet letztere zu einem Programm.



Header-Dateien

- Angenommen, wir schreiben eine Datei bib.c mit vielen praktischen Funktionsdefinitionen:
  - absolut, stringlaenge, kopierestring...
- Dann muss jeder Aufrufer zunächst sämtliche Funktionen deklarieren, die er benutzen möchte.
- Bei mehr als einem Aufrufer führt dies zu sehr vielen redundante Deklarationen.
- Bei Änderungen in bib.c müssen die betroffenen Deklarationen aller Aufrufer angepasst werden.
- Gibt es eine Möglichkeit, die Deklarationen an einer Stelle zusammenzufassen und allen Aufrufern leicht zur Verfügung zu stellen?

Header-Dateien

bib.c

```
int absolut(int x)
{
    return (x < 0) ? -x : x;
}

int stringlaenge(char s[ ])
{
    int i;
    for (i = 0; s[i] != '\0'; ++i);
    return i;
}

int kopierestring(char to[ ], char from[ ])
{
    int i;
    for (i = 0; to[i] = from[i]; ++i);
}
```

bib.h

```
int absolut(int);
int stringlaenge(char [ ]);
int kopierestring(char [ ], char [ ]);
```

benutzer1.c

```
#include "bib.h"

/* jetzt sind die Funktionen bekannt */
```

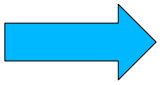
benutzer2.c

```
#include "bib.h"

/* jetzt sind die Funktionen bekannt */
```

Präprozessor

- Was genau bewirkt eigentlich das #include ?
- Es handelt sich um eine Präprozessor-Direktive.
- Der Präprozessor verändert den Quelltext, bevor der eigentliche Kompiliervorgang beginnt.
- #include "*file*" bewirkt, dass der Inhalt der Datei *file* in den aktuellen Quelltext hineinkopiert wird.

```
#include "bib.h"


int main(void)
{
    int antwort = absolut(-42);
}
```

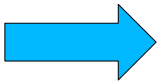


```
int absolut(int);
int stringlaenge(char [ ]);
int kopierestring(char [ ], char [ ]);

int main(void)
{
    int antwort = absolut(-42);
}
```

Präprozessor

- Eine weitere Direktive ist #define *name text*, mit der wir symbolische Konstanten festgelegt hatten.
  - Der Präprozessor ersetzt einfach alle Vorkommen von *name* im Quelltext durch *text*.
- Makros können auch Argumente haben:
  - #define absolut(x) ((x) < 0 ? -(x) : (x))
- Der Präprozessor macht reine Textersetzung, die eigentliche Programmiersprache C kennt er nicht!

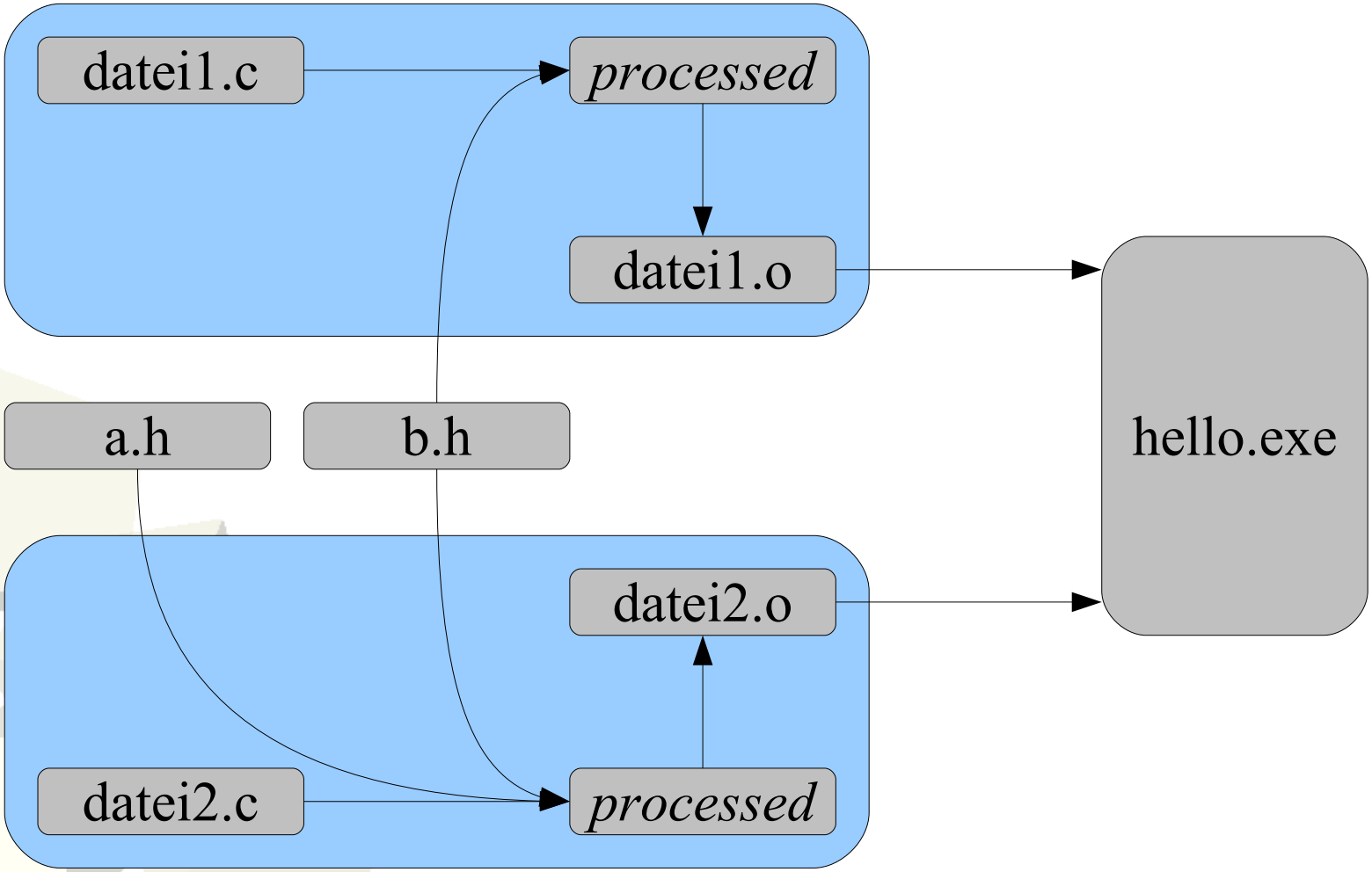
```
#define losgehts int main(void) {
#define daswars return 0; }
#define schreibe(x) printf(#x);

losgehts
schreibe(hallo welt)
daswars
```



```
int main(void) {
printf("hallo welt");
return 0; }
```

Kompilierungsmodell

- Der Präprozessor überarbeitet den Quelltext, bevor Compiler und Linker in Aktion treten.



Standardbibliothek

- Besteht aus 15 Header-Dateien
  - <assert.h>
  - <ctype.h>
  - <errno.h>
  - <float.h>
  - <limits.h>
  - <locale.h>
  - <math.h>
  - <setjmp.h>
  - <signal.h>
  - <stdarg.h>
  - <stddef.h>
  - <stdio.h>
  - <stdlib.h>
  - <string.h>
  - <time.h>

<assert.h>

- Validierung des Programms zur Laufzeit
- Java hat assert als eigene Kontrollstruktur:
  - assert expression;
- In C verwendet man stattdessen ein Makro:
  - assert(expression);

```
#include <assert.h>

void sort(int a[ ], int n)
{
    int i, k, s = 0;
    for (i = 0; i < n; ++i) s += a[i];
    for (i = 0; i < n-1; ++i)
        for (k = i+1; k < n; ++k)
            if (a[i] > a[k]) swap(a, i, k);
    assert(a[0] <= a[n-1]);   /* Haben wir wirklich aufsteigend sortiert? */
    for (i = 0; i < n; ++i) s -= a[i];
    assert (s == 0);   /* Ist die Summe gleich geblieben? */
}
```

<ctype.h>

- Funktionen zur Klassifizierung und Umwandlung:
  - int islower(int)
  - int isupper(int)
  - int isalpha(int)
  - int isdigit(int)
  - int isalnum(int)
  - ...
  - int tolower(int)
  - int toupper(int)

```
void strupper(char s[ ])
{
    int i;
    for (i = 0; s[i] != '\0'; ++i)
        s[i] = toupper(s[i]);
}
```

<limits.h>

- Implementationsabhängige Grenzen:
  - INT_MAX
  - INT_MIN
  - UINT_MAX
  - SHORT_MAX
  - SHORT_MIN
  - USHORT_MAX
  - LONG_MAX
  - LONG_MIN
  - ULONG_MAX
  - ....

- Funktionen zur Behandlung von Zeichenketten:
  - int strlen(const char s[ ])
  - void strcpy(char dst[ ], const char src[ ])
  - void strcat(char dst[ ], const char src[ ])
  - int strcmp(const char x[ ], const char y[ ])
  - ...

```
int main(void)
{
    char a[ ] = "hello ";
    char b[ ] = "world!";
    char c[13];
    assert(sizeof(c) >= strlen(a) + strlen(b) + 1);

    strcpy(c, a);
    strcat(c, b);
    puts(c);
    return 0;
}
```

- Datum und Uhrzeit
- Zeitmessung:
  - clock_t
  - clock(void)
  - CLOCKS_PER_SEC

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

int main(void)
{
    clock_t a, b;
    a = clock();
    expensive_function();
    b = clock();
    printf("Dauer: %.3f Sekunden\n", (double)(b-a) / CLOCKS_PER_SEC);
    return 0;
}
```

Standardbibliothek

- Häufig benötigte Funktionen gibt es oft bereits in der Standardbibliothek. Erfinde das Rad nicht neu!
  - strlen, toupper
- Implementationsabhängige Größen:
  - INT_MAX, CLOCKS_PER_SEC
- Um bestimmte Komponenten zu benutzen, muss man nur die passenden Header-Dateien einfügen:
  - #include <assert.h>
  - #include <ctype.h>
  - #include <string.h>
- Der entsprechende Bibliothekscode liegt bereits kompiliert vor und wird automatisch mitgelinkt.

Kompilierungsmodell

- Der Linker bindet auch vorkompilierten Bibliothekscode mit ein.