

# Grundlagen der Programmierung

## VL09: Programmieren in C

Prof. Dr. Samuel Kounev  
M.Sc. Jóakim v. Kistowski  
M.Sc. Norbert Schmitt

- Diese Vorlesung ist eine Übersicht über C und die Unterschiede zu Java
- Wir behandeln den C99 Standard. Moderne Compiler bieten oft Zusatzfunktionen, die einige der Einschränkungen von C99 umgehen.
- Weitere Materialien zum C Lernen:
  - Der “C++ Primer” (siehe Literaturliste in der 1. Vorlesung)
  - Allgemeines Tutorial:  
<http://www.cprogramming.com/tutorial/c-tutorial.html>
  - Tutorial für Java-Programmierer:  
<http://www.comp.lancs.ac.uk/~ss/java2c/ndiffs>

- C Basics
  - Mitgelieferte Datentypen
  - Arrays, Structs, Unions
- Pointer und Pointer-Arithmetik
- Funktionen
- Speicherverwaltung
- Deklarationen
- Compiling und Linking

# „Hello World“

## Java

### Klassenrumpf

In Java ist alles in Klassen organisiert

### Main-Methode

Rückgabtyp: **void**  
Parameter: **String[]**  
**static**-Methode in einer Klasse

```
/* HelloWorld.java */
```

```
public class HelloWorld {
```

```
    private static int zahl = 42;
```

```
    // Hauptprogramm
```

```
    public static void main(String[] args) {
        System.out.println("Hello " + zahl);
    }
}
```

### Attribut

An die Klasse gebundene Variable

### Ausgabe

Mit **System.out** und **String**-Konkatenation

### #include-Anweisung

Vergleichbar mit **Java-import**

```
/* helloworld.c */
```

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

```
int zahl = 5;
```

```
/* Hauptprogramm */
```

```
int main(void) {
    printf("Hello %d\n", zahl);
}
```

### Importierte header-Datei

Header behandeln wir später im C++-Kapitel

### Globale Variable

Von überall aus zugreifbar.  
Nicht an eine Klasse gebunden.

### Main-Funktion

Rückgabtyp: **int**  
Parameter: leer  
An keine Klasse gebunden!

### Ausgabe

Mit **printf** aus der importierten **stdio.h**.  
**String**-Konkatenation ist nicht trivial.  
Wird in der Übung besprochen.

## C

- Datentypen: `int`, `float`, `double`, `char`, `enum`
- Modifizierer, z.B.:
  - `signed`: mit Vorzeichen (z.B. `signed int`)
  - `unsigned`: ohne Vorzeichen

➔ *Tatsächliche Größen und Bereiche sind Plattform-abhängig!*

- **Arrays**: angezeigt mit `[ ]`
- **Boolean Werte**
  - C: `int = 0` für *false*; `int ≠ 0` für *true*
  - C++: `bool`
- **Strings**
  - C: `char[ ]`, endet mit `'\0'`
  - C++: `std::string` (unterstützt durch die Klasse `string` in der Standardbibliothek)

- Arrays fester Größe werden in C mit den Klammern nach dem Namen des Arrays definiert und initialisiert
  - Beispiel: int Array der Größe 5
  - `int foo[5];`
  - Müssen nicht extra initialisiert werden
  - Solche Arrays können nur lokal verwendet werden

## Java

Deklaration

Initialisierung

```
/* Java */
int max = 4;
int[] feld;
feld = new int[max];

for (int i = 0; i < max; i++) {
    feld[i] = i;
}
```

## C/C++

Deklaration und Initialisierung

**Laufvariable**

Kann in C99 und neuer auch innerhalb der **for**-Schleife definiert werden.  
In C++ auch möglich.

```
/* C */
int max = 4;
int feld[max];

int i;
for (i = 0; i < max; i++) {
    feld[i] = i;
}
```

- Nutzer-definierte Datentypen (derived types)
- Ähnlich wie Klassen in Java wenn sie zur Definition von neuen Datentypen verwendet werden
- Gruppieren Variablen miteinander → Mitglieder (members)
- Mitglieder werden über den Punktoperator (.) angesprochen
- `union` ist ähnlich wie `struct` aber nutzt gemeinsamen Speicherbereich für alle Mitglieder

```
struct myPoint {int dimX; int dimY; int dimZ;};

int main() {
    struct myPoint p;
    p.dimX = 10;
    p.dimY = 20;
    p.dimZ = 30;
    printf("%d\n", p.dimX);

    return 0;
}
```

- Nutzer-definierte Datentypen (derived types)
- Ähnlich wie Klassen in Java wenn sie zur Definition von neuen Datentypen verwendet werden
- Gruppieren Variablen miteinander → Mitglieder (members)
- Mitglieder werden über den Punktoperator (.) angesprochen
- `union` ist ähnlich wie `struct` aber nutzt gemeinsamen Speicherbereich für alle Mitglieder

```
struct myPoint {int dimX; int dimY; int dimZ;};  
  
int main() {  
    struct myPoint p;  
    p.dimX = 10;  
    p.dimY = 20;  
    p.dimZ = 30;  
    printf("%d %d %d", p.dimX, p.dimY, p.dimZ);  
    return 0;  
}
```

**ACHTUNG:** Keine Java-Klassen-Syntax  
bei der Deklaration der Variable.  
“**struct**” muss immer mitgeführt werden!



- Können Gruppen von Daten speichern
  - So wie Java-Klassen in VL 06
- Können **nicht**:
  - Methoden enthalten
  - Erben / vererbt werden (möglich für Klassen – kommt später)
  - Alles andere, was in den folgenden Vorlesungen zu Objektorientierung kommt

```
/* Java */  
public class Date {  
    public int day, month, year;  
}  
public class Main {  
    public static void main(String[] args) {  
        Date xmas;  
        xmas = new Date();  
        xmas.day = 24;  
        xmas.month = 12;  
        xmas.year = 2014;  
    }  
}
```

Deklaration

Initialisierung

```
/* C */  
struct date {int day; int month; int year;};  
  
int main(void) {  
    struct date xmas;  
    xmas.day = 24;  
    xmas.month = 12;  
    xmas.year = 2014;  
}
```

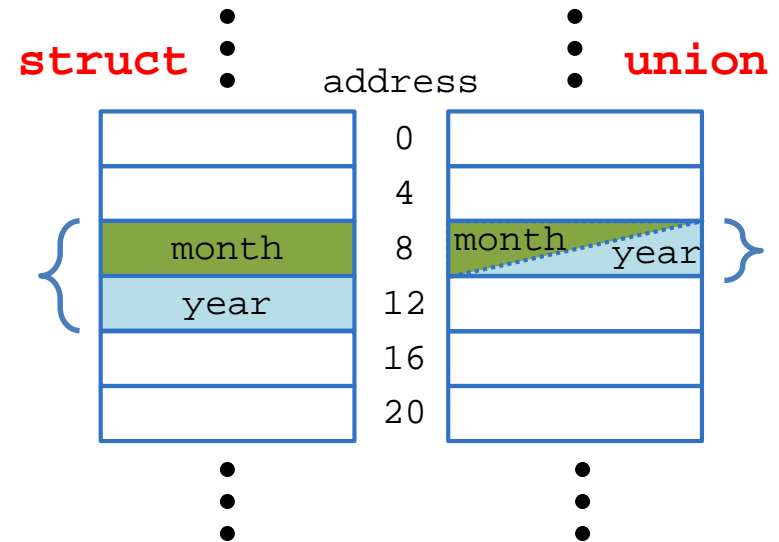
Deklaration + Initialisierung

- Ein Datum mit Monat **UND** Jahr

```
struct date {  
    int month;  
    int year;  
};
```

- Ein Datum mit Monat **ODER** Jahr

```
union date {  
    int month;  
    int year;  
};
```



```
struct myPoint {int dimX; int dimY;};

int main() {
    struct myPoint p;
    p.dimX = 10;
    p.dimY = 20;
    printf("%d\n", p.dimX);    /* output? */

    return 0;
}
```

```
union myPoint {int dimX; int dimY;};

int main() {
    union myPoint p;
    p.dimX = 10;
    p.dimY = 20;
    printf("%d\n", p.dimX);    /* output? */

    return 0;
}
```

```
struct my3DPoint {int dimX; int dimY; int dimZ;};

int main() {
    struct my3DPoint p;
    p.dimX = 10;
    p.dimY = 20;
    p.dimZ = 30;
    printf("%d\n", p.dimX);    /* output is 10 */
    return 0;
}
```

```
union my3DPoint {int dimX; int dimY; int dimZ;};

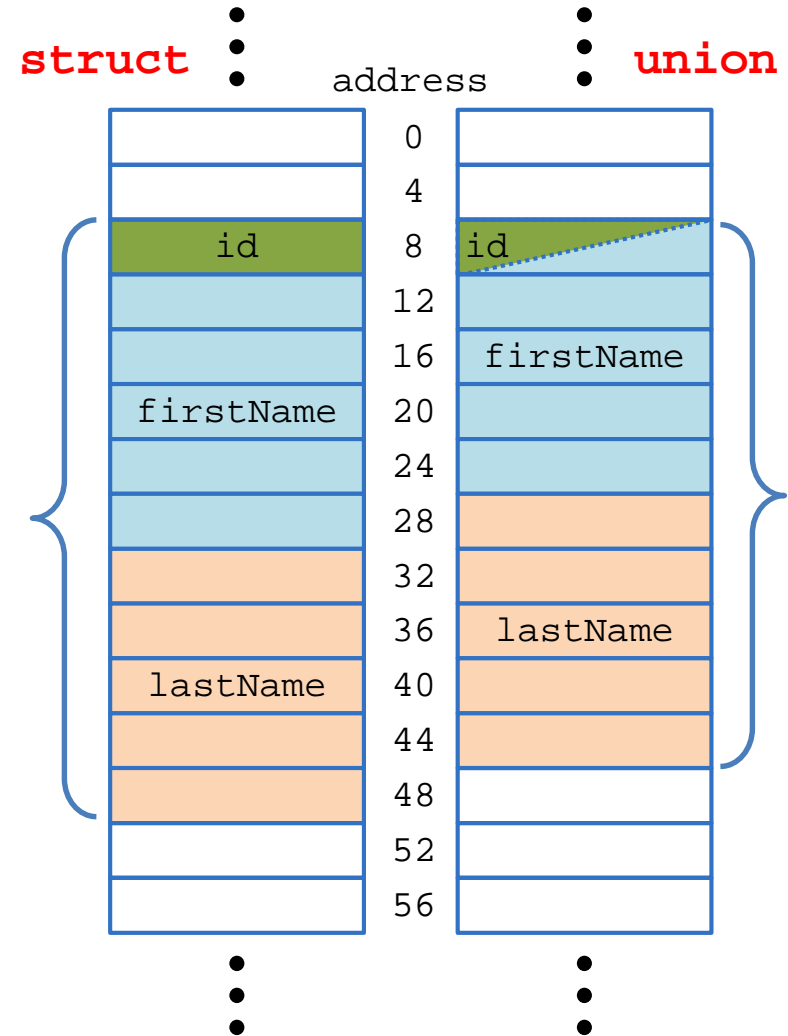
int main() {
    union my3DPoint p;
    p.dimX = 10;
    p.dimY = 20;
    p.dimZ = 30;
    printf("%d\n", p.dimX);    /* output is 30 */
    return 0;
}
```

- Ein Student mit **id UND** name

```
struct student {
    int id;
    struct {
        char firstName[20];
        char lastName[20];
    } name;
};
```

- Ein Student mit **id ODER** name

```
union student {
    int id;
    struct {
        char firstName[20];
        char lastName[20];
    } name;
};
```

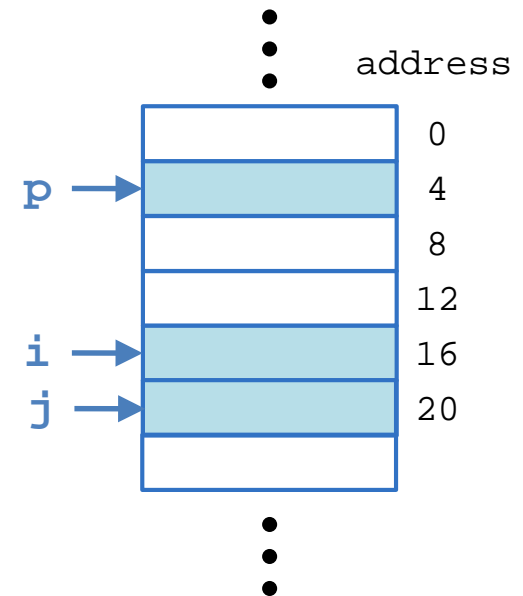


- C Basics
  - Mitgelieferte Datentypen
  - Arrays, Structs, Unions
- Pointer und Pointer-Arithmetik
- Funktionen
- Speicherverwaltung
- Deklarationen
- Compiling und Linking

- C-Variable, die eine *Adresse* einer anderen Variable enthält
- Ähnlich zu Referenzdatentypen in Java, aber viel flexibler
- Pointer-Syntax:
  - Pointer Deklaration: Sternchen (\*)
  - Auflösung der Adresse einer Variable: Referenzoperator (&)
  - Auflösung des Zielwertes eines Pointers: Dereferenzoperator (\*)

```
int *p;           /* Pointer auf int */
int i, j;

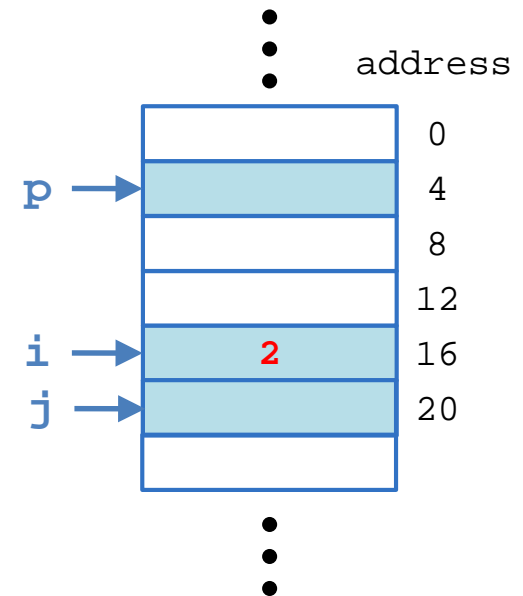
int main() {
    i = 2;
    p = &i;       /* p zeigt auf die Adresse von i */
    j = *p;       /* j wird der Wert von i zugewiesen */
    return 0;
}
```



- C-Variable, die eine *Adresse* einer anderen Variable enthält
- Ähnlich zu Referenzdatentypen in Java, aber viel flexibler
- Pointer-Syntax:
  - Pointer Deklaration: Sternchen (\*)
  - Auflösung der Adresse einer Variable: Referenzoperator (&)
  - Auflösung des Zielwertes eines Pointers: Dereferenzoperator (\*)

```
int *p;           /* Pointer auf int */
int i, j;

int main() {
    i = 2;
    p = &i;       /* p zeigt auf die Adresse von i */
    j = *p;       /* j wird der Wert von i zugewiesen */
    return 0;
}
```

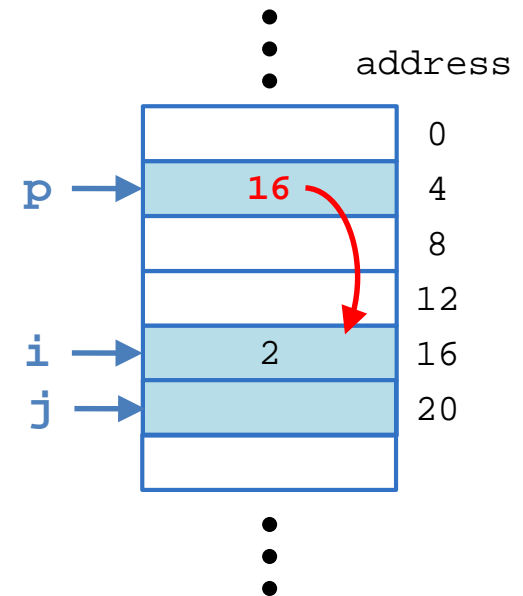




- C-Variable, die eine *Adresse* einer anderen Variable enthält
- Ähnlich zu Referenzdatentypen in Java, aber viel flexibler
- Pointer-Syntax:
  - Pointer Deklaration: Sternchen (\*)
  - Auflösung der Adresse einer Variable: Referenzoperator (&)
  - Auflösung des Zielwertes eines Pointers: Dereferenzoperator (\*)

```
int *p;           /* Pointer auf int */
int i, j;

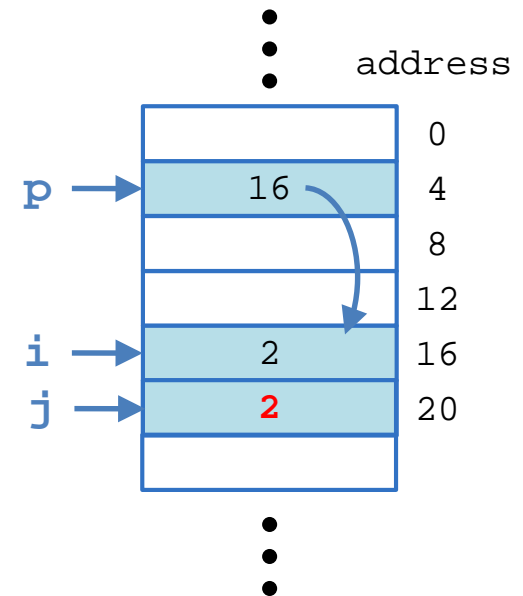
int main() {
    i = 2;
    p = &i;       /* p zeigt auf die Adresse von i */
    j = *p;       /* j wird der Wert von i zugewiesen */
    return 0;
}
```



- C-Variable, die eine *Adresse* einer anderen Variable enthält
- Ähnlich zu Referenzdatentypen in Java, aber viel flexibler
- Pointer-Syntax:
  - Pointer Deklaration: Sternchen (\*)
  - Auflösung der Adresse einer Variable: Referenzoperator (&)
  - Auflösung des Zielwertes eines Pointers: Dereferenzoperator (\*)

```
int *p;           /* Pointer auf int */
int i, j;

int main() {
    i = 2;
    p = &i;       /* p zeigt auf die Adresse von i */
    j = *p;       /* j wird der Wert von i zugewiesen */
    return 0;
}
```



- Werden gerne benutzt um Parameter an Funktionen zu übergeben
  - Vermeidet das Kopieren von Datenstrukturen, trotz „*call by value*“
  - Erlaubt Bearbeiten von Daten in einer Funktion, ohne Datenverlust nachdem die Funktion verlassen wird
- Zeigen auf...
  - ...jeden Datentyp, inklusive structs, Klassen (C++), und `void`
  - ...*Funktionen*
  - ...andere Pointer
- Nützlich, um mit Arrays zu arbeiten
- Können zur Erstellung und Bearbeitung von Datenstrukturen, wie verketteten Listen verwendet werden

- Pointer können *inkrementiert* und *dekrementiert* werden
  - mit der (Plattform-spezifischen) Größe ihrer Datentypen
  - dadurch kann man sich im Hauptspeicher nach vorne und nach hinten bewegen
  - man bewegt sich in Schritten entsprechend der Größe des Datentyps des Pointers
- *Achtung: Mögliche Datenfehler auf Grund von direktem Speicherzugriff!*

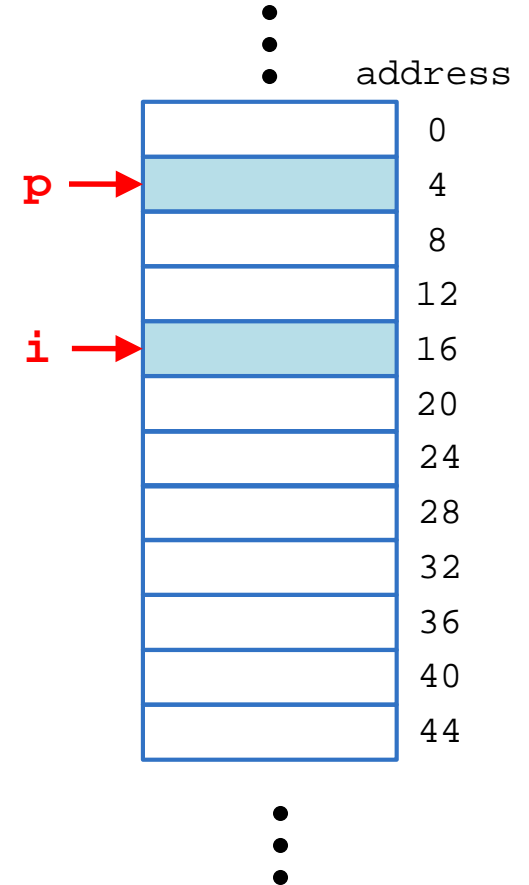
```
char c1;  
char *p;    /* pointer auf char */  
  
int main() {  
    c1 = 'A';  
    p = &c1;    /* p wird die Adresse von Variable c1 zugewiesen */  
  
    p = p + 5;    /* p wird um 5 * sizeof(char) inkrementiert, d.h. 5 chars nach vorne */  
    p = p - 5;    /* p wird um 5 * sizeof(char) dekrementiert, d.h. 5 chars nach hinten */  
    p++;          /* p wird um 1 * sizeof(char) inkrementiert, d.h. 1 char nach vorne */  
    p--;          /* p wird um 1 * sizeof(char) dekrementiert, d.h. 1 char nach hinten */  
    p += 10;      /* p wird um 10 * sizeof(char) inkrementiert, d.h. 10 chars nach vorne */  
    p -= 10;      /* p wird um 10 * sizeof(char) dekrementiert, d.h. 10 chars nach hinten */  
  
    return 0;  
}
```

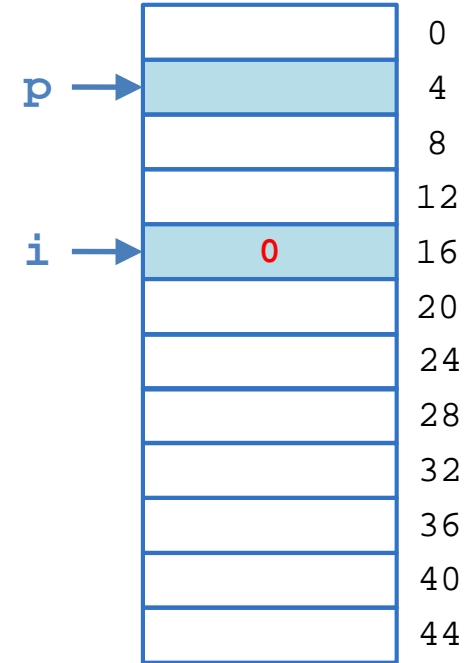
```

int i;
int *p;      /* Pointer auf int */

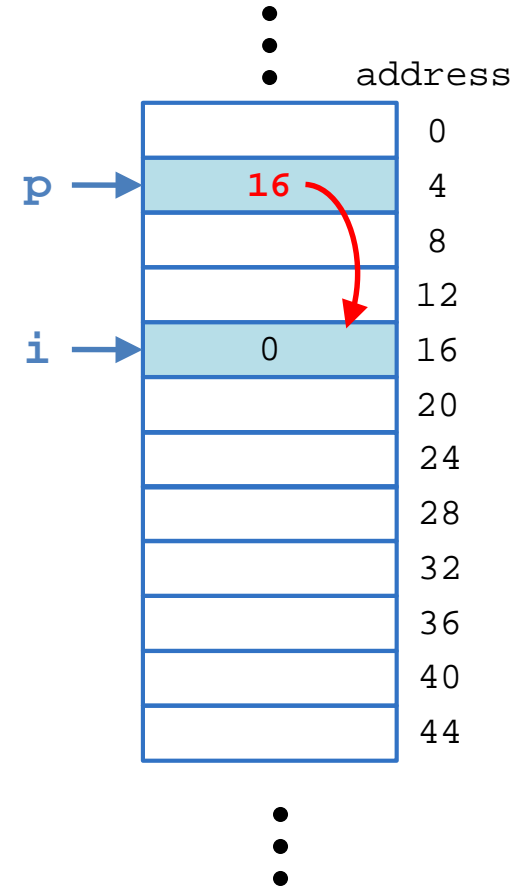
int main() {
    i = 0;
    p = &i;
    p = p + 2;
    *p = 145;
    *(p+3) = 555;
    p[3] = 1000;
    return 0;
}

```

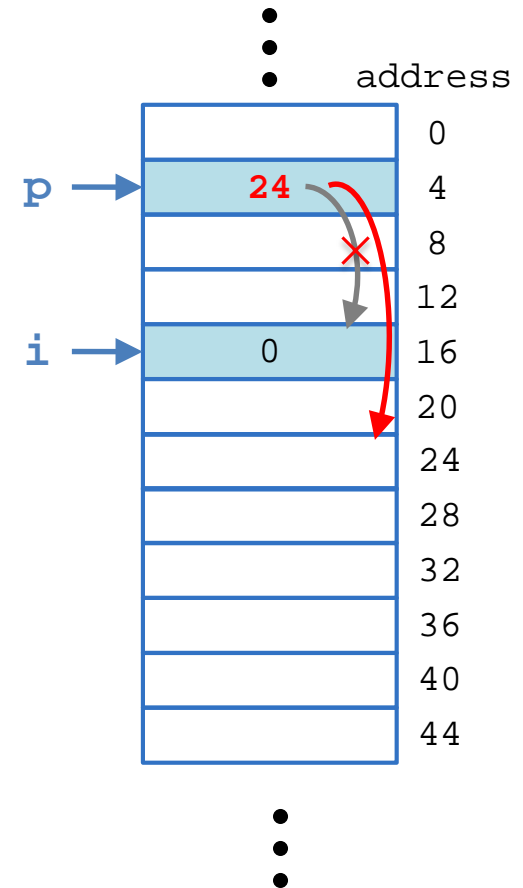




```
int i;  
int *p;      /* Pointer auf int */  
  
int main() {  
    i = 0;  
    p = &i;      /* p zeigt auf die Adresse von i */  
    p = p + 2;  
    *p = 145;  
    *(p+3) = 555;  
    p[3] = 1000;  
    return 0;  
}
```



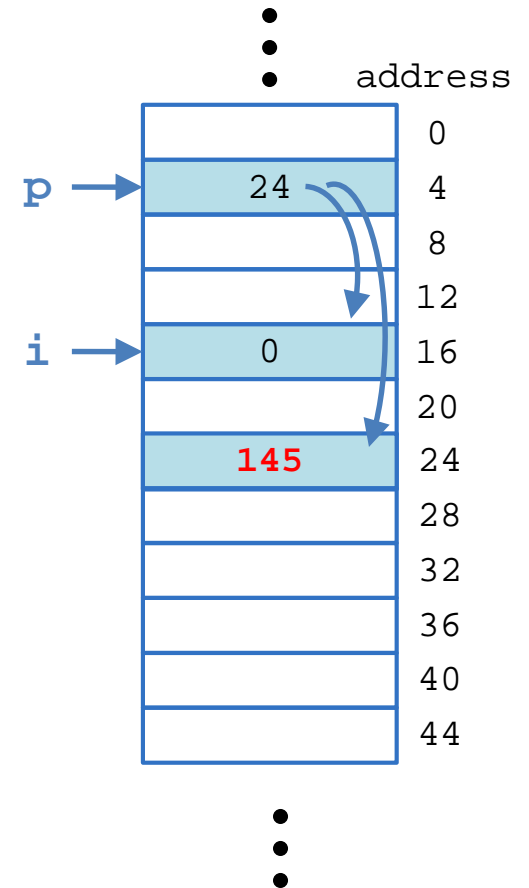
```
int i;  
int *p;      /* Pointer auf int */  
  
int main() {  
    i = 0;  
    p = &i;   /* p zeigt auf die Adresse von i */  
    p = p + 2; /* p wird um 2 * sizeof(int) inkrementiert */  
    *p = 145;  
    *(p+3) = 555;  
    p[3] = 1000;  
    return 0;  
}
```



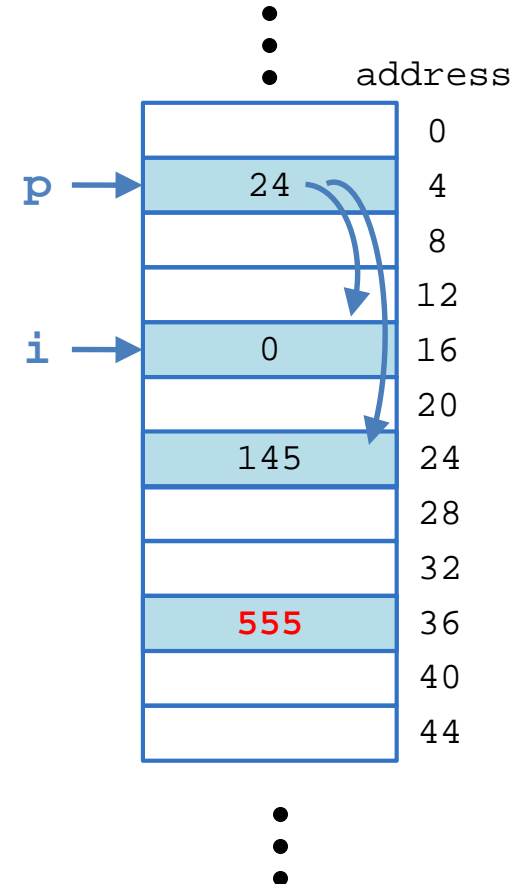
**Anmerkung:** Ein **int** ist normalerweise 4 bytes (d.h. `sizeof(int) == 4`) →  
Zwischen Adresse **p** und Adresse **(p+2)** können genau 2 **int** Werte gespeichert werden!



```
int i;  
int *p;      /* Pointer auf int */  
  
int main() {  
    i = 0;  
    p = &i;   /* p zeigt auf die Adresse von i */  
    p = p + 2; /* p wird um 2 * sizeof(int) inkrementiert */  
    *p = 145;  
    *(p+3) = 555;  
    p[3] = 1000;  
    return 0;  
}
```



```
int i;  
int *p;      /* Pointer auf int */  
  
int main() {  
    i = 0;  
    p = &i;   /* p zeigt auf die Adresse von i */  
    p = p + 2; /* p wird um 2 * sizeof(int) inkrementiert */  
    *p = 145;  
    → *(p+3) = 555;  
    p[3] = 1000;  
    return 0;  
}
```



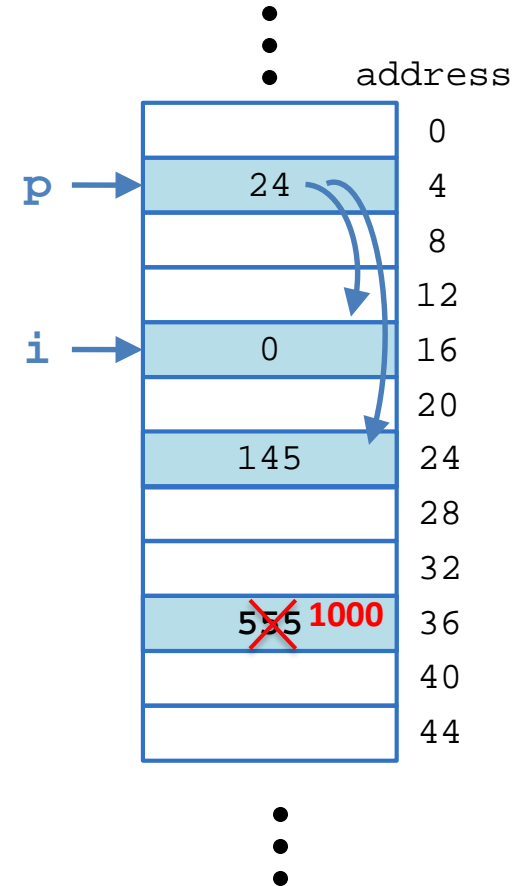
**Anmerkung:** Die Benutzung des Dereferenzoperators \* mit **(p+3)** bewirkt keine Änderung von **p**, sondern von dem **int-Wert** der auf Adresse **p+3\*sizeof(int)** liegt!

```

int i;
int *p;      /* Pointer auf int */

int main() {
    i = 0;
    p = &i;   /* p zeigt auf die Adresse von i */
    p = p + 2; /* p wird um 2 * sizeof(int) inkrementiert */
    *p = 145;
    *(p+3) = 555;
    p[3] = 1000;
    return 0;
}

```



**Anmerkung:** **p[3]** ist das gleiche wie **\*(p+3)**  
d.h. es entspricht der Adresse **p+3\*sizeof(int)**

- Man kann auf Pointer *zugreifen*, als wären sie Arrays

- D.h. `int array[N];` mit `int *p = &array[0];`

gilt: `p[M] == *(p+M) == array[M]` ( $M < N$ )

```
char c1, c2;
char *p;      /* pointer auf char */

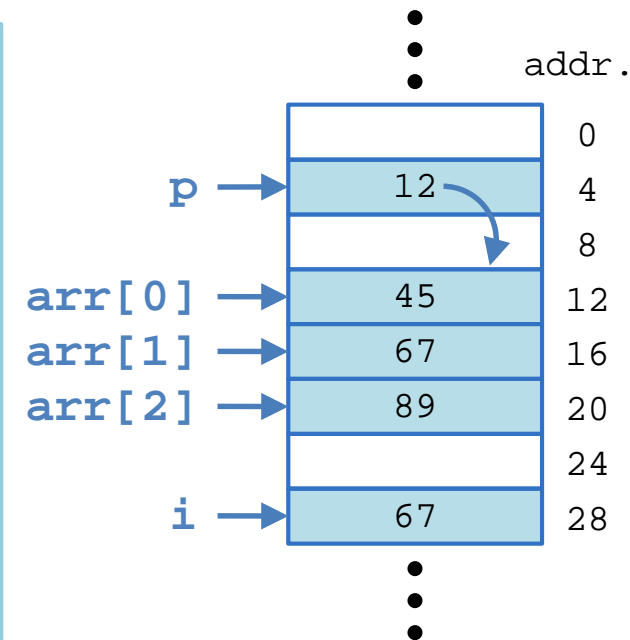
int main() {
    c1 = 'A';
    p = &c1;   /* p wird die Adresse von c1 zugewiesen */
    p += 5;    /* p wird um 5 * sizeof(char) inkrementiert */
    c2 = p[2]; /* c2 wird der Wert zugewiesen, der an der Adresse p + 2 * sizeof(char) steht */
    return 0;
}
```

- Arrays werden als zusammenhängende Speicherbereiche, in denen die Array-Elemente aufeinander folgen, benutzt
- Mit Pointer-Arithmetik können Pointer auf Adressen beliebiger Array-Elemente zeigen
- Arraynamen können auf die Adresse des ersten Array-Elements *heruntergebrochen* werden

```
int arr[] = { 45, 67, 89 };
int *p, i;

int main() {
    p = arr;           /* p is assigned the address of
                        the first element of arr */
    p = &arr[0];       /* same effect as above */

    i = arr[1];        /* i is assigned the value 67 */
    i = p[1];          /* same effect as above */
    i = *(p + 1);      /* same effect as above */
    return 0;
}
```

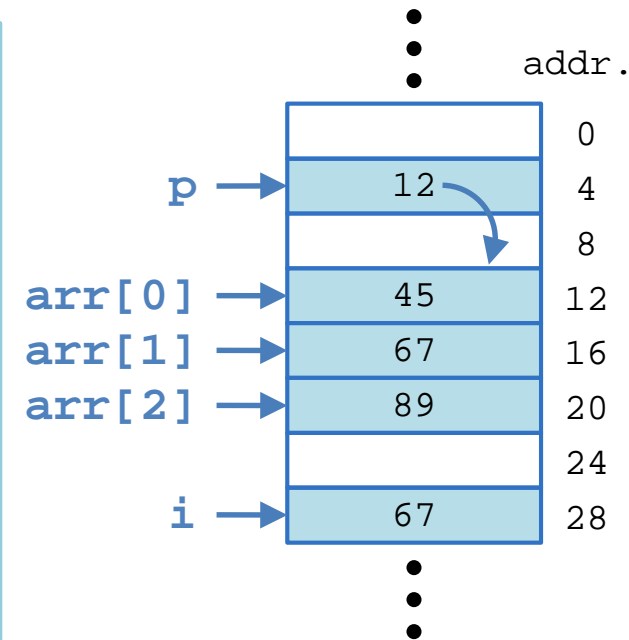


- Frage: Ist `p` äquivalent zu `arr`?

```
int arr[] = { 45, 67, 89 };
int *p, i;

int main() {
    p = arr;           /* p is assigned the address of
                        the first element of arr */
    p = &arr[0];       /* same effect as above */

    i = arr[1];        /* i is assigned the value 67 */
    i = p[1];          /* same effect as above */
    i = *(p + 1);      /* same effect as above */
    return 0;
}
```

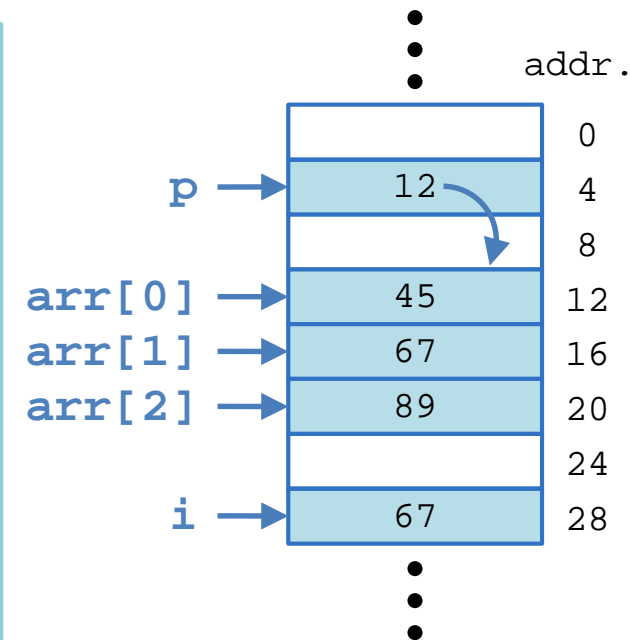


- Achtung: obwohl wir `p = arr` schreiben, ist `p` nicht das gleiche wie `arr`:
  - `arr` (oder `&arr[0]`) **ist** die Adresse des ersten Array-Elements
  - `p` **zeigt auf** die Adresse des ersten Array-Elements

```
int arr[] = { 45, 67, 89 };
int *p, i;

int main() {
    p = arr;           /* p is assigned the address of
                        the first element of arr */
    p = &arr[0];       /* same effect as above */

    i = arr[1];        /* i is assigned the value 67 */
    i = p[1];          /* same effect as above */
    i = *(p + 1);      /* same effect as above */
    return 0;
}
```



# C Pointer vs. Java Referenzdatentypen

- C Pointer sind ähnlich wie Referenzdatentypen in Java
- **Unterschiede**
  - Pointer sind für beliebige Variablen nutzbar (auch einfache Datentypen)
  - Unterstützen Pointer-Arithmetik
  - Explizite Nutzung des Referenzoperators/Dereferenzoperators
- Beispiel: Array

```
/* C */
```

```
int arr[] = { 1, 2, 3 };
```

```
int *p = arr;      /* C Pointer */
```

```
p[0] = 12;
```

```
/* Java */
```

```
int[] arr = { 1, 2, 3 };
```

```
int[] p = arr;     /* Java reference variable */
```

```
p[0] = 12;
```



# C Pointer vs. Java Referenzdatentypen (2)

- C Pointer sind ähnlich wie Referenzdatentypen in Java
- **Unterschiede**
  - Pointer sind für beliebige Variablen nutzbar (auch einfache Datentypen)
  - Unterstützen Pointer-Arithmetik
  - Explizite Nutzung des Referenzoperators/Dereferenzoperators
- Beispiel: Komplexer Datentyp

```
/* C */  
struct date {int day; int month; int year;};  
  
struct date *aDate;  /* C Pointer */  
  
struct date dateInstance;  
aDate = &dateInstance;  
  
(*aDate).day = 12;
```

```
/* Java */  
public class Date {int day; int month; int year;}  
  
Date aDate;  /* Java reference variable */  
  
aDate = new Date();  
  
aDate.day = 12;
```

# Default Werte

- In C werden keine Standardwerte gesetzt. Variablen werden im Speicher allokiert und erhalten den Wert, der bereits an dieser Speicherstelle eingetragen ist
  - Dies gilt auch für Pointer-Adressen

```
int main() {  
    int *p;      /* p zeigt irgendwo hin */  
    *p = 5;      /* schreibt 5 an die in p gespeicherte Adresse */  
    int i;       /* enthält den Wert, der an i's Adresse stand */  
    return 0;  
}
```

- C Basics
  - Mitgelieferte Datentypen
  - Arrays, Structs, Unions
- Pointer und Pointer-Arithmetik
- Funktionen
- Speicherverwaltung
- Deklarationen
- Compiling und Linking

- Funktionen sind wie Methoden in Java, sie haben
  - Signaturen
  - Einen Rückgabebetyp
  - Übergabeparameter
  
- Aber:
  - Sie existieren außerhalb von Klassen
    - C++: können auch in Klassen existieren
  - Rückgabewert und Parameter sind immer Pass-by-Value
    - C++: auch Pass-by-Reference möglich (siehe C++ Kapitel)

- **Pass-by-Value:** Alle Parameter werden bei der Übergabe kopiert
- Auch Datentypen, die in Java als “Referenzdatentypen” gelten!

**C:**

Nimmt eine Kopie des Datums an. Änderungen werden nicht zurückgeschrieben

**C:**

Das ausgegebene Datum ist nicht 24, da das Ergebnis nicht zurückgeschrieben wurde

```
/* falsch */
#include <stdio.h>

struct date {int day; int month; int year;};

void setToXMas(struct date aDate) {
    aDate.day = 24;
    aDate.month = 12;
    aDate.year = 2014;
}

int main() {
    struct date xmas;
    setToXMas(xmas);
    printf("Day: %d\n",xmas.day);
    return 0;
}
```

**Java:**

Es wird eine Referenz übergeben. Änderungen werden zum Aufrufer propagiert.

**Java:**

In Java wäre xmas.day == 24

- **Lösung:** Pointer übergeben
  - Pointer werden immer noch kopiert
  - Die Daten werden jetzt aber überschrieben

## C: Pointer

Nimmt eine Kopie des Pointers an. Änderungen werden auf die Adresse des Originaldatums geschrieben

## C: Pointer

Das ausgegebene Datum ist 24, da das Originaldatum überschrieben wurde

```
/* richtig */
#include <stdio.h>

struct date {int day; int month; int year;};

void setToXMas(struct date *aDate) {
    (*aDate).day = 24;
    (*aDate).month = 12;
    (*aDate).year = 2014;
}

int main() {
    struct date xmas;
    setToXMas(&xmas);
    printf("Day: %d\n",xmas.day);
    return 0;
}
```

## Java:

Intern macht Java genau das gleiche, wie C mit Pointern. "Echtes" **Pass-by-Reference** gibt es nur bei C++

## ■ Sonderfall: Array-Parameter

- Syntax suggeriert Kopie des ganzen Arrays
- Es wird aber nur der Pointer auf die erste Adresse übergeben

```
void incr(int arr[], int length) { /* as a function parameter, int arr[]  
                                   implicitly means int *arr */  
    int i = 0;  
    for(i = 0; i < length; i++) {  
        arr[i]++; /* array elements can be accessed by  
                  standard pointer subscription */  
    }  
}  
  
int main() {  
    int arr[] = { 45, 67, 89 };  
    incr(arr, 3); /* arr is the same as &arr[0] */  
    return 0;  
}
```

- C erlaubt **Pointer auf Funktionen**
  - Diese erlauben es, Funktionen in Variablen zu speichern
- Deklaration ist ähnlich, wie in anderen Sprachen
  - Deklariert eine Variable „foo“, welche auf eine Funktion zeigt
    - Erwartet ein **int** und hat als Rückgabetyt **void** `void (*foo) (int);`
- Die **Adresse** einer Funktion kann so aufgelöst werden:

```
void my_int_func(int x) {  
    printf( "%d\n", x );  
}
```

...

```
void (*foo)(int);  
foo = &my_int_func;  
foo(2);
```

*fetch address of my\_int\_function*

cf. e.g. <http://www.cprogramming.com/tutorial/function-pointers.html>



- C Basics
  - Mitgelieferte Datentypen
  - Arrays, Structs, Unions
- Pointer und Pointer-Arithmetik
- Funktionen
- Speicherverwaltung
- Deklarationen
- Compiling und Linking

- Jedes Programm wird in drei Speicherbereiche (Segmente) aufgeteilt:
  - *text segment* (oder *code segment*): Programminstruktionen und Konstanten
  - *stack segment*: Lokale Variablen und Parameter bei Funktionen
  - *heap segment*: Globale Variablen und dynamisch allozierter Speicher (im Code)
- Direkte Speichermanipulation mit Pointern ist möglich → **Mögliche Speicherfehler!**
- Explizite **dynamische Allokation** von Speicher mit
  - `void * malloc(int size)`: Allokiert `size` zusammenhängende Bytes. Gibt einen Pointer auf die Adresse des ersten Bytes zurück.
  - Größe von erforderlichem Speicher kann mit `sizeof(<Datentyp>)` ermittelt werden
  - `void free(void * ptr)`: Deallokiert den Speicherblock, der bei `ptr` beginnt.
  - Aufräumen von ungenutztem Speicher nicht vergessen!
    - **Keine garbage collection (wie in Java)!**

## ■ Stack

- Umgesetzt als *Last In First Out* (LIFO)
- Automatische Allokation / Speicherfreigabe bei Betreten / Verlassen von Funktionen und Blöcken
- **Achtung:** Rekursive Funktionsaufrufe / „große“ Funktionsparameter können einen Stack-Überlauf (overflow) verursachen!

## ■ Heap

- Automatische Allokation von globalen Variablen (sowie statischen Variablen → kommt später) ab ihrer Definition bis zum Ende der Programmausführung
- Explizite dynamische Allokation / Freigabe von beliebig großen Speicherblöcken auf Anfrage
- **Achtung:** Sequentielle Speicheranfragen resultieren nicht zwangsläufig in aufeinanderfolgenden Speicherblöcken

```
#include <stdlib.h>

int i;                                /* Globale Variable – Sichtbar bis zum Ende dieser Datei. Auf dem Heap allokiert. */

void incr(int *ptr) {
    int delta = 2;                    /* Lokale Variable – In incr sichtbar. Auf dem Stack allokiert. */
    *ptr += delta;
}

int main() {
    double d = 3.7;                  /* Lokale Variable – In main sichtbar. Auf dem Stack allokiert. */
    int *ptr;                        /* Lokale Pointer-Variable – s.o. */
    ptr = (int *) malloc(sizeof(int)); /* Dynamische Speicher-Allokation auf dem Heap. */
    free(ptr);                       /* Freigabe des allokierten Speichers auf dem Heap. */
    return 0;
}
```

- Direkte Speicher-Allokation in der Deklaration, als lokale Variable
  - Beispiel: Integer Array der Größe 5
  - `int foo[5];`
- Oder: Pointer, der auf einen freien Speicherbereich zeigt
  - `int * foo;`  
`foo = (int *) malloc(5 * sizeof(int));`
  - Achtung: Mit `malloc()` allozierter Speicher bleibt allokiert, bis er mit `free()` wieder freigegeben wird.

```
#include <stdlib.h>

int main() {
    int * array = createArray();
    int someInt = array[3]; /* was steht in array[3]? */
    return 0;
}

/* falsch, array wird nur lokal deklariert.
                                   Gültig nur bis Funktions-Ende. */
int * createArray() {
    int array[5];
    array[3] = 42;
    return array;
}

/* richtig */
int * createArray() {
    int * array = (int *) malloc(5*sizeof(int));
    array[3] = 42;
    return array;
}
```

- Es gibt in C keine Strings
- Stattdessen werden char-Arrays benutzt
  - Haben maximal-Länge (Array Länge)
  - Das letzte Zeichen ist immer das 0-Byte (\0)
  - `char text[6] = „Hello\0“;`
- `<string.h>` enthält wichtige Funktionen zur Bearbeitung von char-Arrays:
  - Konkatenation: `char *strcat(char *s1, const char *s2);`
  - Kopieren: `char *strcpy(char *dest, const char *src);`
  - ...

# Beispiel: Strings in C

```
public class Achtung {  
    public static void main(String[] args){  
        System.out.println("Ach" + "tung");  
    }  
}
```

**Java**

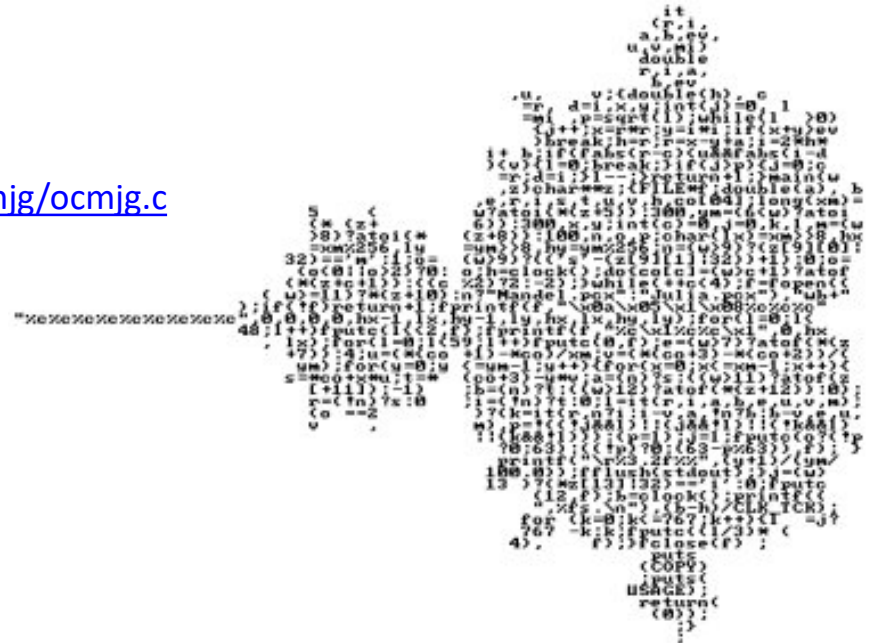
```
#include <stdio.h>  
#include <string.h>  
  
int main()  
{  
    char ach[8] = "Ach";  
    char tung[5] = "tung";  
    strcat(ach, tung);  
    printf("%s\n", ach); /* Ausgabe auf die Konsole*/  
    return 0;  
}
```

**C/C++**



- Kannst du es gut?

<http://www.funiter.org/ocmijg/ocmijg.c>



```
long long n,u,m,b;main(e,r)char **r;{f\ or(;n++ || (e=getchar())|32)>=0;b="ynwtsflrabg"[n%=11]-e?b:b*8+ n)
for(r=b%64-25;e<47&&b/b/=8)for(n=19;n[n["1+DIY/.K430x9\ G(kC["]-42&255^b| |(m+=n>15?n:n>9?m%u*~-u:
~(int)r?n+ !(int)r*16:n*16,b=0))u=1ll<<6177%n--*4;printf("%llx\n",m);}
```

<http://www.ioccc.org/2012/kang/kang.c>

- C Basics
  - Mitgelieferte Datentypen
  - Arrays, Structs, Unions
- Pointer und Pointer-Arithmetik
- Funktionen
- Speicherverwaltung
- Deklarationen
- Compiling und Linking

- Prinzip der *Vorwärtsdeklaration*
  - Alle Entitäten (**Funktionen**, Variablen, Typen) müssen vor ihrer Nutzung deklariert werden
- Der selbe Bezeichner darf mehrfach deklariert werden
  - z.B., in verschiedenen Dateien
- Die tatsächliche Definition des Bezeichners geschieht an genau einem Ort
  - und kann mit der Deklaration integriert werden

- Prinzip der **Vorwärtsdeklaration**
  - Problem: Zyklische Aufrufe sind unmöglich. Methodenordnung ist schwierig.
  - Beispiel: Inkrementiert und verdoppelt Wert *n*-Mal. Enthält zyklische Abhängigkeit.

```
int doubleValue(int a, int i) {  
    return increment(2*a, i);  
}  
int increment(int a, int i) {  
    if (i == 0) { return a; }  
    return doubleValue(a+1, i-1);  
}  
int main() {  
    int i = 5;  
    return increment(1, i);  
}
```

- Compilerausgabe:

```
main.cpp: In function 'int doubleValue(int, int)':  
main.cpp:2:27: error: 'increment' was not declared in this scope  
    return increment(2*a,i);  
                   ^
```

- Lösung: Separate Vorwärtsdeklaration

## Vorwärtsdeklaration

Deklariert `increment(int,int)` vor der Implementierung.

Deklarationen werden normalerweise in separaten **header-Dateien** (.h) abgelegt die mit **#include** eingebunden werden (zum Beispiel **#include <string.h>** bindet die `string.h` header-Datei aus der C Standard Library ein).

```
int increment(int, int);

int doubleValue(int a, int i) {
    return increment(2*a, i);
}

int increment(int a, int i) {
    if (i == 0) { return a; }
    return doubleValue(a+1, i-1);
}

int main() {
    int n = 5;
    return increment(1, i);
}
```

## Referenzierung von deklarierter Funktion

`doubleValue(int,int)` kann `increment(int,int)` referenzieren, da es bereits deklariert wurde.

## Auflösung der separaten Implementierung

Als Teil des Kompiliervorgangs ordnet der Linker die Implementierung von `increment(int,int)` der Deklaration zu.

**ACHTUNG: Das geht auch Datei-übergreifend!**

- Typ-Spezifizierer: `void`, `char`, `short`, `int`, `long`, `signed`, `unsigned`, `float`, `double`
- `const`
  - read-only nach Definition, ähnlich wie *final* in Java
  - Achtung: Mit Pointern und direktem Speicherzugriff können Speicherdaten auch mit Hilfe einer read-only Pointervariable überschrieben werden!

```
int i;                /* i: normal integer variable */  
  
const double pi = 3.14159; /* pi: read-only double */  
  
int * const j = &i;    /* j: read-only pointer to int */
```

- Andere Modifizierer: `extern`, `volatile`, `static`, ...

- Mit `extern` werden globale Variablen deklariert, die in einer anderen Source-Code Datei definiert sind
- Beispiel:

file1.c

```
int x;  
...  
int func1() {  
    x = 3;  
}  
...
```

file2.c

```
extern int x;  
...  
int func2() {  
    x = 5;  
}  
...
```

In beiden Dateien bezieht sich `x` auf dieselbe globale Variable

- `volatile`
  - wird immer vom Speicher gelesen → kein Caching, keine Optimierung
  - Sinnvoll, wenn auf die Variable von außerhalb des Programmablaufs zugegriffen wird (z.B., I/O-Buffer)
- `static`
  - Allokiert Variable in das heap-Speichersegment

```
void my_func(int x) {  
    volatile int i;  
    static int j;      /* j wird in das Heap allokiert */  
}
```

- `j` ist trotzdem nur in `my_func()` sichtbar
- Bleibt allerdings nach Beenden der Funktion bestehen und kann beim nächsten Aufruf der Funktion benutzt werden



- C Deklarationen können schwer zu lesen sein:

- Nicht einfach von links nach rechts

```
int * arr[]; /* arr is an array of  
pointers to int */
```

- Möglicherweise verschachtelt

```
int *(*p)(); /* p is a pointer to a  
function returning a  
pointer to an int */
```

- Modifizierer wie const und volatile

```
volatile int * const i; /* i is a  
read-only pointer to  
a volatile int */
```

- What does the following declaration mean?

```
static unsigned int * const *(*next)();
```

- A      Bezeichner “name” (von links nach rechts)      „[name] ist ein...”
- B      Präzedenzreihenfolge:
- B.1      Klammer ( )      „...” gruppiert Teile einer Deklaration
- B.2      Postfixoperatoren:
- B.2.1 ( )      „...Funktion mit Rückgabewert...”
- B.2.2 [ ]      „...Array von...”
- B.3      Prefixoperator: \*      „...Pointer auf...”
- B.4      Prefixoperator \* und const / volatile Modifizierer:      „...[modifizierter] Pointer auf...”
- B.5      const / volatile Modifizierer neben Typ-Spezifikator:      „...[modifizierter] [Spezifikator]”
- B.6      Typ-Spezifikator:      „...[Spezifikator]”

```
static unsigned int * const *(*next)();
```


```
static unsigned int * const * (*next) ();
```

1.	A	<b>next</b>	„next is ein ...“

```
static unsigned int * const * (*next)();
```

1.	A	next	„next is ein ...“
2.	B.3	*	„...Pointer auf...“

```
static unsigned int * const * (*next)();
```

1.	A	next	„next is ein ...“
2.	B.3	*	„...Pointer auf...“
3.	B.1	( )	„...“ (Gruppierung der Deklaration)

```
static unsigned int * const * (*next)();
```

1.	A	next	„next is ein ...“
2.	B.3	*	„...Pointer auf...“
3.	B.1	( )	„...“ (Gruppierung der Deklaration)
4.	B.2.1	( )	„...eine Funktion mit Rückgabewert...“

```
static unsigned int * const * (*next)();
```

1.	A	<code>next</code>	„next is ein ...“
2.	B.3	<code>*</code>	„...Pointer auf...“
3.	B.1	<code>()</code>	„...“ (Gruppierung der Deklaration)
4.	B.2.1	<code>()</code>	„...eine Funktion mit Rückgabewert...“
5.	B.3	<b><code>*</code></b>	„...Pointer auf...“



```
static unsigned int * const *(*next)();
```

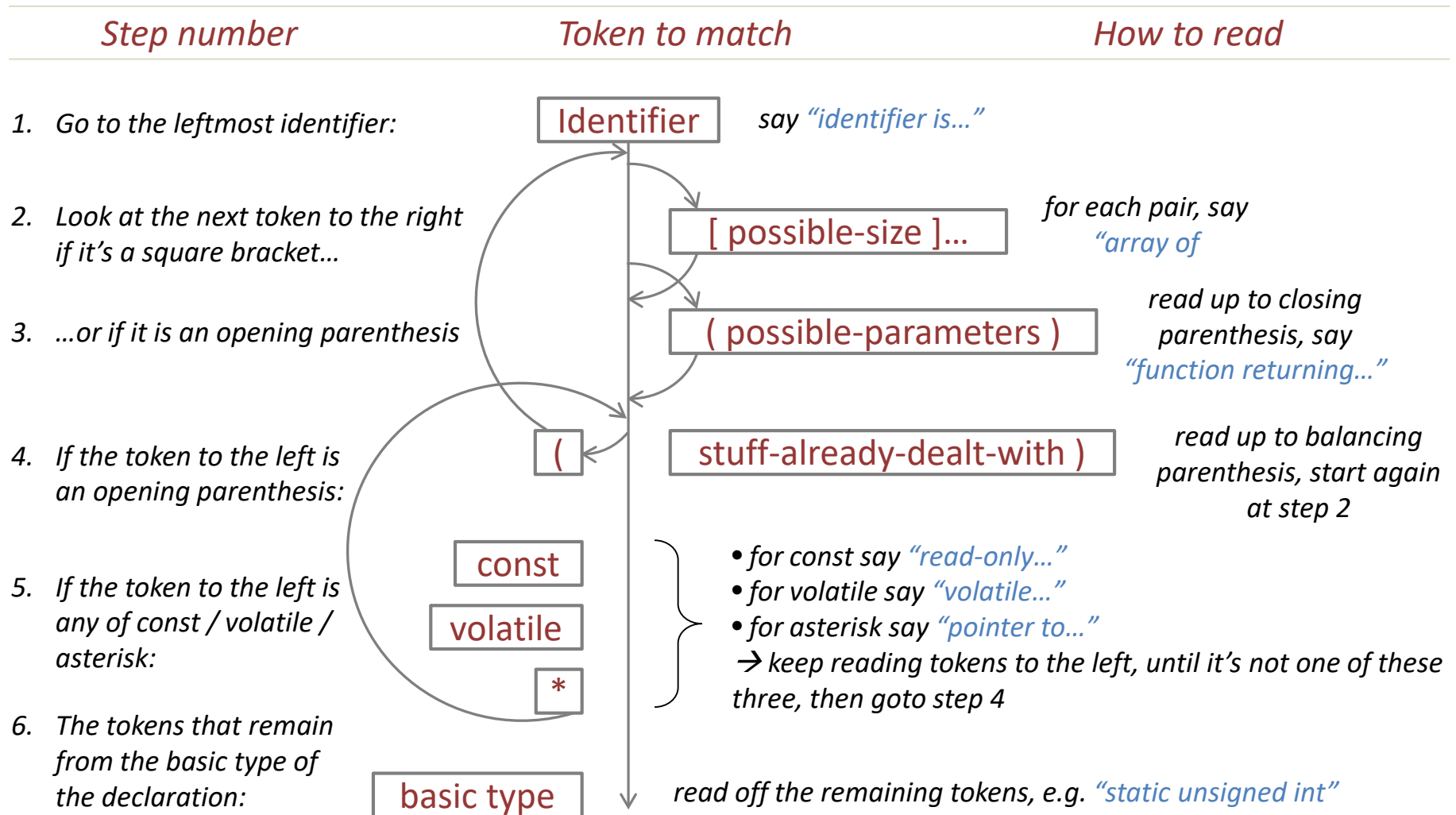
1.	A	<code>next</code>	„next is ein ...“
2.	B.3	<code>*</code>	„...Pointer auf...“
3.	B.1	<code>()</code>	„...“ (Gruppierung der Deklaration)
4.	B.2.1	<code>()</code>	„...eine Funktion mit Rückgabewert...“
5.	B.3	<code>*</code>	„...Pointer auf...“
6.	B.4	<code>* <b>const</b></code>	„...einen read-only Pointer auf...“

```
static unsigned int * const (*next)();
```

1.	A	<code>next</code>	„next is ein ...“
2.	B.3	<code>*</code>	„...Pointer auf...“
3.	B.1	<code>()</code>	„...“ (Gruppierung der Deklaration)
4.	B.2.1	<code>()</code>	„...eine Funktion mit Rückgabewert...“
5.	B.3	<code>*</code>	„...Pointer auf...“
6.	B.4	<code>* const</code>	„...einen read-only Pointer auf...“
7.	B.6	<code>static unsigned int</code>	„...static unsigned int.“

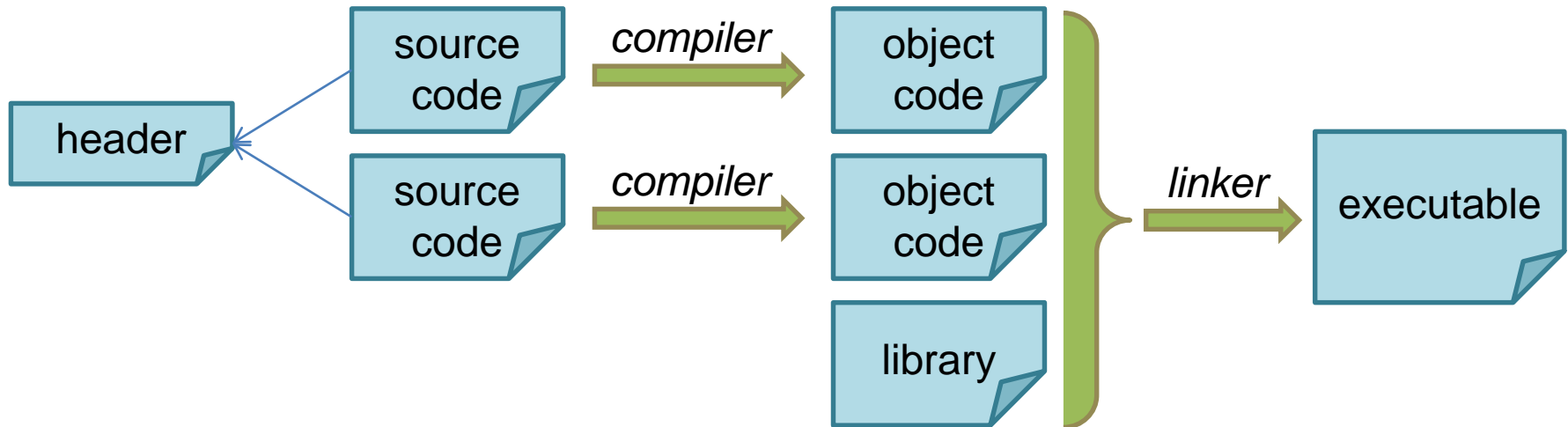
```
static unsigned int * const *(*next)();
```

1.	A	next	„next is ein ...“
2.	B.3	*	„...Pointer auf...“
3.	B.1	( )	„...“ (Gruppierung der Deklaration)
4.	B.2.1	( )	„...eine Funktion mit Rückgabewert...“
5.	B.3	*	„...Pointer auf...“
6.	B.4	* const	„...einen read-only Pointer auf...“
7.	B.6	static unsigned int	„...static unsigned int.“



- C Basics
  - Mitgelieferte Datentypen
  - Arrays, Structs, Unions
- Pointer und Pointer-Arithmetik
- Funktionen
- Speicherverwaltung
- Deklarationen
- Compiling und Linking

- Artefakte
  - Source-Code Dateien (inklusive header-Dateien)
  - Object-Code Dateien (inclusive Bibliotheken)
  - Executable
- Werkzeuge
  - Compiler
  - Linker



## ■ Compiler

- Kompiliert eine Source-Datei in eine Object-Datei:
  - `extern` Deklarationen → undefinierte Symbole
  - Globale Definitionen → Definierte Symbole
  - Lokale Definitionen → Lokale Symbole
- Nutzt header-Dateien für wiederverwendbare Vorwärtsdeklarationen

## ■ Linker

- Verbindet Object-Dateien (inklusive Bibliotheken) in einer ausführbaren Datei
- Löst undefinierte Symbole der individuellen Object-Dateien auf
- *Dynamische Bindung* ermöglicht das erhalten von nicht definierten Symbolen, damit diese von zugehörigen DLLs (Dynamic Link Libraries) zur Laufzeit nachgeladen werden können

- C
  - Keine Objektorientierung, keine Klassen, nur structs
  - Programmablauf wird von einer Funktionsmenge vorgegeben
    - Ausführung beginnt in Funktion `main()`
- C++
  - Zusätzliche Objektorientierung
  - Kann immer noch Funktionen und Variablen außerhalb von Klassen verwenden
  - Kann immer noch prozedurale Programmierung ohne Objektorientierung darstellen
- Java
  - Strikte Objektorientierung (erlaubt trotzdem primitive Datentypen)
    - Alles muss in einer Klasse enthalten sein (auch `static void main()`)
  - Source-Code Dateien werden nach Klassengrenzen organisiert



- C immer noch eine weit verbreitete Sprache
  - Besonders für eingebettete Systeme (embedded systems) und Hochleistungsrechnen (high-performance computing, HPC)
    - Aufgrund von mehr Freiheitsgraden ([manuelle Speicherverwaltung,...](#))
- Obwohl es syntaktisch oft ähnlich zu Java ist, gibt es wichtige Unterschiede:
  - Pointer
  - Keine festen Größen für mitgelieferte Datentypen
  - keine Default-Werte

# Fragen?

# Appendix :Compiling and Linking: Makefiles

- Allow for automation of the build process
- Define targets for compiling and linking
- Keep track of dependencies between artefacts

```
CC = g++
FLAGS = -Wall -g
hafas.o: hafas.cc
    $(CC) -c hafas.cc $(FLAGS) -o hafas.o
dijkstra.o: dijkstra.cc
    $(CC) -c dijkstra.cc $(FLAGS) -o dijkstra.o
hafas: hafas.o dijkstra.o
    $(CC) hafas.o dijkstra.o $(FLAGS) -o hafas
```

[Wilhalm2004]

- [Linden1994] Peter van der Linden, „Expert C Programming“, Prentice Hall, 1994
- [Meyer1997] Bertrand Meyer, „Object-oriented Software Construction“, 2nd Edition, Prentice Hall, 1997
- [SGI1994] SGI Standard Template Library Programmer's Guide, 1994,  
<http://www.sgi.com/tech/stl>
- [Ullenboom2004] Christian Ullenboom, „Java ist auch eine Insel“, 4th Edition, Galileo Computing, 2004
- [Wilhalm2004] Thomas Willhalm, „Von Java nach C++“, Internal Report, KIT, 2004,  
<http://digbib.ubka.uni-karlsruhe.de/volltexte/1000001246>

# Danksagung

- Vorlesungsmaterialien von Prof. Dr. Oliver Hummel wurden als Basis verwendet
- Unterstützung bei der technischen und inhaltlichen Gestaltung des Vorlesungsmaterials leisteten:  
Jóakim v. Kistowski