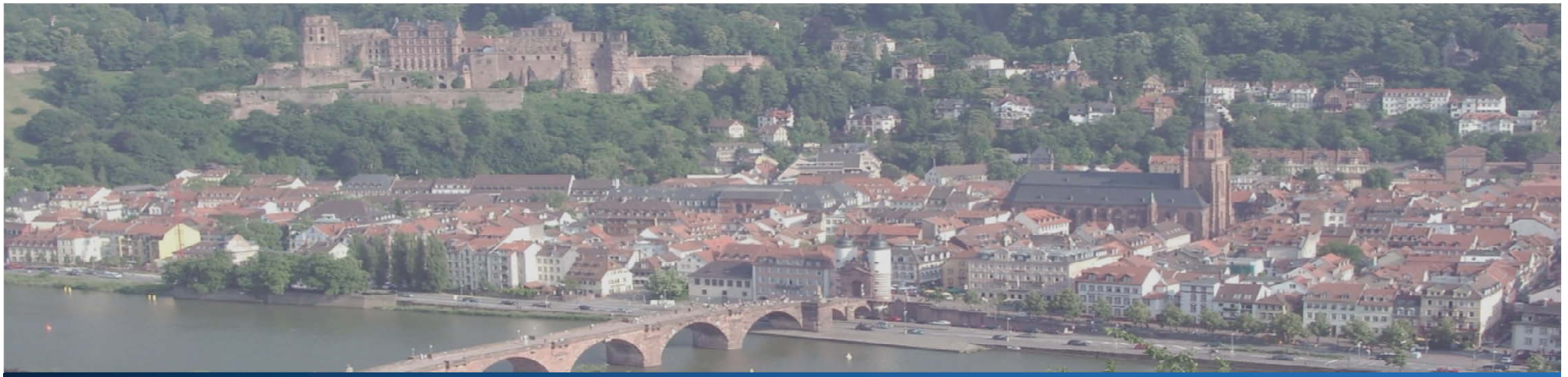
Einführung in die Praktische Informatik

Prof. Björn Ommer **HCI, IWR**

Computer Vision Group



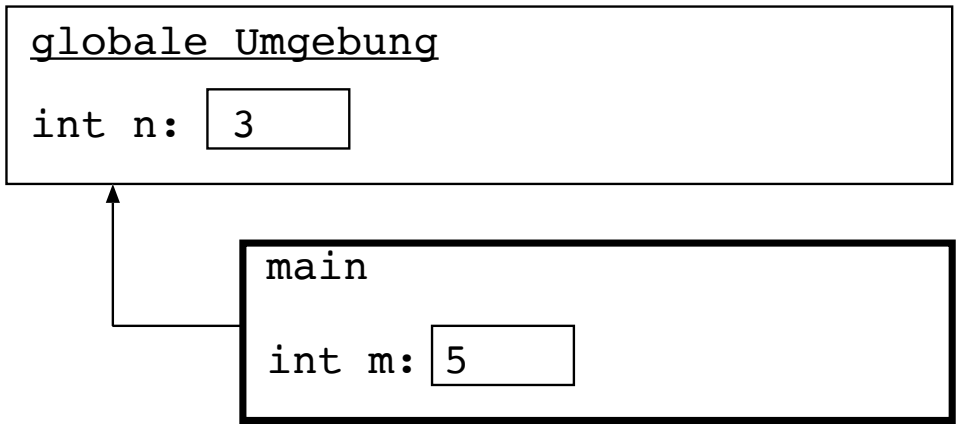
Zeiger & dynamische Datenstrukturen

- Zeiger
- Zeiger im Umgebungsmodell
- Call by reference
- Zeiger und Felder
- Felder als Argumente von Funktionen
- Zeiger auf zusammengesetzte Datentypen
- Problematik von Zeigern
- Dynamische Speicherverwaltung
- Die einfach verkettete Liste
- Endliche Menge

Zeiger

Wir können uns eine Umgebung als Sammlung von Schubladen (Orten) vorstellen, die Werte aufnehmen können. Jede Schublade hat einen Namen, einen Typ und einen Wert:

```
int n = 3;

int main()
{
  int m = 5;
}
```



Idee: Es wäre nun praktisch, wenn man so etwas wie „Erhöhe den Wert in *dieser* Schublade (Variable) um eins“ ausdrücken könnte.

Anwendung: Im Konto-Beispiel möchten wir nicht nur ein Konto sondern viele Konten verwenden. Hierzu benötigt man einen Mechanismus, der einem auszudrücken erlaubt, *welches* Konto verändert werden soll.

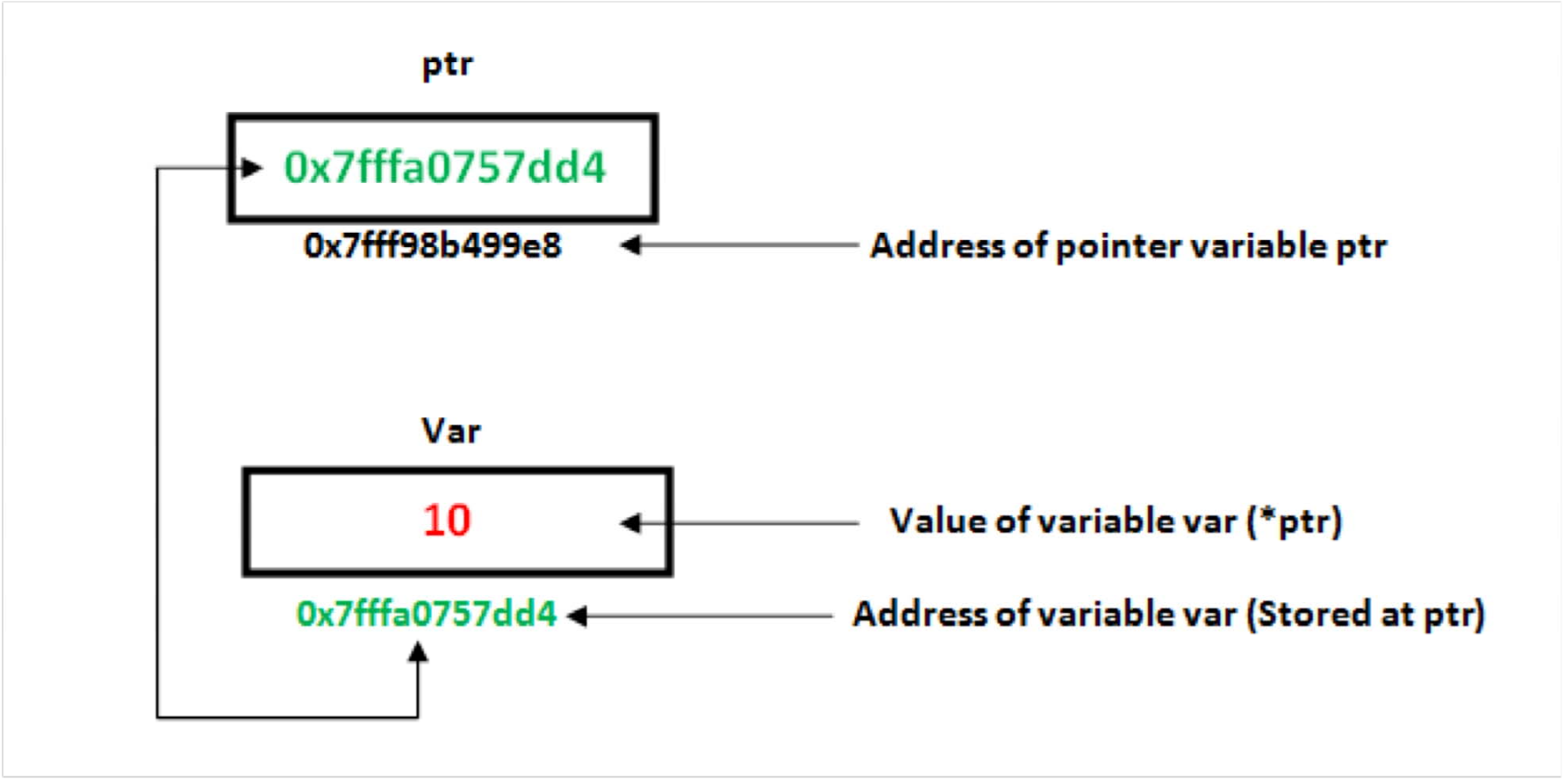
Idee: Man führt einen Datentyp Zeiger (Pointer) ein, der auf Variable (Schubladen) zeigen kann. Variablen, die Zeiger als Werte haben, heißen Zeigervariablen.

Bemerkung: Intern entspricht ein Zeiger der Adresse im physikalischen Speicher, an dem der Wert einer Variablen steht.

Notation: Die Definition „`int* x;`“ vereinbart, dass `x` auf Variablen (Schubladen) vom Typ `int` zeigen kann. Man sagt `x` habe den Typ „`int*`“.

Die Zuweisung „`x = &n;`“ lässt `x` auf den Ort zeigen, an dem der Wert von `n` steht.

Die Zuweisung „`*x = 4;`“ verändert den Wert der Schublade, „auf die `x` zeigt“.


ptr
0x7fffa0757dd4
0x7fff98b499e8
Address of pointer variable ptr
Var
10
Value of variable var (*ptr)
0x7fffa0757dd4
Address of variable var (Stored at ptr)

Zeiger im Umgebungsmodell

Im Umgebungsmodell gibt es eine Bindungstabelle, mittels derer jedem Namen ein Wert (und ein Typ) zugeordnet wird, etwa:

Name	Wert	Typ
`n`	3	`int`
`m`	5	`int`

Mathematisch entspricht das einer Abbildung w, die die *Symbole* $\mathtt{n}, \mathtt{m}$ auf die Wertemenge abbildet:

$$w : \{\mathtt{n}, \mathtt{m}\} \to \mathbb{Z}.$$

Die Zuweisung „`n = 3;`“ („=“ ist Zuweisung) manipuliert die Bindungstabelle so, dass nach der Zuweisung $w(\mathtt{n}) = 3$ („=“ ist Gleichheit) gilt.

Wenn auf der rechten Seite der Zuweisung auch ein Name steht, etwa „`n = m+1;`“, dann gilt nach der Zuweisung $w(\mathtt{n}) = w(\mathtt{m}) + 1$. Auf beide Namen wird also w angewandt.

Problem: Wir haben mehrere verschiedene Konten und möchten eine Funktion schreiben, die Beträge von Konten abhebt. In einer Variable soll dabei angegeben werden, von *welchem* Konto abgehoben wird.

Idee: Wir lassen Namen selbst wieder als Werte von (anderen) Namen zu, z. B.

Name	Wert	Typ
`n`	3	`int`
`m`	5	`int`
`x`	`n`	`int*`

Verwirklichung:

1. & ist der (einstellige) Adressoperator: „`x = &n`“ ändert die Bindungstabelle so, dass $w(\mathtt{x}) = \mathtt{n}$.

2. * ist der (einstellige) Dereferenzierungsoperator: Wenn `x = &n` gilt, so weist „`*x = 4;`“ dem Wert von n die Zahl 4 zu.

3. Den Typ eines Zeigers auf einen Datentyp `X` bezeichnet man mit „`X*`“.

Bemerkung:

- Auf der rechten Seite einer Zuweisung kann auf einen Namen der &-Operator genau einmal angewandt werden. Dieser $verhindert$ die Anwendung von w.

- Der $*$-Operator wendet die Abbildung w einmal auf das Argument rechts von ihm an. Der $*$-Operator kann mehrmals und sowohl auf der linken als auch auf der rechten Seite der Zuweisung angewandt werden.

Bemerkung: Auch eine Zeigervariable `x` kann wieder von einer anderen Zeigervariablen referenziert werden.

Diese hat dann den Typ „**int**∗∗“ oder „Zeiger auf eine **int**∗-Variable“.

Bemerkung: In C wird tendenziell die Notation „**int** ∗x;“ verwendet, wohingegen in C++ die Notation „**int**∗ x;“ empfohlen wird.

Allerdings impliziert die Notation „**int**∗ x, y;“ dass „`x` und `y` vom Typ **int**∗“ sind, was aber **nicht** zutrifft! (`y` ist vom Typ **int**)

Man liest „**int** ∗x“ als „`x` dereferenziert ist vom Typ **int**“ (in anderen Worten: „`x` zeigt auf **int**“).

Beispiel:

```
int     n = 3;
int     m = 5;
int*    x = &n;
int**   y = &x;
int***  z = &y;
```

Name	Wert	Typ
n	3	int
m	5	int
x	n	int*
y	x	int**
z	y	int***

Damit können wir schreiben

```
n = 4;     // das ist
*x = 4;    // alles
**y = 4;   // das
***z = 4; // gleiche !

x = &m;    // auch
*y = &m;   // das
**z = &m; // ist gleich !

y = &n;    // geht nicht, da n nicht vom Typ int*
y = &&n;   // geht auch nicht, da & nur
           // einmal angewandt werden kann
```

Zeiger & dynamische Datenstrukturen

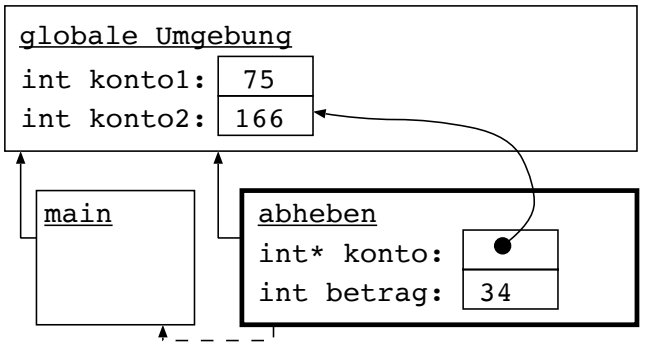
- Zeiger
- Zeiger im Umgebungsmodell
- Call by reference
- Zeiger und Felder
- Felder als Argumente von Funktionen
- Zeiger auf zusammengesetzte Datentypen
- Problematik von Zeigern
- Dynamische Speicherverwaltung
- Die einfach verkettete Liste
- Endliche Menge

Call by reference

Programm: Die Realisation der Kontoverwaltung könnte wie folgt aussehen:

```
int konto1 = 100;
int konto2 = 200;

int abheben( int* konto, int betrag )
{
  *konto = *konto - betrag;      // 1
  return *konto;                 // 2
}

int main()
{
  abheben( &konto1, 25 );        // 3
  abheben( &konto2, 34 );        // 4
}
```

Nach Marke 1, im zweiten Aufruf von abheben:



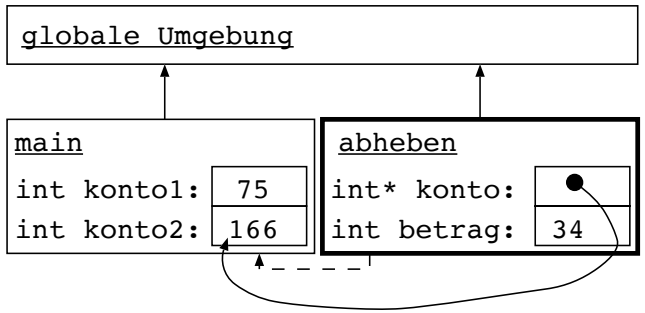
Definition: In der Funktion abheben nennt man betrag einen *call by value* Parameter und konto einen *call by reference* Parameter.

Bemerkung: Es gibt Computersprachen, die konsequent $call\ by\ value$ verwenden (z. B. Lisp/Scheme), und solche, die konsequent $call\ by\ reference$ verwenden (z. B. Fortran). Algol 60 war die erste Programmiersprache, die beides möglich machte.

Bemerkung: Die Variablen `konto1`, `konto2` im letzten Beispiel müssen nicht global sein! Folgendes ist auch möglich:

```
int abheben( int* konto, int betrag )
{
  *konto = *konto - betrag;   // 1
  return *konto;              // 2
}

int main()
{
  int konto1 = 100;
  int konto2 = 200;

  abheben( &konto1, 25 );     // 3
  abheben( &konto2, 34 );     // 4
}
```

Nach (1), zweiter Aufruf von abheben



Bemerkung: `abheben` darf `konto1` in `main` verändern, obwohl dieser Name dort nicht sichtbar ist! Zeiger können also die Sichtbarkeitsregeln durchbrechen und — im Prinzip — kann somit auch jede lokale Variable von einer anderen Prozedur aus verändert werden.

Bemerkung: Es gibt im wesentlichen zwei Situationen in denen man Zeiger als Argumente von Funktionen einsetzt:

- Der Seiteneffekt ist explizit erwünscht wie in `abheben` ($\rightarrow$ Objektorientierung).
- Man möchte das Kopieren großer Objekte sparen ($\rightarrow$ `const` Zeiger).

Referenzen in C++

Beobachtung: Obige Verwendung von Zeigern als Prozedurparameter ist ziemlich umständlich: Im Funktionsaufruf müssen wir ein & vor das Argument setzen, innerhalb der Prozedur müssen wir den ∗ benutzen.

Abhilfe: Wenn man in der Funktionsdefinition die Syntax **int**& x verwendet, so kann man beim Aufruf den Adressoperator & und bei der Verwendung innerhalb der Funktion den Dereferenzierungsoperator ∗ weglassen. Dies ist wieder sogenannter „syntaktischer Zucker“.

Programm: **(Konto mit Referenzen)**

```
int abheben( int& konto, int betrag )
{
  konto = konto - betrag;   // 1
  return konto;             // 2
}
```

```
int main()
{
  int konto1 = 100;
  int konto2 = 200;

  abheben( konto1, 25 );   // 3
  abheben( konto2, 34 );   // 4
}
```

Bemerkung: Referenzen können nicht nur als Funktionsargumente benutzt werden:

```
int  n = 3;
int& r = n; // independent reference
r = 5;      // selber Effekt n = 5;
```

Zeiger & dynamische Datenstrukturen

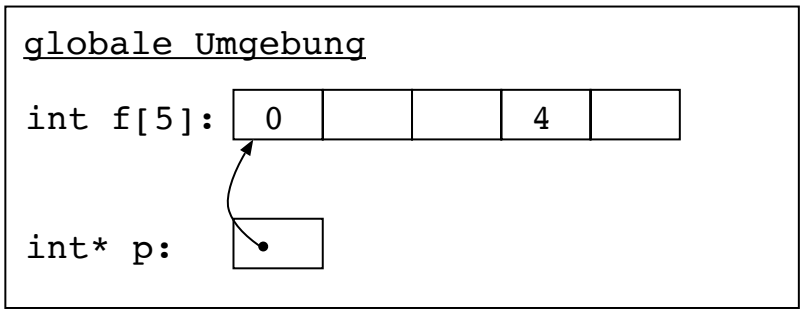
- Zeiger
- Zeiger im Umgebungsmodell
- Call by reference
- Zeiger und Felder
- Felder als Argumente von Funktionen
- Zeiger auf zusammengesetzte Datentypen
- Problematik von Zeigern
- Dynamische Speicherverwaltung
- Die einfach verkettete Liste
- Endliche Menge

Zeiger und Felder

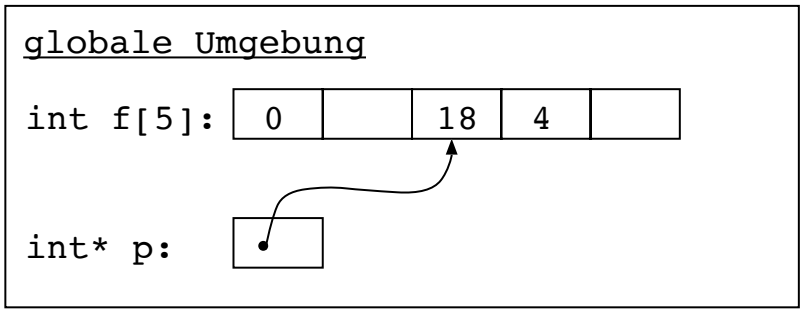
Beispiel: Zeiger und (eingebaute) Felder sind in C/C++ synonym:

```
int f[5];
int* p = f; // f hat Typ int*

...

p[0] = 0;
p[3] = 4;    // 1

p = &(f[2]);
*p = 18;     // p[0] = 18;
```

Nach Marke 1:



Am Ende



Bemerkung:

- Die Äquivalenz von eingebauten Feldern mit Zeigern ist eine höchst problematische Eigenschaft von C. Insbesondere führt sie dazu, dass man innerhalb einer mit einem Feld aufgerufenen Funktion die Länge dieses Feldes nicht direkt zur Verfügung hat. Diese muss anderweitig bekannt sein, oder es muss auf eine Bereichsüberprüfung verzichtet werden, was unter anderem ein Sicherheitsproblem darstellt (siehe auch die Diskussion bei `char`-Feldern).

- In C++ werden daher bessere Feldstrukturen (`vector`, `string`, `valarray`) in der Standard-Bibliothek STL (*Standard Template Library*) zur Verfügung gestellt.

Wiederholung

Zeigervariablen: Variablen, die auf Orte verweisen an denen der Wert einer Variablen gespeichert wird.

`int* p;` Zeiger auf eine `int` Variable.

`int n; p = &n; *p = 3;` Adresse-von- und Dereferenzierungsoperator.

`void f( int n, int* m );` Call-by-value und call-by-reference Parameter.

`void f( int n, int& m );` Referenzen in C++

`int a[10]; int* p = a; p[3] = 5;` (Eingebaute) Felder und Zeiger.

Zeiger erlauben die Änderungen beliebiger Werte von beliebigen Stellen innerhalb eines Programmes. Sie sollten mit großer Vorsicht eingesetzt werden.

Zeiger & dynamische Datenstrukturen

- Zeiger
- Zeiger im Umgebungsmodell
- Call by reference
- Zeiger und Felder
- Felder als Argumente von Funktionen
- Zeiger auf zusammengesetzte Datentypen
- Problematik von Zeigern
- Dynamische Speicherverwaltung
- Die einfach verkettete Liste
- Endliche Menge

Felder als Argumente von Funktionen

```
void f( int a[10] )
{
  a[3] = 17;
}
```

ist äquivalent zu

```
void f( int* a )
{
  a[3] = 17;
}
```

(Eingebaute) Felder werden also immer by reference übergeben!

Es findet in keinem Fall eine Bereichsprüfung statt und die Länge ist in f unbekannt.

Zeiger auf zusammengesetzte Datentypen

Beispiel: Es sind auch Zeiger auf Strukturen möglich. Ist `p` ein solcher Zeiger, so kann man mittels `p-><Komponente>` eine Komponente selektieren:

```
struct rational
{
  int n;
  int d;
};

int main()
{
  rational  q;
  rational* p = &q;
  (*p).n = 5;     // Zuweisung an Komponente n von q
  p->n = 5;       // eine Abkuerzung
}
```

Zeiger & dynamische Datenstrukturen

- Zeiger
- Zeiger im Umgebungsmodell
- Call by reference
- Zeiger und Felder
- Felder als Argumente von Funktionen
- Zeiger auf zusammengesetzte Datentypen
- Problematik von Zeigern
- Dynamische Speicherverwaltung
- Die einfach verkettete Liste
- Endliche Menge

Problematik von Zeigern

Beispiel: Betrachte folgendes Programm:

```
char* alphabet()
{
  char buffer[27];
  for ( int i=0; i<26; i++ ) buffer[i] = i + 65;
  buffer[26] = 0;
  return buffer;
}

int main()
{
  char* c = alphabet();
  print(c);
}
```

Beobachtung: Der Speicher für das lokale Feld ist schon freigegeben, aber den Zeiger darauf gibt es noch.

Bemerkung:

- Der gcc-Compiler warnt für das vorige Beispiel, dass ein Zeiger auf eine lokale Variable zurückgegeben wird. Er merkt allerdings schon nicht mehr, dass auch der Rückgabewert (Rückgabe-Adresse) `buffer+2` problematisch ist.

- Zeiger sind ein sehr *maschinennahes* Konzept (vgl. Neumann-Architektur). In vielen Programmiersprachen (z. B. Lisp, Java, etc.) sind sie daher für den Programmierer nicht sichtbar.

- Um die Verwendung von Zeigern sicher zu machen, muss man folgendes Prinzip beachten: Speicher darf nur dann freigegeben werden, wenn keine Referenzen darauf mehr existieren. Dies ist vor allem für die im nächsten Abschnitt diskutierte dynamische Speicherverwaltung wichtig.

Dynamische Speicherverwaltung

Bisher: Zwei Sorten von Variablen:

- Globale Variablen, die für die gesamte Laufzeit des Programmes existieren.
- Lokale Variablen, die nur für die Lebensdauer des Blockes/der Prozedur exisitieren.

Jetzt: Dynamische Variablen. Diese werden vom Programmierer explizit ausserhalb der globalen/aktuellen Umgebung erzeugt und vernichtet. Dazu dienen die Operatoren **new** und **delete**. Dynamische Variablen haben keinen Namen und können (in C/C++) nur indirekt über Zeiger bearbeitet werden.

Beispiel:

```
int m;
rational* p = new rational;
p->n = 4; p->d = 5;
m = p->n;
delete p;
```

Bemerkung:

- Die Anweisung `rational* p = new rational` erzeugt eine Variable vom Typ `rational` und weist deren Adresse dem Zeiger `p` zu. Man sagt auch, dass die Variable dynamisch allokiert wurde.

- Dynamische Variablen werden nicht auf dem Stack der globalen und lokalen Umgebungen gespeichert, sondern auf dem so genannten Heap. Dadurch ist es möglich, dass dynamisch allokierte Variablen in einer Funktion allokiert werden und die Funktion überdauern.

- Dynamische Variablen sind notwendig, um Strukturen im Rechner zu erzeugen, deren Größe sich während der Rechnung ergibt (und von der aufrufenden Funktion nicht gekannt wird).

- Die Größe der dynamisch allokierten Variablen ist nur durch den maximal verfügbaren Speicher begrenzt.

- Auch Felder können dynamisch erzeugt werden:

```
int   n = 18;
int* q = new int[n];   // Feld mit 18 int Eintraegen
q[5] = 3;
delete[] q;            // dynamisches Feld löschen
```

Probleme bei dynamischen Variablen

Beispiel: Wie schon im vorigen Abschnitt bemerkt, kann auf Zeiger zugegriffen werden, obwohl der Speicher schon freigegeben wurde:

```
int f()
{
  rational* p = new rational;
  p->n = 50;
  delete p;        // Vernichte Variable
  return p->n;     // Ooops, Zeiger gibt es immer noch
}
```

Beispiel: Wenn man alle Zeiger auf dynamisch allokierten Speicher löscht, kann dieser nicht mehr freigegeben werden ($\rightsquigarrow$ u.U. Speicherüberlauf):

```
int f()
{
  rational* p = new rational;
  p->n = 50;
  return p->n;   // Ooops, einziger Zeiger verloren
}
```

Problem: Es gibt zwei voneinander unabhängige Dinge, den Zeiger und die dynamische Variable. Beide müssen jedoch in konsistenter Weise verwendet werden. C++ stellt das nicht automatisch sicher!

Abhilfe:

- Manipulation der Variablen und Zeiger in Funktionen (später: Klassen) verpacken, die eine konsistente Behandlung sicherstellen.
- Benutzung spezieller Zeigerklassen (*smart pointers*).
- Die für den Programmierer angenehmste Möglichkeit ist die Verwendung von Garbage collection (= Sammeln von nicht mehr referenziertem Speicher).

Zeiger & dynamische Datenstrukturen

- Zeiger
- Zeiger im Umgebungsmodell
- Call by reference
- Zeiger und Felder
- Felder als Argumente von Funktionen
- Zeiger auf zusammengesetzte Datentypen
- Problematik von Zeigern
- Dynamische Speicherverwaltung
- Die einfach verkettete Liste
- Endliche Menge

Die einfach verkettete Liste

Zeiger und dynamische Speicherverwaltung benötigt man zur Erzeugung *dynamischer Datenstrukturen*.

Dies illustrieren wir am Beispiel der einfach verketteten Liste. Das komplette Programm befindet sich in der Datei `intlist.cc`.

Eine Liste natürlicher Zahlen

$$(12\ 43\ 456\ 7892\ 1\ 43\ 43\ 746)$$

zeichnet sich dadurch aus, dass

- die Reihenfolge der Elemente wesentlich ist, und
- Zahlen mehrfach vorkommen können.

Zur Verwaltung von Listen wollen wir folgende Operationen vorsehen

- Erzeugen einer leeren Liste.
- Einfügen von Elementen an beliebiger Stelle.
- Entfernen von Elementen.
- Durchsuchen der Liste.

Bemerkung: Der Hauptvorteil gegenüber dem Feld ist, dass das Einfügen und Löschen von Elementen schneller geschehen kann (es ist eine O(1)-Operation, wenn die Stelle nicht gesucht werden muss).

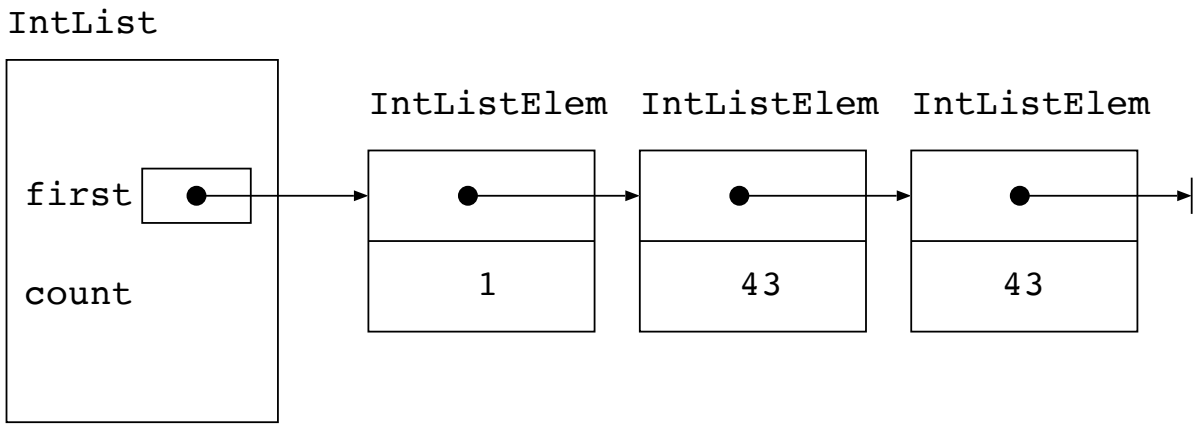
Eine übliche Methode zur Speicherung von Listen (natürlicher Zahlen) besteht darin ein $Listenelement$ zu definieren, das ein Element der Liste sowie einen Zeiger auf das nächste Listenelement enthält:

```
struct IntListElem
{
  IntListElem* next;   // Zeiger auf nächstes Element
  int value;           // Daten zu diesem Element
};
```

Um die Liste als Ganzes ansprechen zu können, definieren wir den folgenden zusammengesetzten Datentyp, der einen Zeiger auf das erste Element sowie die Anzahl der Elemente enthält:

```
struct IntList
{
  int count;           // Anzahl Elemente in der Liste
  IntListElem* first;  // Zeiger auf 1. Element der Liste
};
```

Das sieht also so aus:



Das Ende der Liste wird durch einen Zeiger mit dem Wert 0 gekennzeichnet.

Das klappt deswegen, weil 0 kein erlaubter Ort eines Listenelementes (irgendeiner Variable) ist.

Bemerkung: Die Bedeutung von 0 ist in C/C++ mehrfach überladen. In manchen Zusammenhängen bezeichnet es die Zahl 0, an anderen Stellen einen speziellen Zeiger. Auch der bool-Wert `false` ist synonym zu 0. In C++11 gibt es nun das Schlüsselwort nullptr .

Initialisierung

Folgende Funktion initialisiert eine `IntList`-Struktur mit einer leeren Liste:

```
void empty_list( IntList* l )
{
  l->first = 0;    // Liste ist leer
  l->count = 0;
}
```

Bemerkung: Die Liste wird *call-by-reference* übergeben, um die Komponenten ändern zu können.

Durchsuchen

Hat man eine solche Listenstruktur, so gelingt das Durchsuchen der Liste mittels

```
IntListElem* find_first_x( IntList l, int x )
{
  for ( IntListElem* p=l.first; p!=0; p=p->next )
    if ( p->value == x ) return p;
  return 0;
}
```

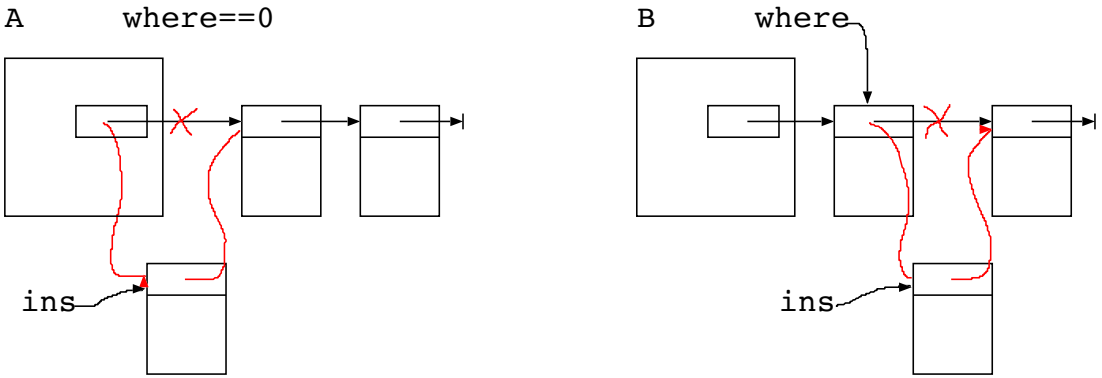
Einfügen

Beim Einfügen von Elementen unterscheiden wir zwei Fälle:

A Am Anfang der Liste einfügen.

B $Nach$ einem gegebenem Element einfügen.

Nur für diese beiden Fälle ist eine effiziente Realisierung der Einfügeoperation möglich. Es sind folgende Manipulationen der Zeiger erforderlich:



Programm: Folgende Funktion behandelt beide Fälle:

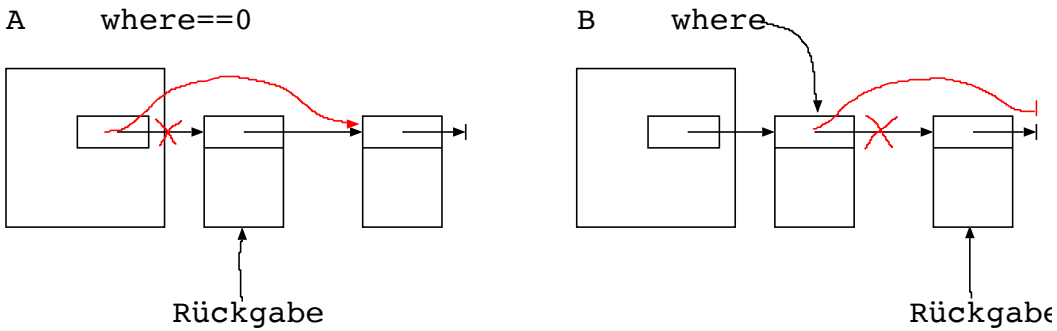
```
void insert_in_list( IntList* list, IntListElem* where,
                     IntListElem* ins )
{
  if ( where == 0 )
  { // fuege am Anfang ein
    ins->next = list->first;
    list->first = ins;
    list->count = list->count + 1;
  }
  else
  {
    // fuege nach where ein
    ins->next = where->next;
    where->next = ins;
    list->count = list->count + 1;
  }
}
```

Entfernen

Auch beim Entfernen von Elementen unterscheiden wir wieder ob

1. das erste Element gelöscht werden soll, oder
2. das Element $nach$ einem gegebenem.

entsprechend graphisch:



Programm: Beide Fälle behandelt folgende Funktion:

```
IntListElem* remove_from_list( IntList* list ,
                               IntListElem* where )
{
  IntListElem* p;   // das entfernte Element

  // where == 0 dann entferne erstes Element
  if ( where == 0 ) {
    p = list->first;
    if ( p != 0 )
    {
      list->first = p->next;
      list->count = list->count - 1;
    }
    return p;
  }
```

```
    // entferne Element nach where
    p = where->next;
    if ( p != 0 ) {
      where->next = p->next;
      list->count = list->count - 1;
    }
    return p;
}
```

Bemerkung:

- Es wird angenommen, dass `where` ein Element der Liste ist. Wenn dies nicht erfüllt sein sollte, so ist Ärger garantiert!

- Alle Funktionen auf der Liste beinhalten *nicht* die Speicherverwaltung für die Objekte. Dies ist Aufgabe des benutzenden Programmteiles.

Kritik am Programmdesign

- Ob die Anzahl der Elemente überhaupt benötigt wird, hängt von der konkreten Anwendung ab. Man hätte die Liste auch einfach als Zeiger auf Listenelemente definieren können:

```
struct list_element
{
  list_element* next;          // Zeiger auf nächstes
  list_element_type value;     // Datum dieses Elements
};
typedef list_element* list;
```

- Man wird auch Listen anderer Typen brauchen. Dies ist erst mit den später behandelten Werkzeugen wirklich befriedigend zu erreichen (Templates). Etwas mehr Flexibilität erhielte man aber über:

```
typedef int list_element_type;
```

  und Verwendung dieses Datentyps später.

- Die Liste ist ein Spezialfall eines **Containers**. Mit der *Standard Template Library (STL)* bietet C++ eine leistungsfähige Implementierung von Containern unterschiedlicher Funktionalität.

Bemerkung: Lässt man nur das Einfügen und Löschen am **Listenanfang** zu, so implementiert die Liste das Verhalten eines Stacks mit dem Vorteil, dass man die maximale Zahl von Elementen nicht im Voraus kennen muss.

Listenvarianten

- Bei der doppelt verketteten Liste ist auch der Vorgänger erreichbar und die Liste kann auch in umgekehrter Richtung durchlaufen werden.
- Listen, die auch ein (schnelles) Einfügen am Ende erlauben, sind zur Implementation von Warteschlangen (Queues) nützlich.
- Manchmal kann man zirkuläre Listen gebrauchen. Diese sind für simple Speicherverwaltungsmechanismen (reference counting) problematisch.
- In dynamisch typisierten Sprachen können Elemente beliebigen Typ haben (zum Beispiel wieder Listen), und die Liste wird zum Spezialfall einer Baumstruktur. Am elegantesten ist dieses Konzept wohl in der Sprache Lisp ($= \mathit{List\ Processing}$) verwirklicht.

Wiederholung Zeiger

Verwendungsmuster für Zeiger:

- Call by reference
- Dynamisch allokierte Variablen
- Dynamische Datenstrukturen
  - Felder variabler Größe
  - Listen, Bäume, etc.

Problematik des Speicherüberlaufs → interagiert mit dem Betriebssystem:

- Exceptions
- Test auf nullptr

Zeiger & dynamische Datenstrukturen

- Zeiger
- Zeiger im Umgebungsmodell
- Call by reference
- Zeiger und Felder
- Felder als Argumente von Funktionen
- Zeiger auf zusammengesetzte Datentypen
- Problematik von Zeigern
- Dynamische Speicherverwaltung
- Die einfach verkettete Liste
- Endliche Menge

Endliche Menge

Im Gegensatz zu einer Liste kommt es bei einer endlichen Menge

$$\{34\ 567\ 43\ 1\}$$

1. nicht auf die Reihenfolge der Mitglieder an und

2. können Elemente auch nicht doppelt vorkommen!

Schnittstelle

Als Operationen auf einer Menge benötigen wir

- Erzeugen einer leeren Menge.
- Einfügen eines Elementes.
- Entfernen eines Elementes.
- Mitgliedschaft in der Menge testen.

In der nachfolgenden Implementierung einer Menge von **int**-Zahlen enthalten die genannten Funktionen auch die Speicherverwaltung!

Wir wollen zur Realisierung der Menge die eben vorgestellte einfach verkettete Liste verwenden.

Datentyp: (Menge von Integer-Zahlen)

```
struct IntSet
{
  IntList list;
};
```

Man versteckt damit auch, dass `IntSet` mittels `IntList` realisiert ist.

Programm: (Leere Menge)

```
IntSet* empty_set()
{
  IntSet* s = new IntSet;
  empty_list( &s->list );
  return s;
}
```

Test auf Mitgliedschaft

Programm:

```
bool is_in_set( IntSet* s, int x )
{
  for ( IntListElem* p=s->list.first; p!=0; p=p->next )
    if ( p->value == x ) return true;
  return false;
}
```

Bemerkung:

- Dies nennt man sequentielle Suche. Der Aufwand ist $O(n)$ wenn die Liste n Elemente hat.
- Später werden wir bessere Datenstrukturen kennenlernen, mit denen man das in $O(\log n)$ Aufwand schafft (Suchbaum).

Einfügen in eine Menge

Idee: Man testet, ob das Element bereits in der Menge ist, ansonsten wird es am Anfang der Liste eingefügt.

```
void insert_in_set( IntSet* s, int x )
{
  if ( !is_in_set( s, x ) )
  {
    IntListElem* p = new IntListElem;
    p->value = x;
    insert_in_list( &s->list, 0, p );
  }
}
```

Bemerkung: Man beachte, dass diese Funktion auch die `IntListElem`-Objekte dynamisch erzeugt.

Ausgabe

Programm: (Ausgabe der Menge)

```
void print_set( IntSet* s )
{
  print( "{" );
  for ( IntListElem* p=s->list.first; p!=0; p=p->next )
    print( "␣␣", p->value, 0 );
  print( "}" );
}
```

Entfernen

Idee: Man sucht zuerst den Vorgänger des zu löschenden Elementes in der Liste und wendet dann die entsprechende Funktion für Listen an.

```
void remove_from_set( IntSet* s, int x )
{
  // Hat es ueberhaupt Elemente?
  if ( s->list.first == 0 ) return;

  // Teste erstes Element
  if ( s->list.first->value == x )
  {
    IntListElem* p = remove_from_list( &s->list, 0 );
    delete p;
    return;
  }

  // Suche in Liste, teste immer Nachfolger
```

```
    // des aktuellen Elementes
    for ( IntListElem* p=s->list.first; p->next!=0; p=p->next )
      if ( p->next->value == x )
      {
        IntListElem* q = remove_from_list( &s->list, p );
        delete q;
        return;
      }
}
```

Vollständiges Programm

Programm: (useintset.cc)

```
#include "fcpp.hh"
#include "intlist.cc"
#include "intset.cc"

int main()
{
  IntSet* s = empty_set();
  print_set( s );
  for ( int i=1; i<12; i=i+1 ) insert_in_set( s, i );
  print_set( s );
  for ( int i=2; i<30; i=i+2 ) remove_from_set( s, i );
  print_set( s );
}
```