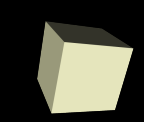
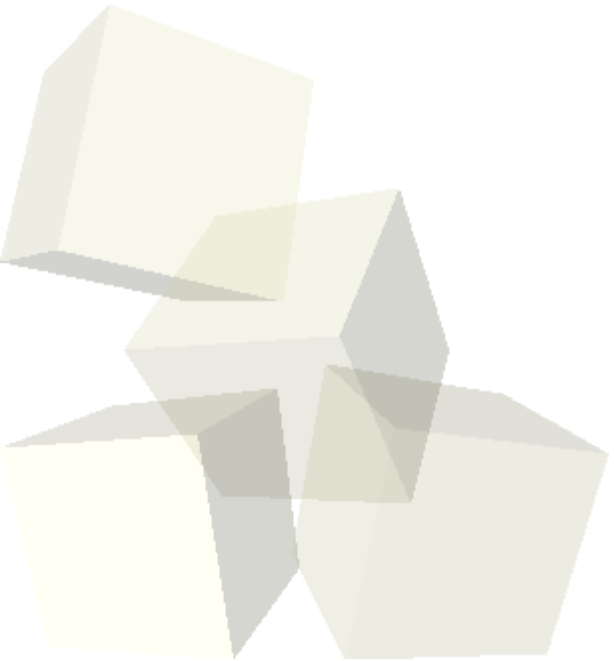


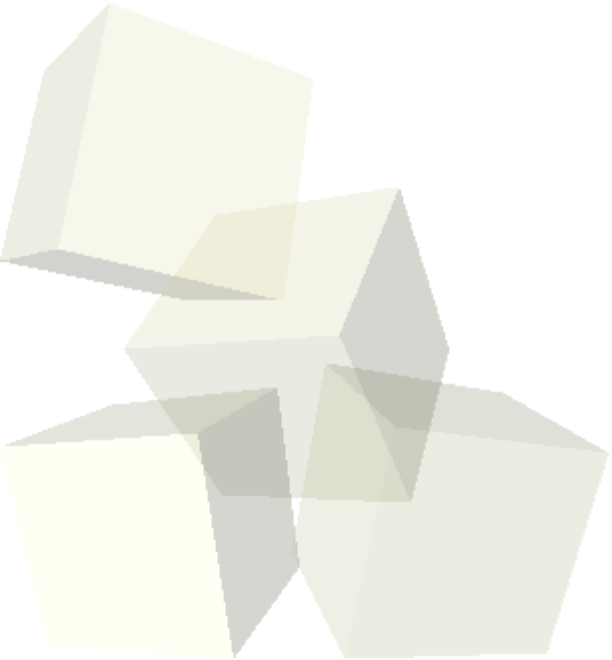
C-Schulung Tag 5

Herzlich willkommen



Agenda

- Funktionszeiger
- Deklarator Syntax
- typedef
- enum
- struct



Motivation Funktionszeiger

```
void print_binary_stupid(unsigned x);
void print_binary(unsigned x);
void print_binary_buffered(unsigned x);
void print_binary_lut(unsigned x);
void print_binary_no_leading_zeros(unsigned x);

void binary_sequence(int from, int to)
{
    int i;
    for (i = from; i <= to; ++i)
    {
        print_binary_stupid(i);   /* Hier legen wir uns auf eine Implementation fest :( */
        newline();
    }
}

void test_print_binary(void)
{
    binary_sequence(-16, +16);
}
```

Motivation Funktionszeiger

```
void print_binary_stupid(unsigned x);
void print_binary(unsigned x);
void print_binary_buffered(unsigned x);
void print_binary_lut(unsigned x);
void print_binary_no_leading_zeros(unsigned x);

void binary_sequence(int from, int to, void (* print)(unsigned))
{
    int i;
    for (i = from; i <= to; ++i)
    {
        (*print)(i);
        newline();
    }
}

void test_print_binary(void)
{
    binary_sequence(-16, +16, &print_binary_stupid);
    binary_sequence(-16, +16, &print_binary_lut);
}
```

Syntaktischer Zucker

```
void print_binary_stupid(unsigned x);
void print_binary(unsigned x);
void print_binary_buffered(unsigned x);
void print_binary_lut(unsigned x);
void print_binary_no_leading_zeros(unsigned x);

void binary_sequence(int from, int to, void print(unsigned))
{
    int i;
    for (i = from; i <= to; ++i)
    {
        print(i);
        newline();
    }
}

void test_print_binary(void)
{
    binary_sequence(-16, +16, print_binary_stupid);
    binary_sequence(-16, +16, print_binary_lut);
}
```

Generische Algorithmen

In so gut wie allen Programmen müssen Datenmengen sortiert und durchsucht werden. Die Standardbibliothek bietet dafür zwei generische Algorithmen an:

```
void qsort(void * base,   /* Wo fängt das Array im Speicher an? */
           size_t num,    /* Aus wievielen Zellen besteht es? */
           size_t size,   /* Wie groß ist eine einzelne Zelle? */
           int (* comparator)(const void * x, const void * y));
           /* In welcher Ordnung stehen zwei Elemente zueinander? */

void * bsearch(const void * key,     /* Nach welchem Element suchen wir? */
               const void * base,    /* Wo fängt das Array im Speicher an? */
               size_t num,           /* Aus wievielen Zellen besteht es? */
               size_t size,          /* Wie groß ist eine einzelne Zelle? */
               int (* comparator)(const void * x, const void * y));
               /* In welcher Ordnung stehen zwei Elemente zueinander? */
```

Der ungetypte Zeiger void* gibt dabei den Beginn eines Arrays im Speicher an, weiß jedoch nicht, von welchem Typ die Elemente in dem Array sind. Diese fehlende Typinformation muss in der comparator-Funktion hinzugefügt werden.

Generische Algorithmen

```
int ascending(const void * x, const void * y)
{
    const int * p = x;   /* Wir wissen, dass wir ints vergleichen sollen. */
    int i = *p;

    int j = *(const int *)y;   /* kompaktere Schreibweise (“Cast”) */

    return (i > j) - (i < j);
}

int main(void)
{
    int a[ ] = {3, 7, 1, 2, 1, 8, 3};

    qsort(a, sizeof(a) / sizeof(a[0]), sizeof(a[0]), ascending);

    print_numbers(a, sizeof(a) / sizeof(a[0]));

    return 0;
}
```

Deklarator Syntax

- Bisher können wir deklarieren: Einfache Variablen
  - int i;
- Arrays
  - int a[26];
- Zeiger
  - int * p;
- Funktionen
  - int f(int, int);
- Funktionszeiger
  - int (* pf)(int, int);

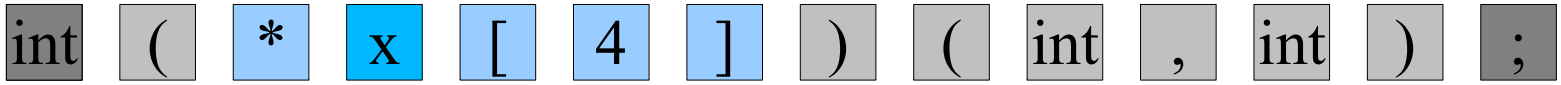
Deklarator Syntax

- Die Kombinationsmöglichkeiten sind unendlich:
  - int (*x[4])(int, int);
- Aber wie liest man sowas???

- Man deklariert Namen in C so, wie man sie innerhalb von Ausdrücken verwenden würde.
- Was bedeutet die Deklaration int *p; ?
  - p ist vom Typ int* (bisheriges Verständnis)
  - *p ist vom Typ int (Deklarator Syntax)

- Für komplexere Deklarationen muss man die Operatorpräzedenzen kennen.
  - Dafür gibt es das Handout von Tag 2 :)

Deklarator Syntax



Deklarator Syntax

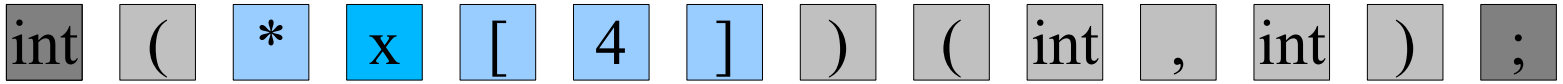
int (* x [4]) (int , int) ;

[ ] bindet stärker als *

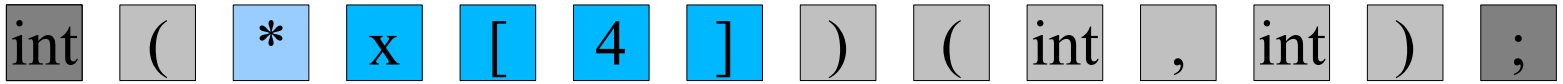
int (* x [4]) (int , int) ;

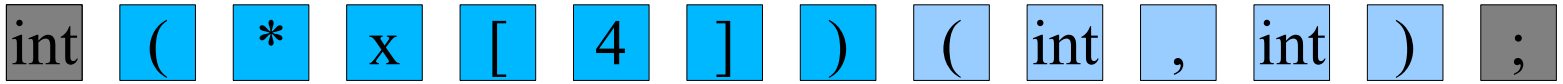
x ist ein Array der Größe 4

Deklarator Syntax



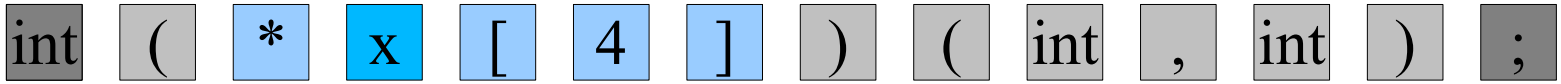
[ ] bindet stärker als *



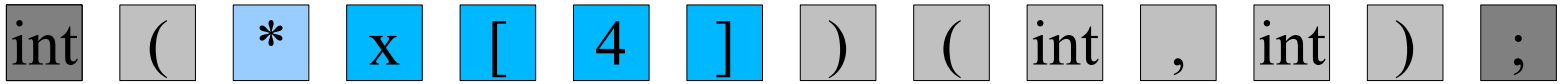


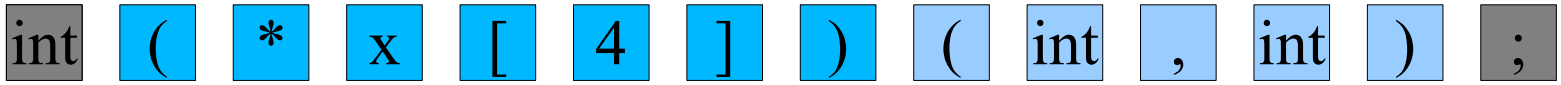
x ist ein Array von 4 Zeigern

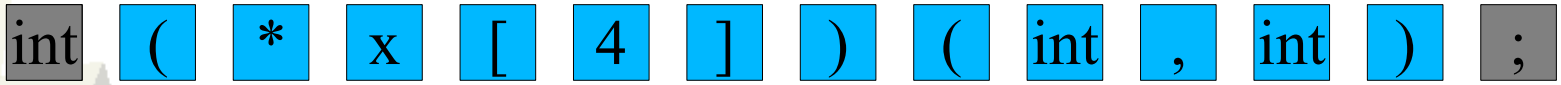
Deklarator Syntax



[ ] bindet stärker als *







x ist ein Array von 4 Zeigern auf Funktionen,
die (int, int) entgegennehmen und int zurückliefern.

cdecl

C gibberish ↔ English

```
int (*x[4])(int, int)
```

```
declare x as array 4 of pointer to function (int, int)
returning int
```

typedef

- x ist ein Array von 4 Zeigern auf Funktionen, die (int, int) entgegennehmen und int zurückliefern.
  - int (* x[4])(int, int);
- Das Verständnis fällt leichter, wenn wir für Zwischentypen neue Namen einführen:
  - typedef int (* binary_function)(int, int);
  - binary_function x[4];
- typedef ist auch für einfache Typen interessant:
  - typedef unsigned char byte;
- typedef definert (entgegen des Namens) keinen neuen Typ, sondern nur ein Typ*synonym*.

- enum ermöglicht die Definition eines Datentyps durch Aufzählungen seiner möglichen Werte:

```
enum richtung { NORDEN, SUEDEN, WESTEN, OSTEN };

enum richtung aufgang = OSTEN;

void go(enum richtung r)
{
    switch (r)
    {
        case NORDEN: --y; break;
        case SUEDEN: ++y; break;
        case WESTEN: --x; break;
        case  OSTEN: ++x; break;
    }
}
```

- Ein Aufzählungstyp kann zusammen mit Variablen desselben Typs deklariert werden:

```
enum richtung { NORDEN, SUEDEN, WESTEN, OSTEN } aufgang = OSTEN;



void go(enum richtung r)
{
    switch (r)
    {
        case NORDEN: --y; break;
        case SUEDEN: ++y; break;
        case WESTEN: --x; break;
        case  OSTEN: ++x; break;
    }
}
```

- Aufzählungselemente sind in Wirklichkeit nur konstante Zahlen und nicht typsicher:

```
enum Farbe { rot, gruen, blau };
enum Aggregatzustand { fest, fluessig, gas, plasma };

int main(void)
{
    enum Farbe x = plasma;
    int i = x;
    enum Aggregatzustand z = -1;
    return 0;
}
```

- Die Werte werden automatisch vergeben, können aber auch manuell gesetzt werden:

```
enum Temperatur
{
    a,                      0
    absolutnull = -273,     -273
    b,                      -272
    gefrierpunkt = 0,       0
    c,                      1
    siedepunkt = 100,       100
    d,                      101
};
```

- Die Elemente eines Aufzählungstyps dürfen sich nicht in anderen Aufzählungstypen wiederholen:

```
enum Organ { Lunge, Herz, Leber, Niere };
enum Farbe { Karo, Herz, Pik, Kreuz };   /* redeclaration of enumerator 'Herz' */
```

```
enum Organ { Lunge, Herz, Leber, Niere };
enum Farbe { Karo, Pik = 2, Kreuz };
```

- struct fasst bestehende Datentypen zu einem neuen Datentyp zusammen:

```
struct person
{
    int alter;
    char name[16];
};

struct person hans = {25, "Hans"};
```

- Auch ein Strukturtyp kann zusammen mit Variablen desselben Typs deklariert werden:

```
struct person
{
    int alter;
    char name[16];
} hans = {25, "Hans"};
```

Initialisierung vs. Zuweisung

- Die = {...} Syntax funktioniert nur bei der Initialisierung, nicht bei der Zuweisung:

```
enum geschlecht { MAENNLICH, WEIBLICH };

struct person
{
    enum geschlecht geschlecht;
    int alter;
    char name[16];
};

    struct person hans = {MAENNLICH, 25, "Hans"};

    // hans kann nicht neu zugewiesen werden:
    hans = {WEIBLICH, 26, "Hansine"};

    // Stattdessen müssen die Attribute einzeln verändert werden:
    hans.geschlecht = WEIBLICH;
    hans.alter++;
    strcpy(hans.name, "Hansine");
```

“Methoden”

- “Methoden” sind Funktionen, die als ersten Parameter einen passenden Zeiger akzeptieren:

```
void person__feiere_geburtstag(struct person * self)
{
    (*self).alter++;

    self->alter++;
}

char person__geschlecht(const struct person * self)
{
    return "MW"[self->geschlecht];
}


    struct person hans = {MAENNLICH, 25, "Hans"};
    person__feiere_geburtstag(&hans);
```

Der “Methoden”-Präfix “person__” und der Parametername “self” sind persönliche Präferenzen, diese haben in der Sprache C keine besondere Bedeutung.