Outline

Variablen

- Deklaration: «*Datentyp*» «*Variablenname*»
- Datentyp bestimmt Größe in Bytes: `sizeof`
- Beispiel: `long int v;`
  - Größe: 4 Bytes
  - `v` ist Stück im Speicher, der 4 Bytes lang ist
  - Speicherzugriff? Über Adressen!
  - Pointer (Zeiger) bilden Konzept vom Speicherblock direkt in C.

Zeiger

- Deklaration: «*Datentyp*» * «*Zeigername*»
- Zeiger sind typisierte Speicheradressen
- Dereferenzierung mit *: Zugriff auf den *Inhalt* des Speicherblocks als Variable des Datentyps
- Referenzierung mit &: Die *Adresse* des Speichers einer Variable eines Datentyps.

Zeiger

```c
/* ref */

int v = 3;
printf("%p -> %d\n",&v,v);
```

Ausgabe:

```
0x251241aa -> 3
```

```c
/* deref */

int*v=(int*)malloc(sizeof(int));
printf( "%d\n", *v );
*v = 3;
printf( "%p -> %d\n", v, *v );
printf( "%p, %p\n", &v, &*v );
```

Ausgabe:

```
-64123
0x648010 -> 3
0x7fff65636b98, 0x648010
```

Zeiger

- Speicher muss alloziert sein
- Inhalt in alloziertem Speicher undefiniert! Also: stets initialisieren!
- Adresse einer Variable: `&v`, zeigt immer auf erstes Byte im Speicher
- Spezieller Zeigertyp: `void *` – Zeiger ohne Typ
- Spezieller Zeiger: `NULL` – Nullpointer
- Zur Erinnerung: Felder werden durch Zeiger auf das erste Element gespeichert.

Funktionsaufrufe

Call-by-Value Kopie des Wertes in der Variable

- Orginal Wert in Funktion nicht änderbar, nur Kopie!
- Achtung: *deep copy* und *flat copy*.

Call-by-Reference Kopie der Referenz auf Speicher der Variable

- Orginal Wert kann in Funktion geändert werden. Vorsicht Seiteneffekte!
- Schnell, da nur Speicheraddresse kopiert werden muss.

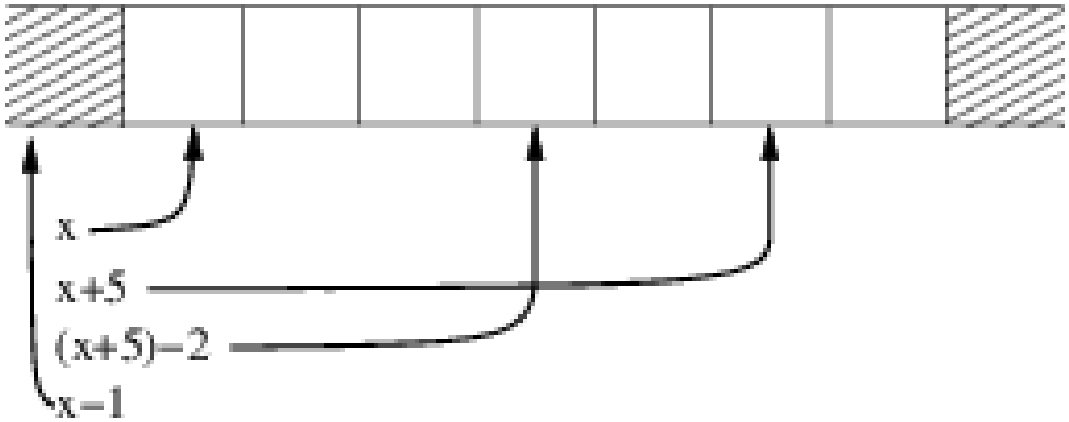
Call-By-Value

```
1  void vertausche( int a, int b )
2  {
3      int c = a;
4      a = b;
5      b = c;
6  }
7
8  int main( int argc, char *argv[] )
9  {
10     int a = 3;
11     int b = 4;
12     vertausche( a, b );
13     printf( "%d %d\n", a, b );
14     return EXIT_SUCCESS;
15 }
```

Call-By-Reference

```
1  void vertausche( int *a, int *b )
2  {
3      int c = *a;
4      *a = *b;
5      *b = c;
6  }
7
8  int main( int argc, char *argv[] )
9  {
10     int a = 3;
11     int b = 4;
12     vertausche( &a, &b );
13     printf( "%d %d\n", a, b );
14     return EXIT_SUCCESS;
15 }
```

Zeigerarithmetik



- Zeiger + Ganzzahl
- Zeiger – Ganzzahl
- Zeiger++, ++Zeiger
- Zeiger−−, −−Zeiger
- Zeiger – Zeiger
- Feld- und Zeigeroperationen sind äquivalent:

```
*pointer + i = pointer[i]
```

Zeichenketten: Beispiel

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int string_laenge( char s[] )
4  {
5      int i = 0;
6      while( s[i] != '\0' )
7      {
8          i++;
9      }
10     return i;
11 }
12 char name[] = "Hallo Welt!";
13 int main( int argc, char *argv[] )
14 {
15     printf( "%d\n", string_laenge( name ) );
16     return EXIT_SUCCESS;
17 }
```

Zeigerarithmetik: Beispiel

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int string_laenge( char *s )
4  {
5      char *p = s;
6      while( *p != '\0' )
7      {
8          p++;
9      }
10     return p-s;
11 }
12 char name[] = "Hallo Welt!";
13 int main( int argc, char *argv[] )
14 {
15     printf( "%d\n", string_laenge( name ) );
16     return EXIT_SUCCESS;
17 }
```

Zusammengesetzte Datentypen

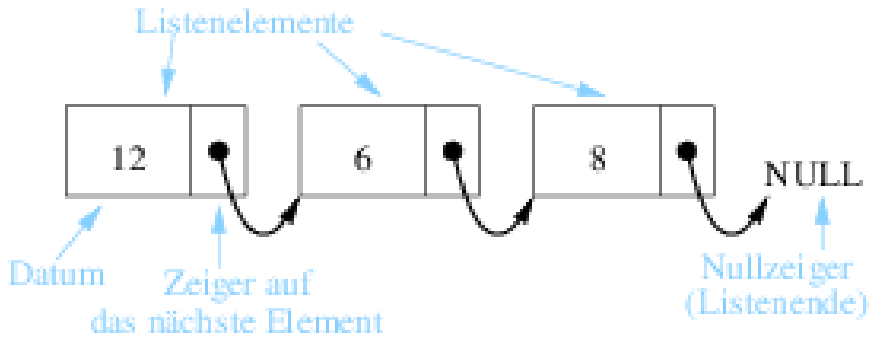
- Oft enge Beziehung zwischen einzenen Daten.
- Beispiel: Adresse = Name, Strasse, Hausnummer, PLZ, Ortsname
- Daten zusammenfassen mit `struct`

```
1 struct Address {
2   char* name;
3   char* street;
4   unsigned int number;
5   unsigned char postalCode[5];
6   char* city;
7 };
```

- Init: `struct Address adr = { "Mathias Goldau", ... };`
- Zugriff: `adr.number = 10;`
- Ebenso: `typedef struct {...} Address;`

Strukturen und Zeiger können benutzt werden um grundlegende Datenstrukturen zu programmieren. Beispiel: Einfach verkettete Liste.

```
1  struct ListenElement
2  {
3      int datum;
4      struct ListenElement *naechstes;
5  };
6
7  int main( int argc, char *argv[] )
8  {
9      struct ListenElement le;
10     le.datum = 3;
11 }
```



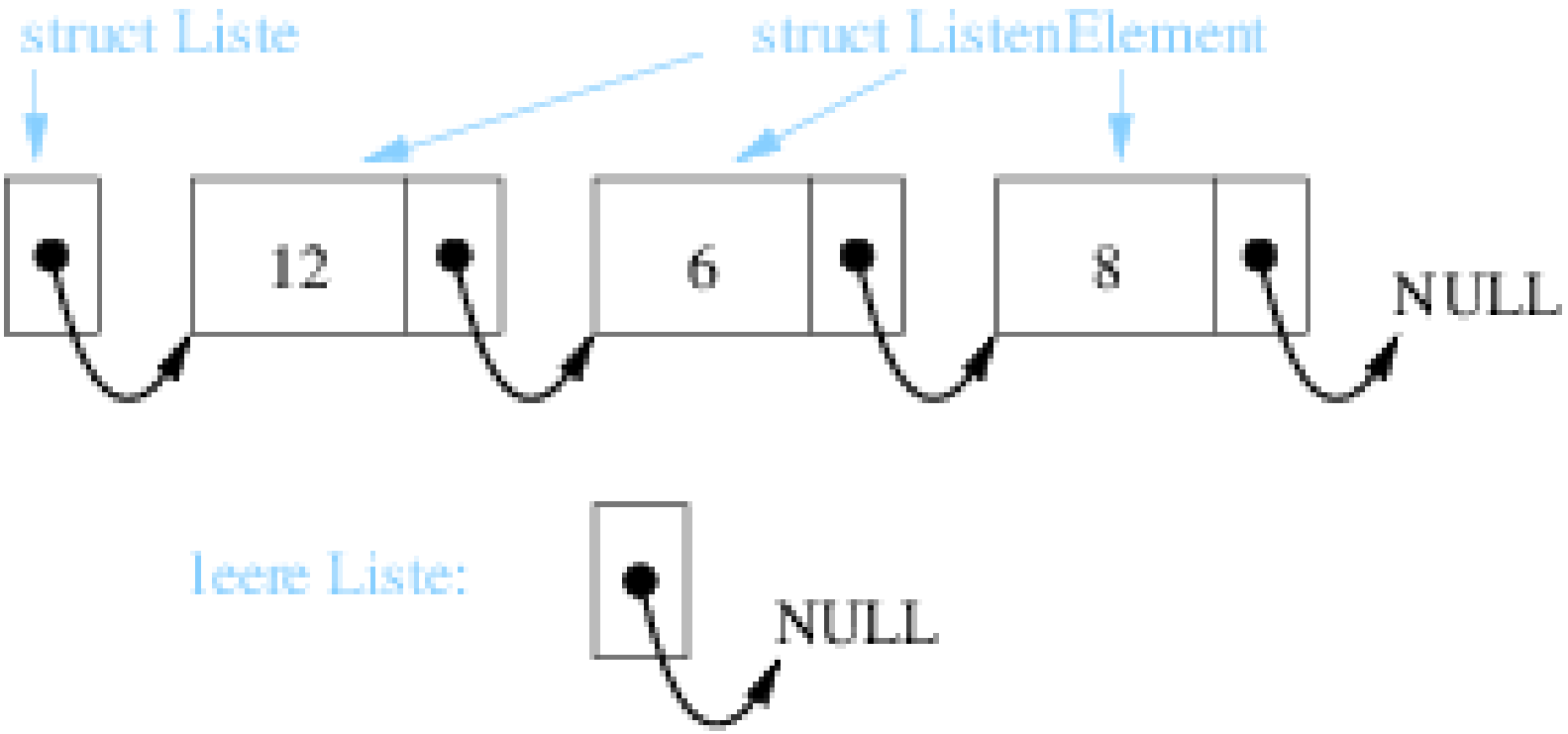
Modularisierung

- Trennung von dem *Was* das Modul bietet und dem *Wie* das Modul arbeitet.
- Damit insb. Trennung in *public* und *private* möglich
- Vorwärts-Deklaration:
  - Muss selbe Funktionssignatur wie bei Definition haben. Return-type gehört nicht dazu!
  - Anweisungsblock durch ein Semikolon ersetzen.
  - Beispiel: `Address* crawlNextAddress( char* url );`

Header-Datei/Deklarationen

```
1  struct Liste;
2
3  /* Erzeugt eine neue Liste.
4   * Diese enthält zunächst keine Einträge. */
5  struct Liste * liste_erzeugen( void );
6
7  /* Lösche die Liste liste */
8  void liste_loeschen( struct Liste *liste );
9
10 /* Füge Element elem an die Liste liste vorne an */
11 void liste_voranstellen( struct Liste *liste, int elem );
12
13 /* Liste liste ausgeben */
14 void liste_ausgeben( struct Liste *liste );
```

Aufbau einer Liste



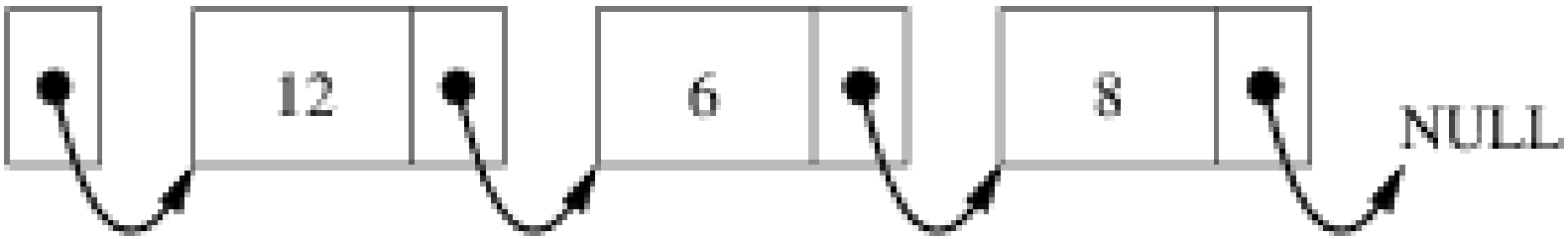
Implementierung Teil 1

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "liste.h"
4
5  struct ListenElement
6  {
7      int datum;
8      struct ListenElement *naechstes;
9  };
10
11 struct Liste
12 {
13     struct ListenElement *start;
14 };
```

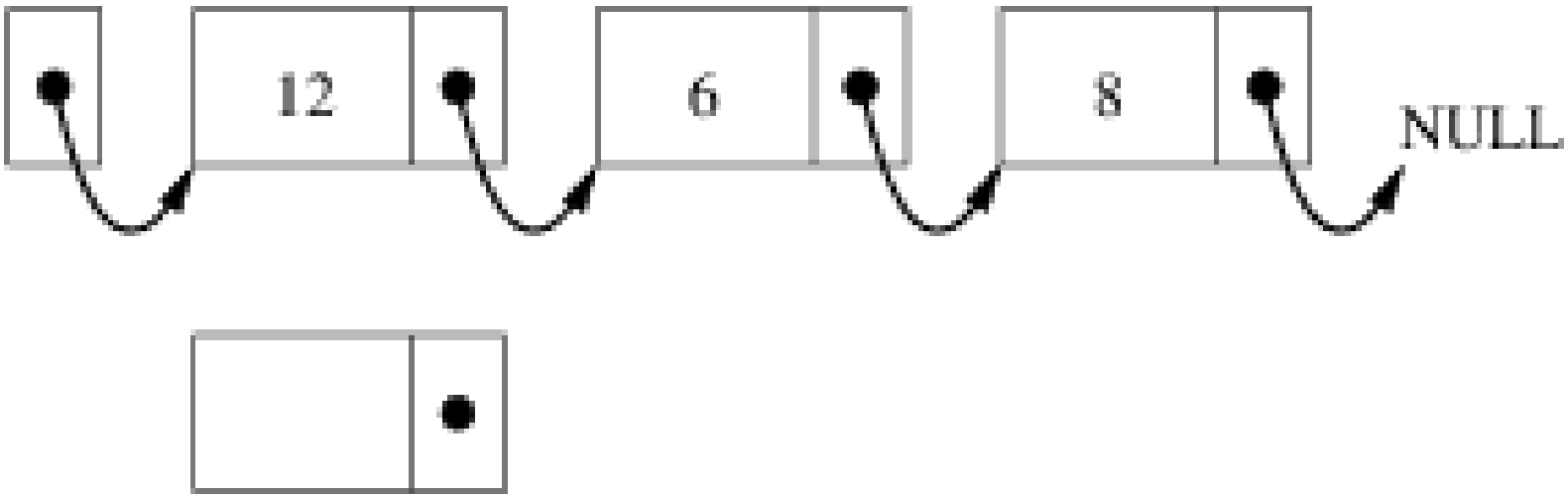
Implementierung Teil 2

```
1  struct Liste * liste_erzeugen( void )
2  {
3      struct Liste *l =
4          (struct Liste *)malloc(sizeof(struct Liste));
5      if( l == NULL )
6      {
7          fprintf( stderr, "Fehler: kein Speicher mehr!\n" );
8          exit( EXIT_FAILURE );
9      }
10
11     /* Anfangs leere Liste -> erstes Element = NULL */
12     (*l).start = NULL;
13     return l;
14 }
```

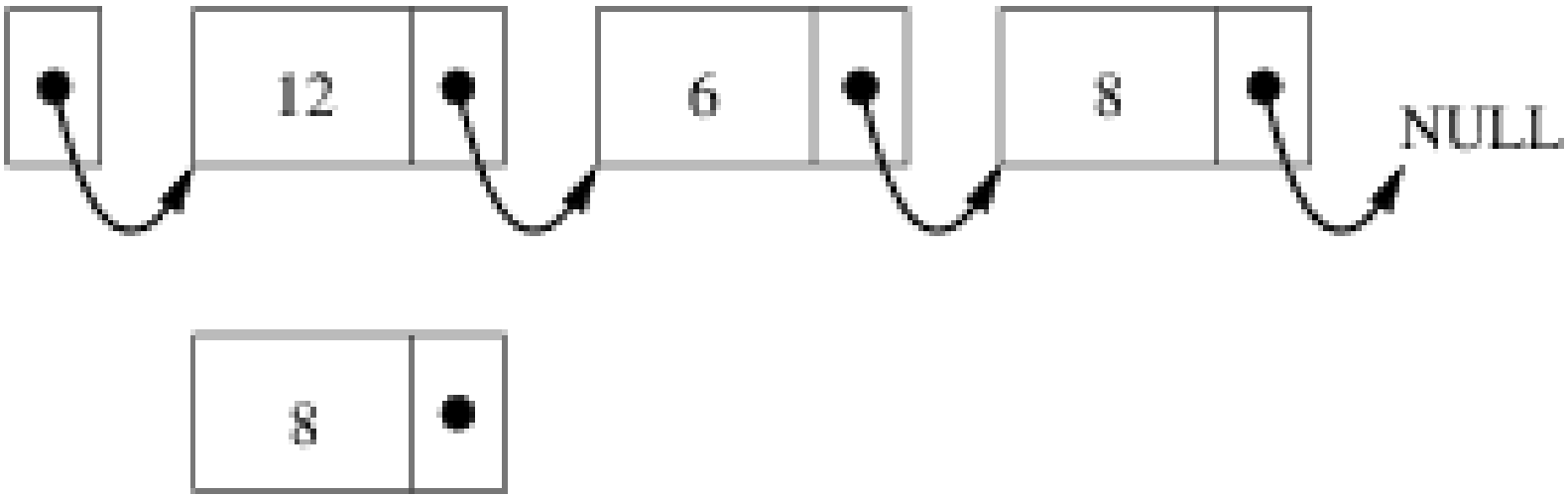
1. Gegeben



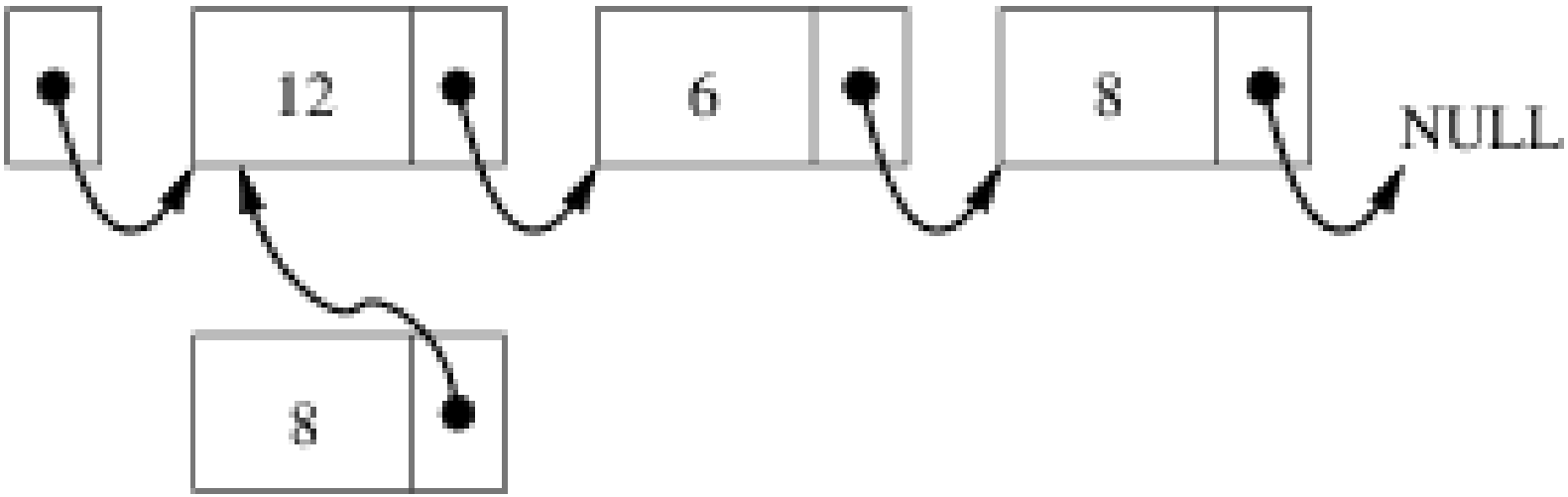
2. Neues `ListenElement` erzeugen



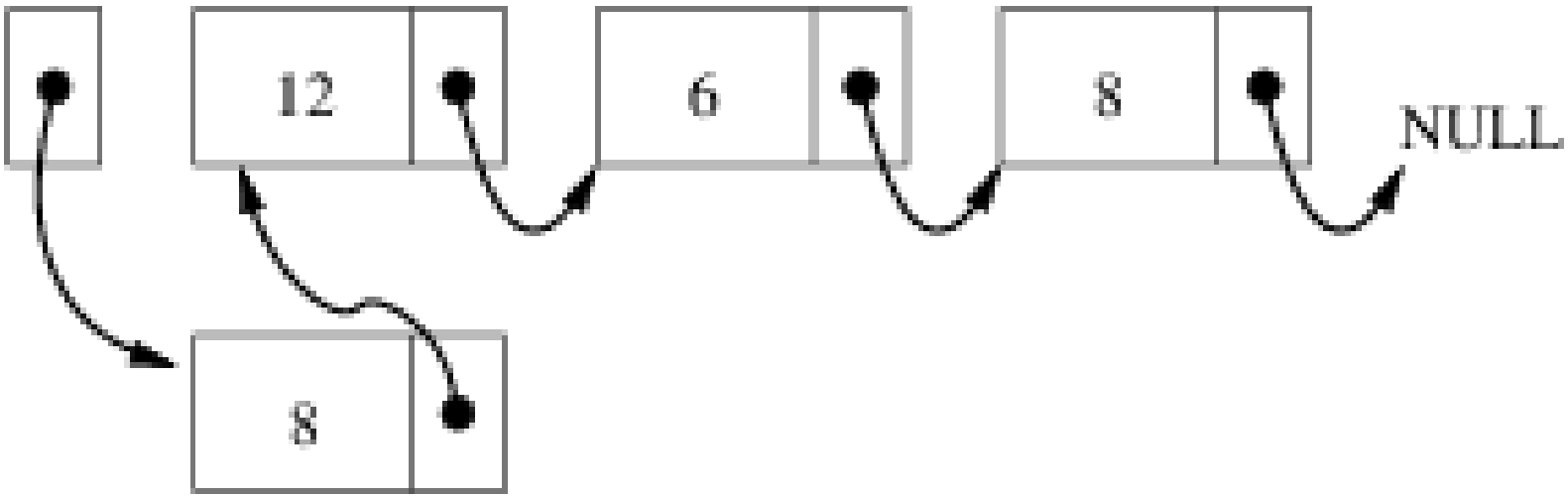
3. Wert in neues `ListenElement` schreiben



4. `naechstes`-Zeiger des neuen `ListenElement` auf Nachfolger setzen



5. `naechstes`-Zeiger des Vorgängers auf neues `ListenElement` setzen

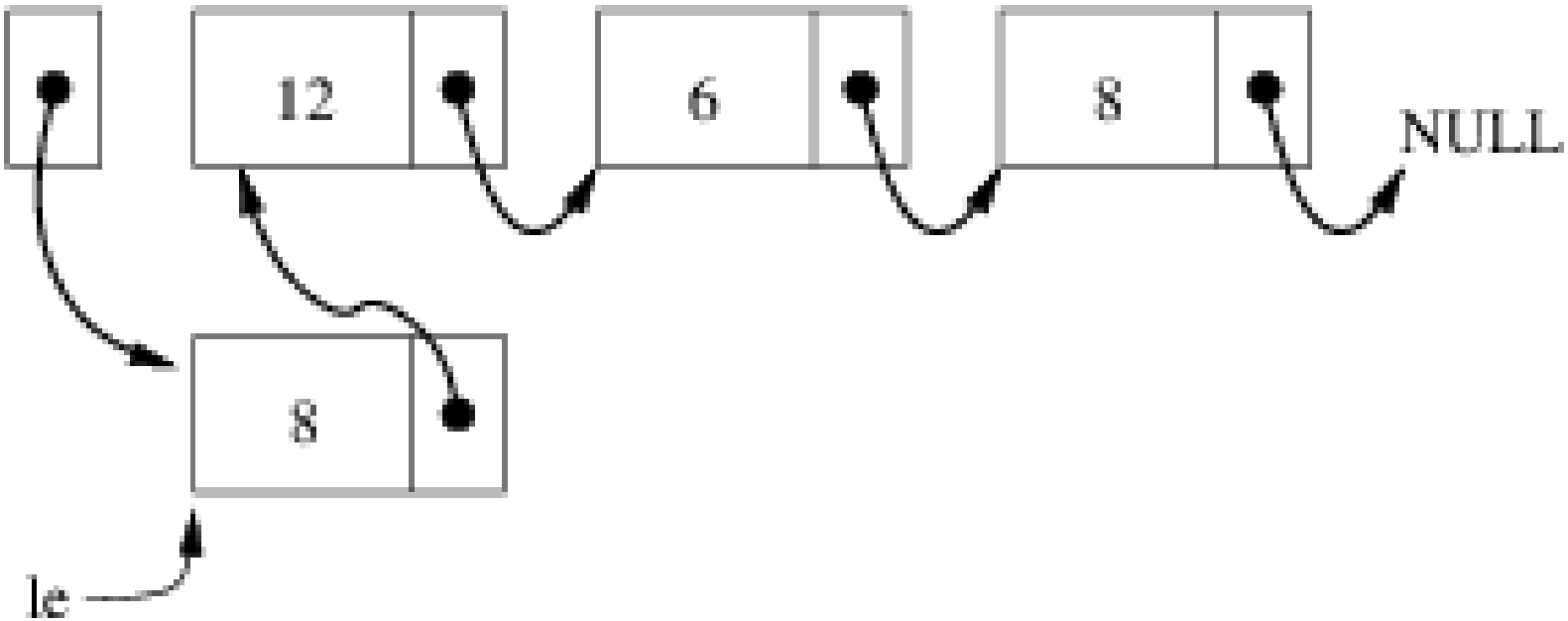


Implementierung Teil 3

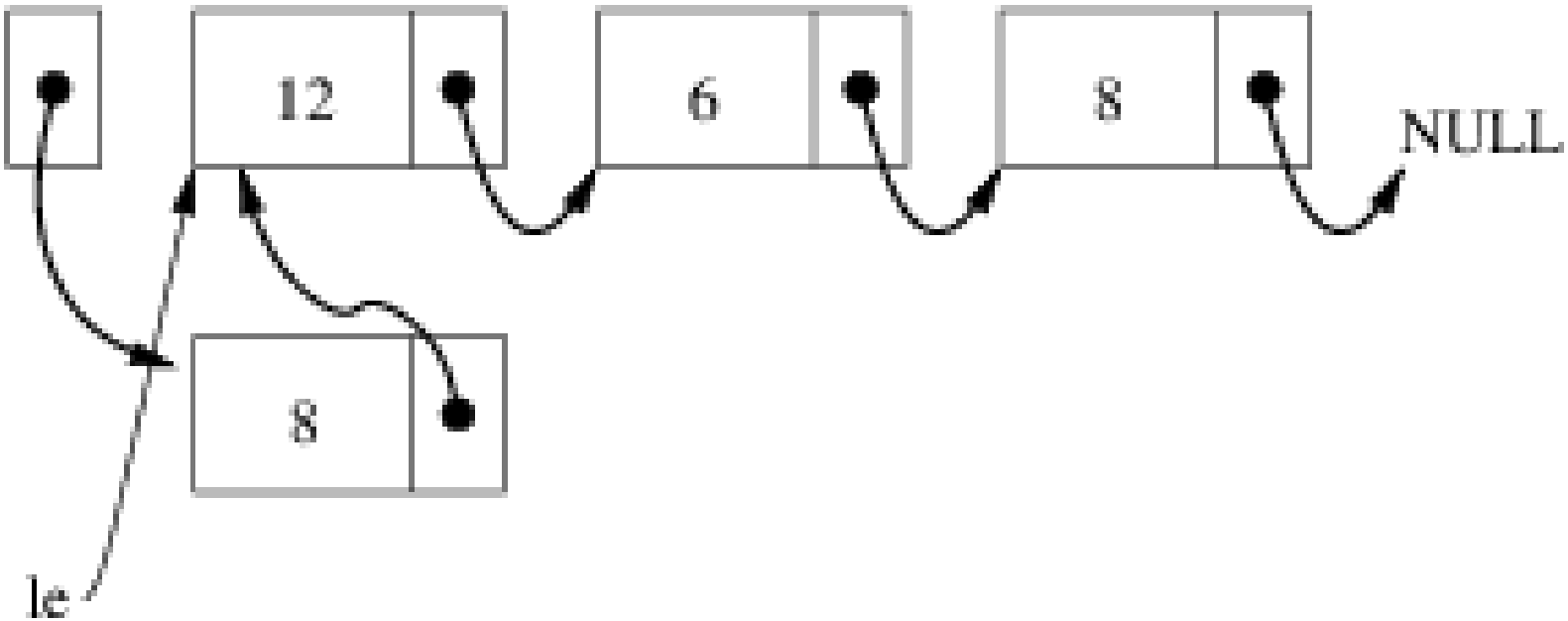
```
1  void liste_voranstellen( struct Liste *liste, int elem )
2  {
3      struct ListenElement *le;
4      if( liste == NULL )
5      {
6          return;
7      }
8      le = (struct ListenElement *)
9          malloc(sizeof(struct ListenElement));
10     if( le == NULL )
11     {
12         fprintf( stderr, "Fehler: kein Speicher mehr!\n" );
13         exit( EXIT_FAILURE );
14     }
15     (*le).datum = elem;
16     (*le).naechstes = (*liste).start;
17     (*liste).start = le;
18 }
```

Traversieren einer Liste

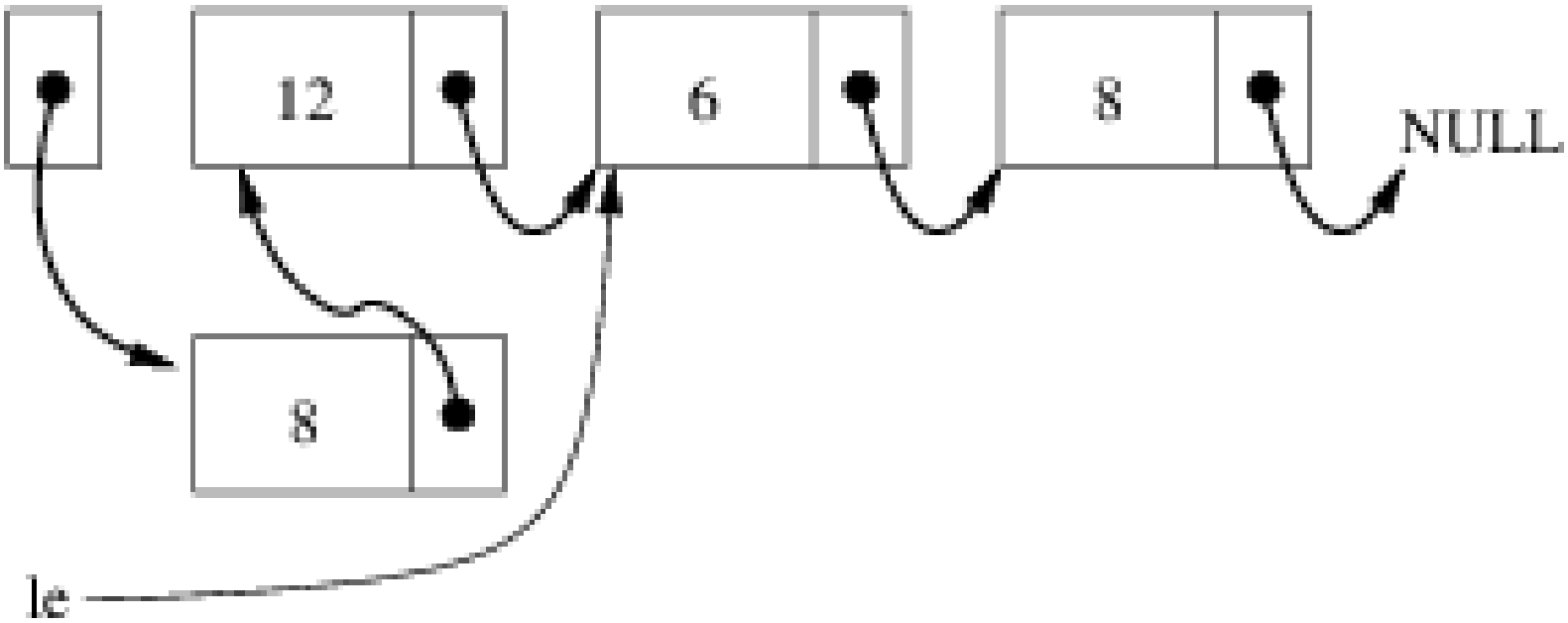
1. aktuelles `ListenElement le` ist `start` Pointer.



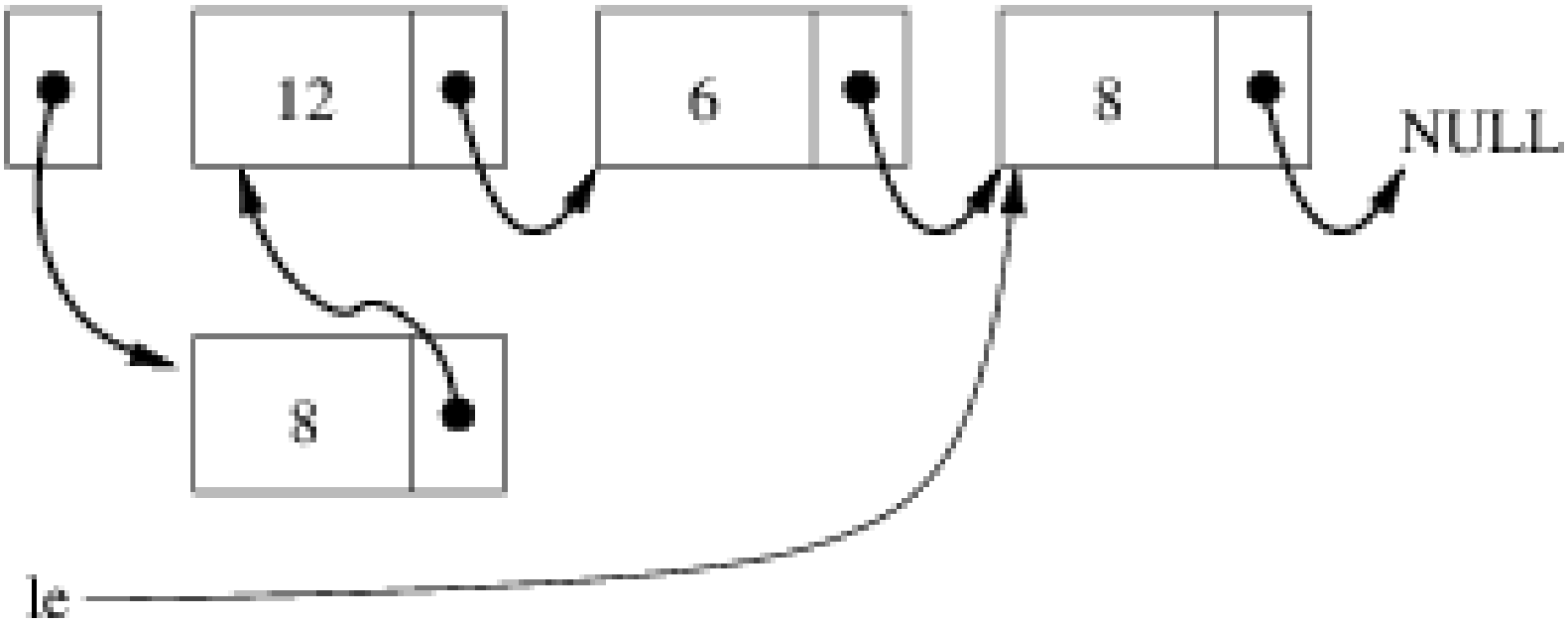
2. Nimm `naechstes`-Pointer des aktuellen Elements als aktuelles Element.
3. Gib Wert aus und wiederhole ab Schritt 2 bis ...



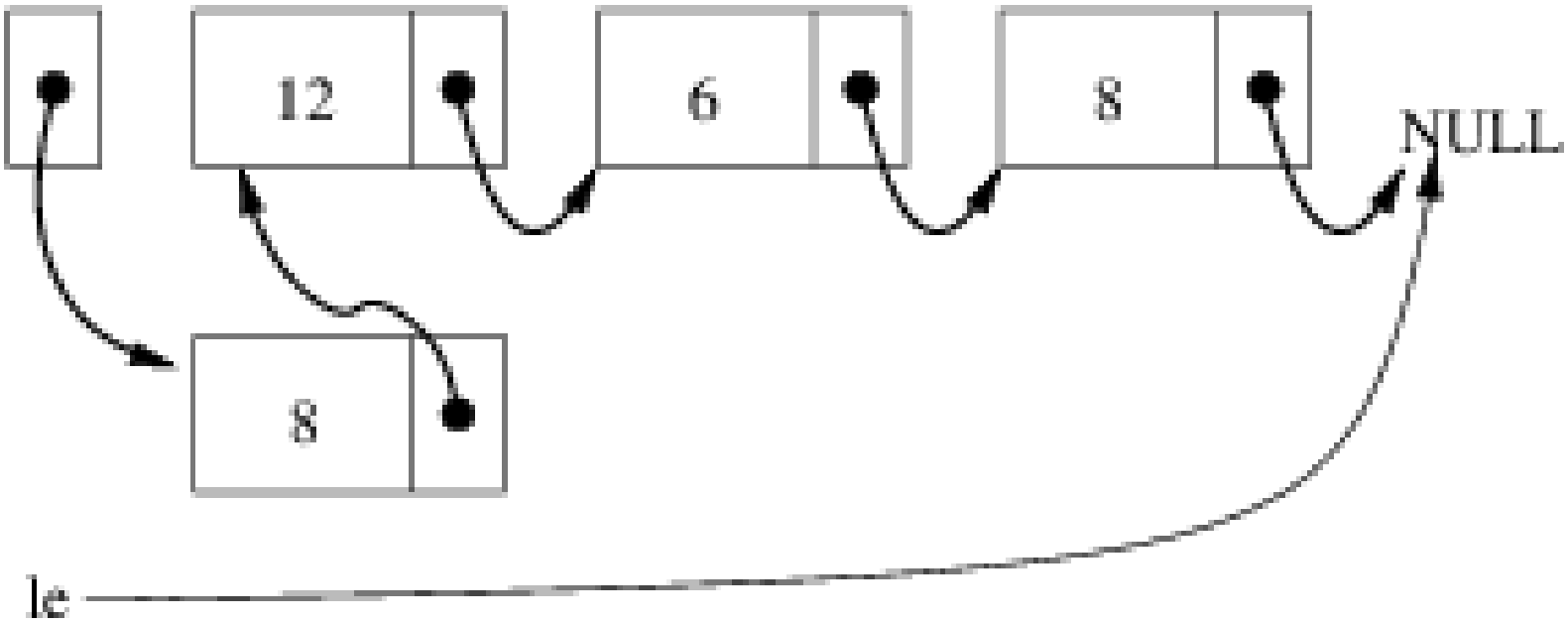
2. Nimm `naechstes`-Pointer des aktuellen Elements als aktuelles Element.
3. Gib Wert aus und wiederhole ab Schritt 2 bis ...



2. Nimm `naechstes`-Pointer des aktuellen Elements als aktuelles Element.
3. Gib Wert aus und wiederhole ab Schritt 2 bis ...



4. Wiederhole bis der `naechstes`-Zeiger auf `NULL` zeigt.



Implementierung Teil 4

```
1  void liste_ausgeben( struct Liste *liste )
2  {
3      struct ListenElement *le;
4      if( liste == NULL )
5      {
6          return;
7      }
8      le = (*liste).start;
9      while( le != NULL )
10     {
11         printf( "%d ", (*le).datum );
12         le = (*le).naechstes;
13     }
14 }
```

Hauptprogramm

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "liste.h"
4
5  int main( int argc, char *argv[] )
6  {
7      int i;
8      struct Liste *liste = liste_erzeugen();
9      for( i = 3; i < 12; i++ )
10     {
11         liste_voranstellen( liste, i );
12     }
13     liste_ausgeben( liste );
14     return EXIT_SUCCESS;
15 }
```

Compilieren von Modulen

- Zur Erinnerung an erstes C-Übungsseminar:
- Quick and Dirty:

```
$ gcc -o listemain liste.c listemain.c
```

- Modularisiert:

```
$ gcc -c liste.c
$ gcc -c listemain.c
$ gcc -o listemain liste.o listemain.o
```

Kommandozeilenargumente, Rückgabewert

```c
#include <stdio.h>
#include <stdlib.h>
int main( int argc, char *argv[] )
{
    if( argc == 1 )
    {
        printf( "keine Parameter gegeben!\n" );
        return EXIT_FAILURE;
    }
    else
    {
        int i;
        for( i = 1; i < argc; i++ )
        {
            printf( "%s\n", argv[i] );
        }
        return EXIT_SUCCESS;
    }
}
```