# Algorithmen und Datenstrukturen

- Dynamische Datenobjekte
  - Pointer/Zeiger, Verkettete Liste
- Eigene Typdefinitionen

# Zeigeroperatoren & und *

- Ein Zeiger ist die Speicheradresse irgendeines Objektes.
- Eine Zeigervariable ist eine Variable, die gemäß Deklaration einen Zeiger auf ein Objekt des vereinbarten Typs enthält.
- Mit dem **Adressoperator &** kann man die Adresse einer Speicherzelle ermitteln. Der Adressoperator kann auf Variablen und Arrayelemente angewendet werden, nicht auf Ausdrücke (sie haben keinen Speicherplatz).
- Da ein Zeiger die Adresse eines Objektes enthält, kann auf das Objekt auf zwei Arten zugegriffen werden:
  - Direkt über den Namen
  - Indirekt über den Zeiger.
- Um indirekt zuzugreifen, wird der Operator * zum **Dereferenzieren** verwendet.

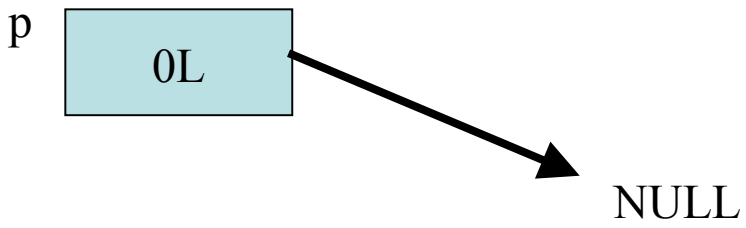
# Zeigeroperatoren & und *

- Wird eine Zeigervariable deklariert, so ist sie nicht automatisch initialisiert; sie zeigt igendwo hin. Will man sie auf einen definierten Wert setzen, aber noch nicht auf ein bestimmtes Objekt, so kann der spezielle Zeigerwert **NULL** verwendet werden.
- NULL wird intern als OL dargestellt; man sollte der besseren Lesbarkeit wegen aber den Namen NULL verwenden. Die Definition von Null ist im Header <cstddef> enthalten.
- In Ausdrücken, wie y = *p + 1; haben die Operatoren * und **& höheren** Rang als arithmetische Operatoren. D.h. **zuerst** wird oben das Objekt ermittelt (**dereferenziert**) dann wird der Wert um 1 erhöht. Achtung: y = *(p+1) möglich, hat aber andere Bedeutung.
- Verweise sind auch links in Zuweisungen möglich.

# Zeiger

- Beispiele für Zugriffe:

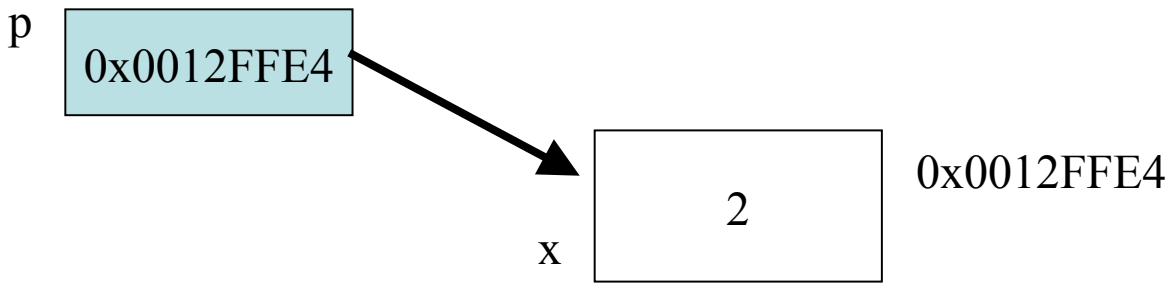
| C++-Fragment | Bedeutung |
|---|---|
| `int *p = NULL;` | p ist eine Zeigervariable auf ein int-Objekt.<br>Da keine Initialisierung vorgenommen ist, ist der Inhalt unbestimmt. |



# Zeiger

- Beispiele für Zugriffe:

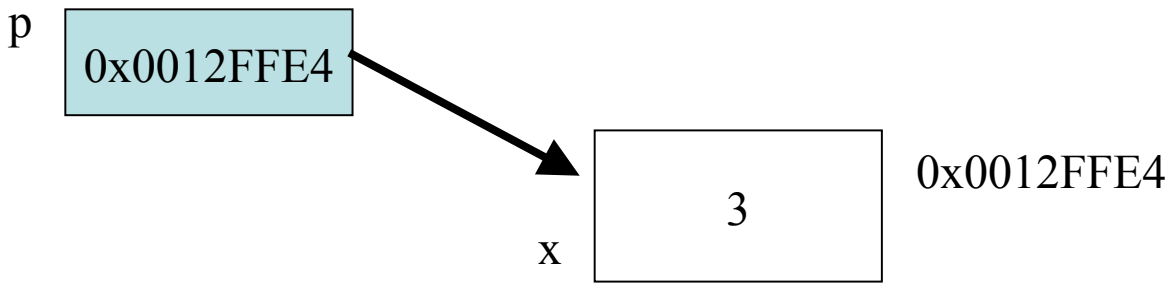
| C++-Fragment | Bedeutung |
|---|---|
| `int *p = NULL;`<br>`int x = 2;`<br>`p = &x;` | p wird die Adresse von x zugewiesen.<br>Man sagt p zeigt auf x. |



# Zeiger

- Beispiele für Zugriffe:

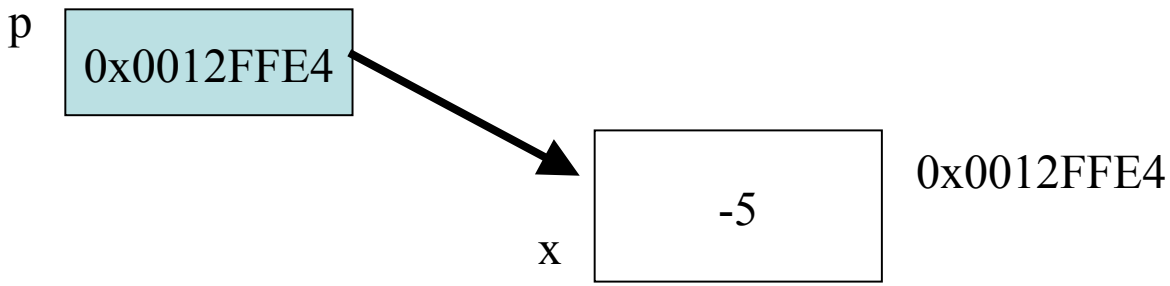
| C++-Fragment | Bedeutung |
|---|---|
| `int *p = NULL;`<br>`int x = 2;`<br>`p = &x;`<br>`x = *p + 1;` | p zeigt auf x.<br>Zuerst wird dereferenziert, d.h. der Wert von der Speicherstelle, auf die p zeigt wird ermittelt (x==2), dann der Wert erhöht (3) und der Variablen x zugewiesen. |



# Zeiger

- Beispiele für Zugriffe:

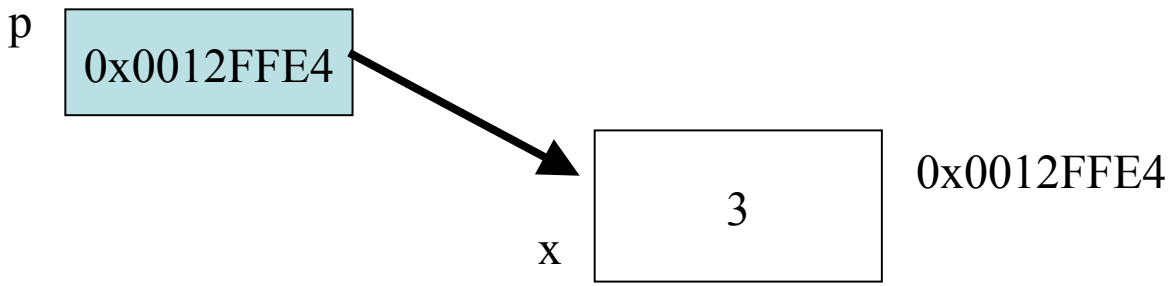
| C++-Fragment | Bedeutung |
|---|---|
| `int *p = NULL;`<br>`int x = 2;`<br>`p = &x;`<br>`*p = -5;` | p zeigt auf x.<br>Zuerst wird dereferenziert, d.h. der Wert von der Speicherstelle, auf die p zeigt wird ermittelt (x==2), dann wird der Wert dieser Speicherstelle auf -5 gesetzt. |



# Zeiger

- Beispiele für Zugriffe:

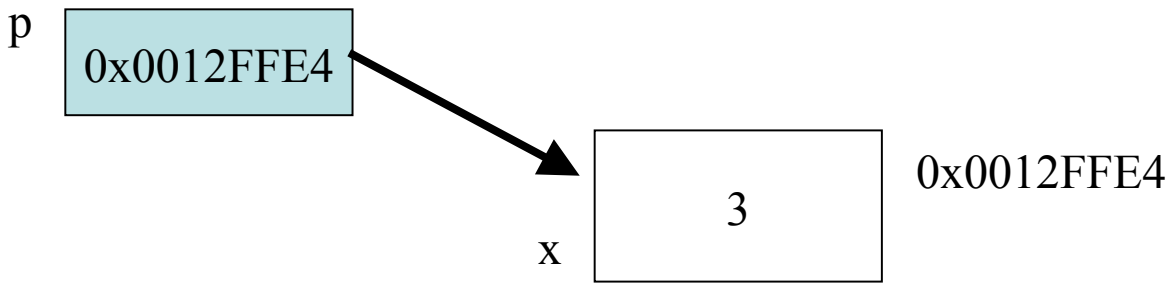
| C++-Fragment | Bedeutung |
|---|---|
| `int *p = NULL;`<br>`int x = 2;`<br>`p = &x;`<br>`*p += 1;` | p zeigt auf x.<br>Zuerst wird dereferenziert, d.h. der Wert von der Speicherstelle, auf die p zeigt wird ermittelt (x==2), dann wird der Wert dieser Speicherstelle um 1erhöht. |



# Zeiger

- Beispiele für Zugriffe:

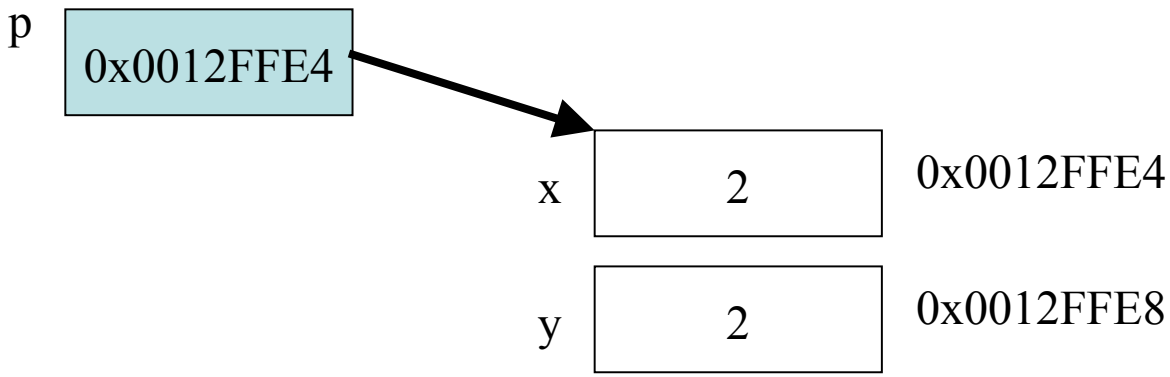
| C++-Fragment | Bedeutung |
|---|---|
| `int *p = NULL;`<br>`int x = 2;`<br>`p = &x;`<br>`(*p)++;` | p zeigt auf x.<br>Zuerst wird dereferenziert, d.h. der Wert von der Speicherstelle, auf die p zeigt wird ermittelt (x==2), dann wird der Wert dieser Speicherstelle um 1erhöht.<br>Die Klammern sind wegen Rang erforderlich. |



# Zeiger

- Beispiele für Zugriffe:

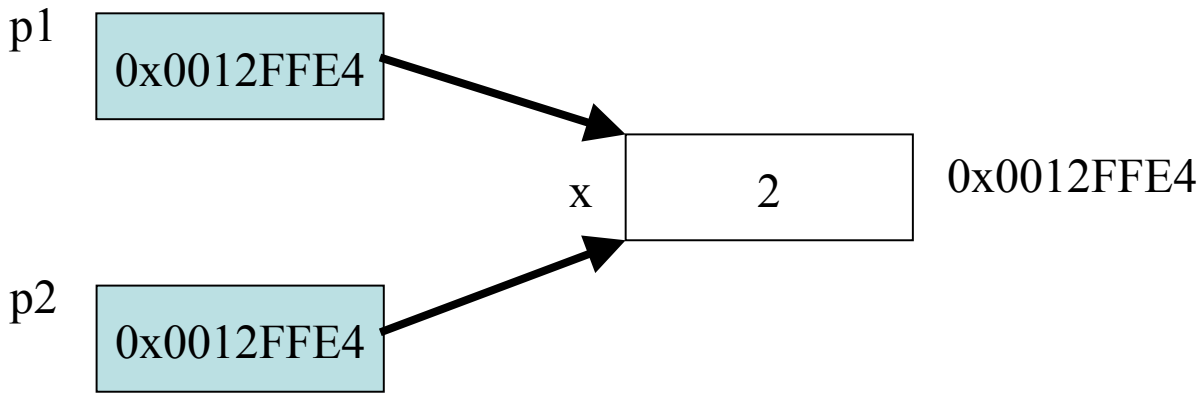
| C++-Fragment | Bedeutung |
|---|---|
| `int *p = NULL;`<br>`int x = 2, y;`<br>`p = &x;`<br>`y = *p;` | p zeigt auf x.<br>Durch y = *p wird derselbe Effekt erzielt wie durch y = x. |



# Zeiger

- Beispiele für Zugriffe:

| C++-Fragment | Bedeutung |
|---|---|
| `int *p1 = NULL;`<br>`int *p2 = NULL;`<br>`int x = 2;`<br>`p1 = &x;`<br>`p2 = p1;` | p1 zeigt auf x.<br>Der Variablen p2 wird der Wert der Variablen p2 zugewiesen.<br>Dadurch zeigt p2 auf dasselbe Objekt wie p1. |

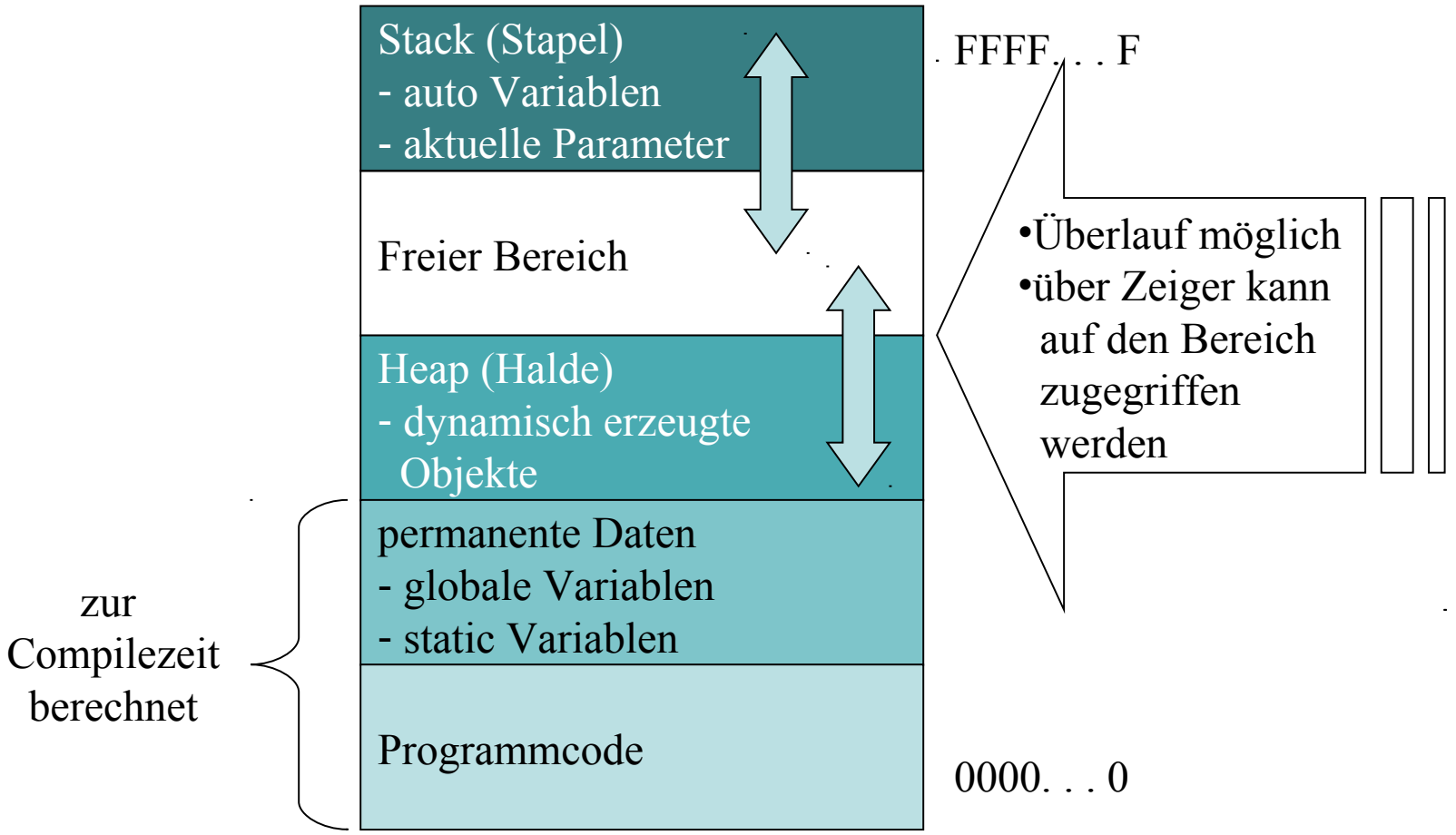


# Dynamische Datenobjekte

- Die bisher behandelten Datentypen waren statisch – der **Compiler** konnte den Speicherplatzbedarf ermitteln.
- In vielen Anwendungen ist der erforderliche Platzbedarf aber erst zur Laufzeit des Programms bekannt; z.B. wenn der Platz von der Anzahl der Benutzereingaben abhängt.
- Eine Lösung ist, ein sehr großes Array von Datenobjekten zu deklarieren und dort die Werte zu speichern.
- Besser wäre es man könnte Speicher zur Laufzeit, bei Bedarf anfordern (allokieren).

# Dynamische Datenobjekte

- Ein Programm wird vom Betriebssystem (je nach Betriebssystem unterschiedlich) im Speicher etwa wie folgt abgelegt.



# Dynamische Datenobjekte

- Der Zugriff auf Daten im Stack und den Bereich permanente Daten erfolgt über Namen (der Variablen).
- Der Zugriff auf Elemente des Heap erfolgt über Zeiger.

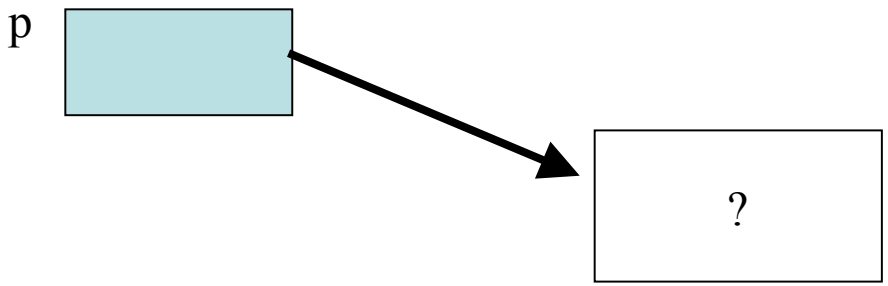
**Erzeugung von dynamischen Datenobjekten**

- Um ein **neues Element** im Heap ablegen zu können, verwendet man den **Operator new**.
- Der erforderliche Speicherplatz wird durch `new` in Abhängigkeit des Typs automatisch ermittelt.
- Beispiele für `new` Operator nächste Seite

# Dynamische Datenobjekte

- Beispiele für `new` Operator:

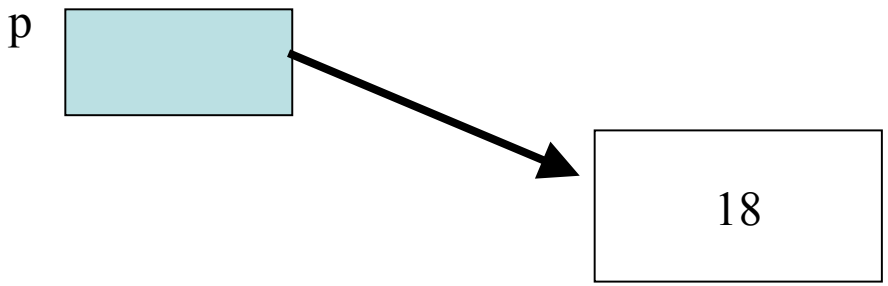
| **C++-Fragment** | **Bedeutung** |
|---|---|
| `int *p = NULL;`<br>`p = new int;` | Ein neues Element wird auf dem Heap angelegt.<br>Der Zugriff erfolgt über den Zeiger p.<br>Der Wert ist undefiniert.<br>Die Größe ist `sizeof(int)`. |



# Dynamische Datenobjekte

- Beispiele für `new` Operator:

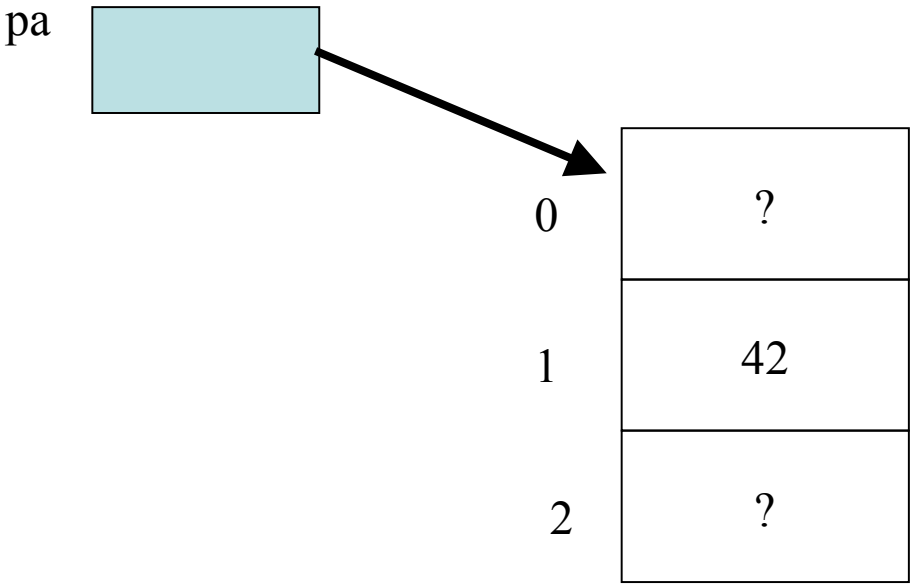
| C++-Fragment | Bedeutung |
|---|---|
| `int *p = NULL;`<br>`p = new int;`<br>`*p = 18;` | Ein neues Element wird auf dem Heap angelegt.<br>Der Zugriff erfolgt über den Zeiger p.<br>Der Wert 18 wird zugewiesen.<br>Die Größe ist `sizeof(int)`. |



# Dynamische Datenobjekte

- Beispiele für `new` Operator:

| C++-Fragment | Bedeutung |
|---|---|
| `int *pa = NULL;`<br>`pa = new int[3];`<br>`pa[1] = 42;` | Anlegen von 3 Int-Objekten.<br>Der Zugriff wie auf Arrays über Zeiger und Index.<br>Der Wert 42 wird dem 2. Element zugewiesen.<br>Die Größe ist 3 * `sizeof(int)`. |



# Erzeugung von dynamischen Datenobjekten

- Beispiel:

  Ein Programm, in dem ein **Vektor von Zeigern** definiert ist, bei dem jedes Vektorelement auf ein double-Objekt zeigt.

- ```
  #include <iostream>
  int  main()
  {
    double *pd[10]; // Vektor von Zeigern auf double;
    for (int i=0; i<10; i++)
    {
      cout << "Double: ";
      pd[i] = new double;      // Komponente i zeigt auf
                               // neues Element im Heap
      cin >> *pd[i];           // Wertzuweisung an neues
                               // Element über Zeiger
    }
    for (int i=9; i>=0; i-- )
    {
      cout << *pd[i] << " ";
    }
    cout << endl;
    return 0;
  }
  ```
  ```

# Erzeugung von dynamischen Datenobjekten

- Beispiel:

  Ein Programm mit einem Zeiger auf ein Array von double-Werten.

- ```
  #include <iostream>
  int main()
  {
    double *vd = new double[10]; // Zeiger auf Vektor
    for (int i =0 ; i < 10; i++)
    {
      cout << "Double: ";
      cin >> vd[i];            // Wertzuweisung an neues
                               // Element über Zeiger
    }
    for (int i = 9; i >= 0; i-- )
    {
      cout << vd[i] << " ";
    }
    cout << endl;
    return 0;
  }
  ```
  ```

# Freigabe von dynamischen Datenobjekten

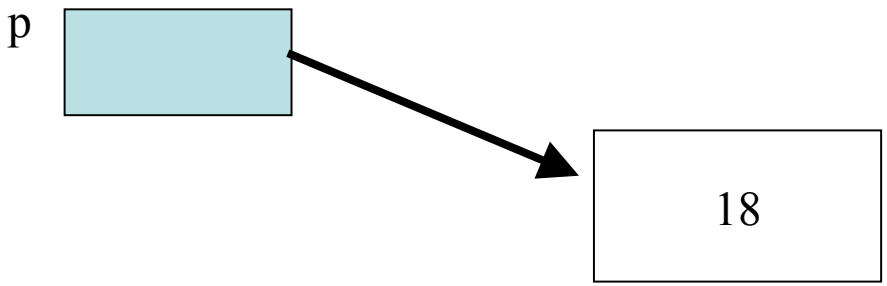
- Der Operator `delete` muss verwendet werden, um ein vorher mit `new` erzeugtes Objekt wieder freizugeben, damit der Speicherplatz erneut allokiert werden kann.

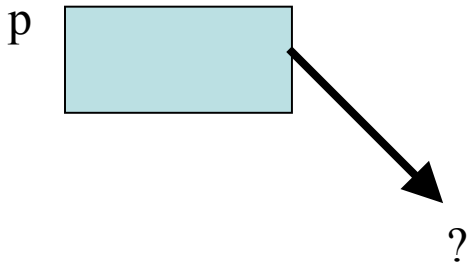
- Ein Array von Datenobjekten wird mit `delete [] pa;` freigegeben.

# Dynamische Datenobjekte

- Beispiele für `delete` Operator:

| C++-Fragment | Bedeutung |
|---|---|
| `int *p = NULL;`<br>`p = new int;`<br>`*p = 18;` | Ein neues Element wird auf dem Heap angelegt.<br>Der Zugriff erfolgt über den Zeiger p.<br>Der Wert 18 wird zugewiesen. |
| `int *p = NULL;`<br>`p = new int;`<br>`*p = 18;`<br>`delete p;` | Freigabe, auf das Element kann man nicht mehr zugreifen. |



# Freigabe von dynamischen Datenobjekten

- Einige Besonderheiten sind bzgl. `delete` zu beachten:
  - Dynamisch allokierter Speicher, auf den kein Zeiger mehr zeigt, ist **verloren**!
  - C++ **erkennt nicht**, ob ein Speicher noch benutzt wird (wegen der Laufzeiteffizienz).
    [dies ist in Java anders: die JVM überwacht den Heap und führt automatische ein Garbage Collection durch].
  - `delete` darf ausschließlich auf Objekte angewendet werden, die mit `new` erzeugt wurden.
  - `delete` auf einen NULL Zeiger ist wirkungslos.
  - mit `new` erzeugte Datenobjekte unterliegen nicht den Gültigkeitsregeln für Variablen.
    Sie existieren solange, bis mit `delete` gelöscht werden.

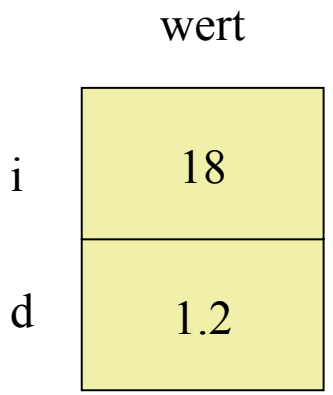
# Rekursive dynamischen Strukturen

- Zeiger sind nicht nur auf einfache Objekte möglich.
- Ein Zeiger kann auch auf eine Struktur zeigen.
- Da ein Element der Struktur selbst wieder ein Zeiger sein kann, entsteht eine rekursive Datenstruktur.
- Wenn ein Zeiger auf eine Struktur zeigt, kann eine Komponente selektiert werden durch Verwendung des Selektion- und des Dereferenzierungsoperators.

# Dynamische Datenobjekte

- Beispiele für dynamische Strukturen:

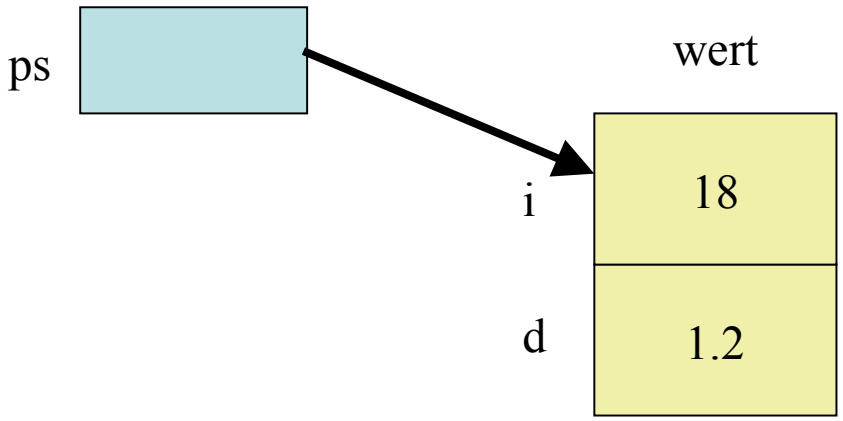
| C-Fragment | Bedeutung |
|---|---|
| `struct tupel {`<br>`  int i;`<br>`  double d;`<br>`} wert;`<br>`wert.i = 18;`<br>`wert.d = 1.2;` | Definition einer Variable wert, die eine Struktur von einem `int` und einem `double` ist. |



# Dynamische Datenobjekte

- Beispiele für dynamische Strukturen:

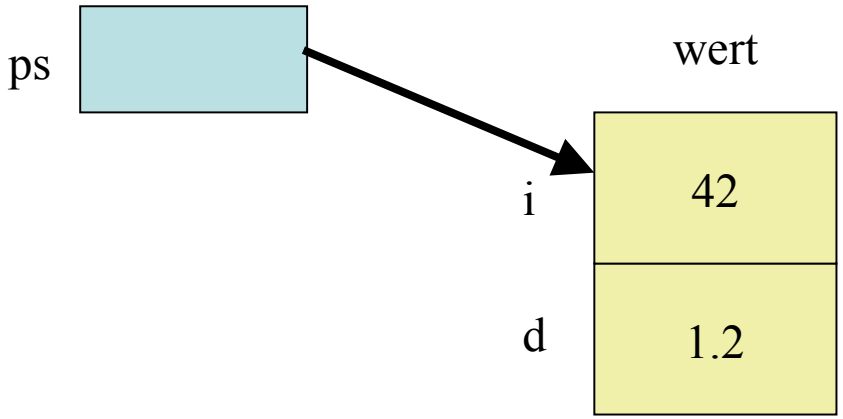
| **C-Fragment** | **Bedeutung** |
|---|---|
| `struct tupel *ps;`<br>`ps = &wert;` | Zeiger auf Struktur `tupel`;<br>`ps` zeigt auf Variable `wert`. |



# Dynamische Datenobjekte

- Beispiele für dynamische Strukturen:

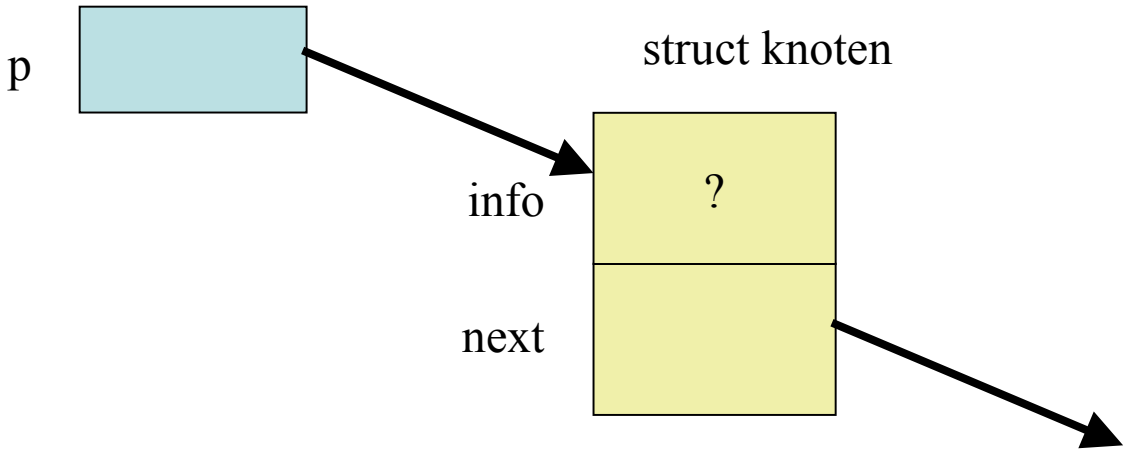
| **C-Fragment** | **Bedeutung** |
|---|---|
| `(*ps).i = 42;`<br><br>`cout << ps->i;` | Komponente i der Struktur selektieren, auf die ps zeigt und den Wert 42 zuweisen.<br>Kurzschreibweise für (*ps).i ist:<br>**ps->i** |



# Dynamische Datenobjekte

- Beispiele für rekursive dynamische Strukturen:

| **C++-Fragment** | **Bedeutung** |
|---|---|
| `struct knoten {`<br>`  int info;`<br>`  struct knoten *next;`<br>`};`<br>`knoten *p = new knoten;` | Definition einer Struktur, die auf sich selbst zeigt (next-pointer).<br>Neuer Zeiger `p` auf Struktur `knoten`. |



# Dynamische Datenobjekte

- Hier ist der Rang der Operatoren wichtig:

  - `++ps->i;`
    Entspricht ++(ps->i) also wird die Komponente i inkrementiert.

  - `(++ps)->i;`
    Inkrementiert ps, zeigt also auf anderes Objekt, dann wird die Komponente i selektiert!

# Zusammenfassung: Dynamische Datenobjekte

- Um ein **neues Element** im Heap ablegen zu können, verwendet man den **Operator new** (in C verwendet man malloc).
- Der erforderliche Speicherplatz wird durch `new` in Abhängigkeit des Typs automatisch ermittelt.
- Beispiele:

```
int *zeiger1;
zeiger1 = new int;
zeiger1 = NULL;

double *zeiger2 = NULL;
zeiger2 = new double[10];
```

# Zusammenfassung: Dynamische Datenobjekte

- Ein Zeiger kann auch auf eine Struktur zeigen.
- Da ein Element der Struktur selbst wieder ein Zeiger sein kann, entsteht eine rekursive Datenstruktur.
- Beispiel:

```
struct student
{
  string nachname;
  string vorname;
  long matrikelnummer;
  student *next;
};

student *zeigerAufListe = new student;
zeigerAufListe->nachname = "Seeber\0";
zeigerAufListe->vorname  = "Dirk\0";
(*zeigerAufListe).matrikelnummer = 123456;
zeigerAufListe->next = NULL;
```

# Zusammenfassung: Dynamische Datenobjekte

- Der Operator `delete` (in C verwendet man `free`) kann verwendet werden, um ein vorher mit `new` erzeugtes Objekt wieder freizugeben, **damit der Speicherplatz erneut allokiert werden kann**.
- Beispiele:

```
int *zeiger1;
zeiger1 = new int;
delete zeiger1;

double *zeiger2 = NULL;
zeiger2 = new double[10];
delete [] zeiger2;
```

# Zusammenfassung: Dynamische Datenobjekte

- Beispiel (Fortsetzung):

```
struct student
{
  string nachname;
  string vorname;
  long   matrikelnummer;
  student *next;
};

student *zeigerAufListe = new student;
zeigerAufListe->nachname = "Seeber\0";
zeigerAufListe->vorname = "Dirk\0";
zeigerAufListe->matrikelnummer = 123456;
zeigerAufListe->next = NULL;

delete zeigerAufListe;
```

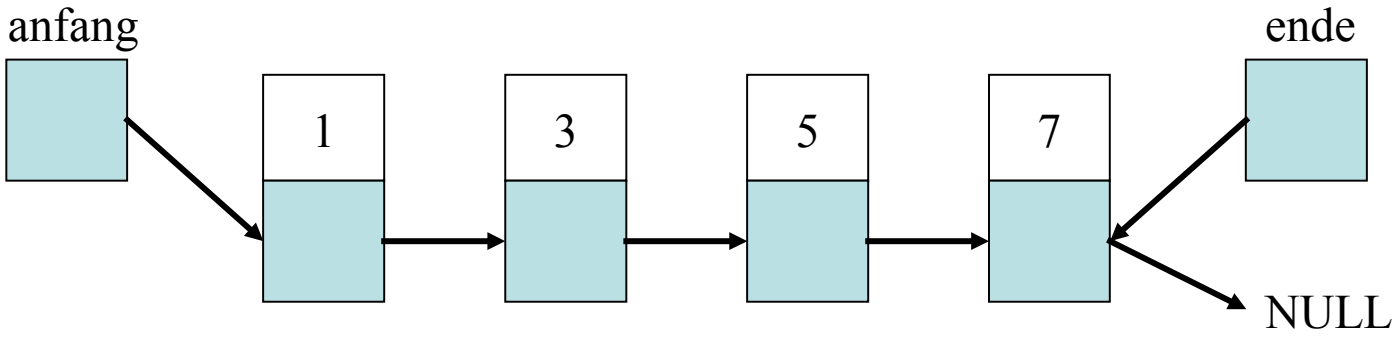
# Verkettete Listen

- Als erstes Anwendungsbeispiel werden **verkettete Listen** demonstriert.
- Eine verkette Liste ist eine rekursive dynamische Struktur, bei dem ein Strukturelement auf ein Datenobjekt des gleichen Typs zeigt.
- Das heißt ein Knoten hat die Form:

```
struct knoten
{
  int info;
  knoten *next;
};
```

# Verkettete Listen

- Damit lassen sich dann Listenelemente verketten:

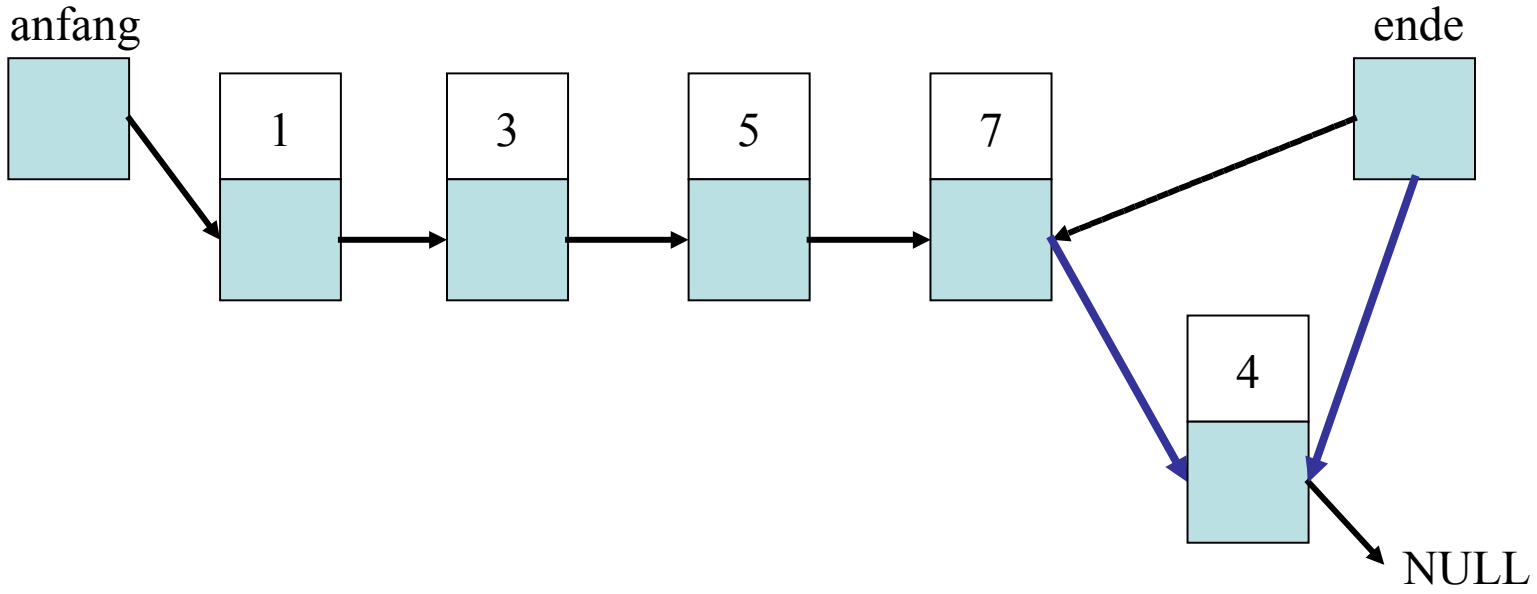


# Verkettete Listen - Operationen

- Folgende Operationen werden nun auf der verketteten Liste implementiert:
  - Ein **neues Element** mit info-Wert `neuWert` soll immer am **Ende** der Liste eingefügt werden.
    ```
    void einfuegen( int neuWert, knoten **anfang, knoten **ende );
    ```
  - Die Liste wird vom Anfang an bis zum Ende durchlaufen und der info-Teil jeden Knotens wird ausgegeben.
    ```
    void ausgeben( knoten *anfang );
    ```
  - Die Liste soll gelöscht werden.
    ```
    void loeschen( knoten *anfang );
    ```
  - Das erste Element mit info-Wert `suchWert` soll gesucht werden. Wenn kein solches Element in der Liste ist, soll das Ergebnis `NULL` sein.
    ```
    knoten * suchen( int suchElement, knoten *anfang );
    ```
  - Die Struktur wird „global“ definiert.

# Verkettete Listen - Einfügen

- Neues Element am Ende einfügen:

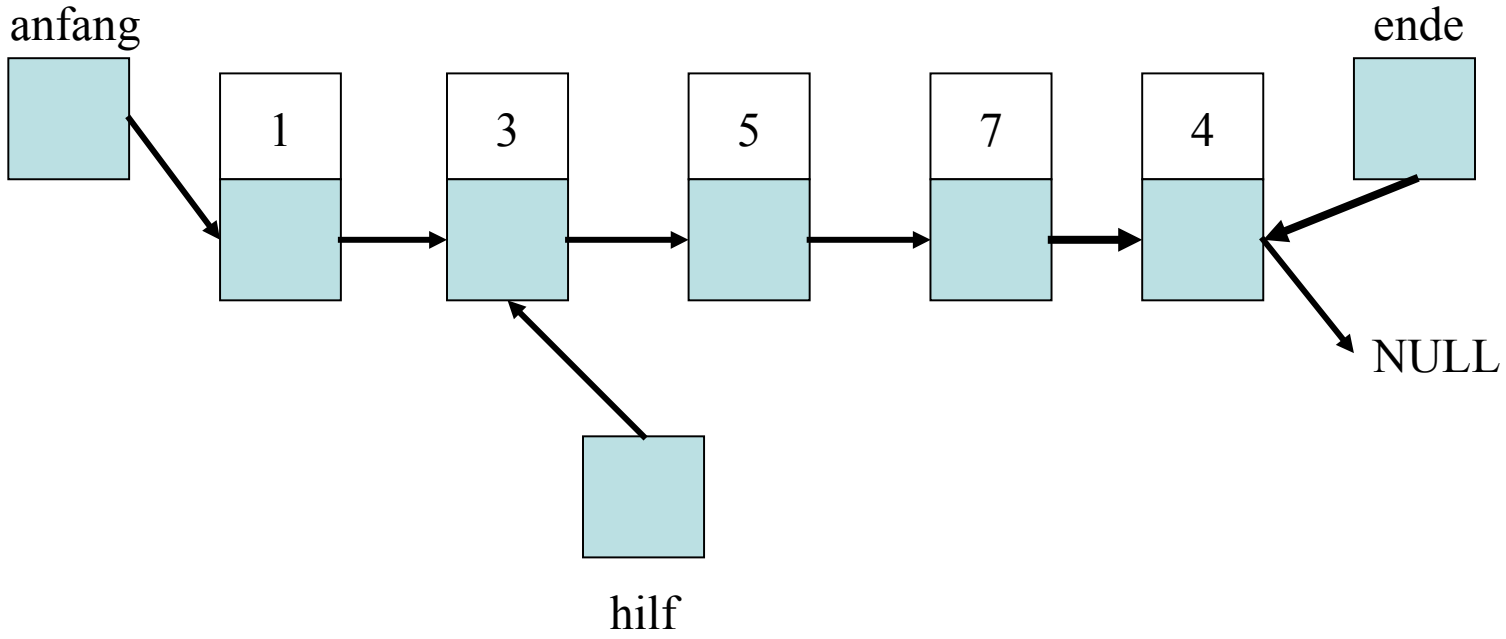


# Verkettete Listen - Einfügen

```
void einfuegen ( int x, ... )        // Einfuegen am Ende
{                                    // der Liste
  struct knoten *p = new knoten;     // erzeuge neuen Knoten
  p->info = x;                       // Infowert wird x
  p->next = NULL;                    // next-Wert wird mit
                                     // NULL initialisiert
  if ( NULL == *anfang ) {           // leere Liste
    *anfang = p;
    *ende = p;
  }
  else {                             // Liste nicht leer
    (*ende)->next = p;               // Einfuegen am Ende
    *ende = p;                       // neues Ende
  }
}
```

# Verkettete Listen - Ausgeben

- Liste von Anfang bis Ende ausgeben:

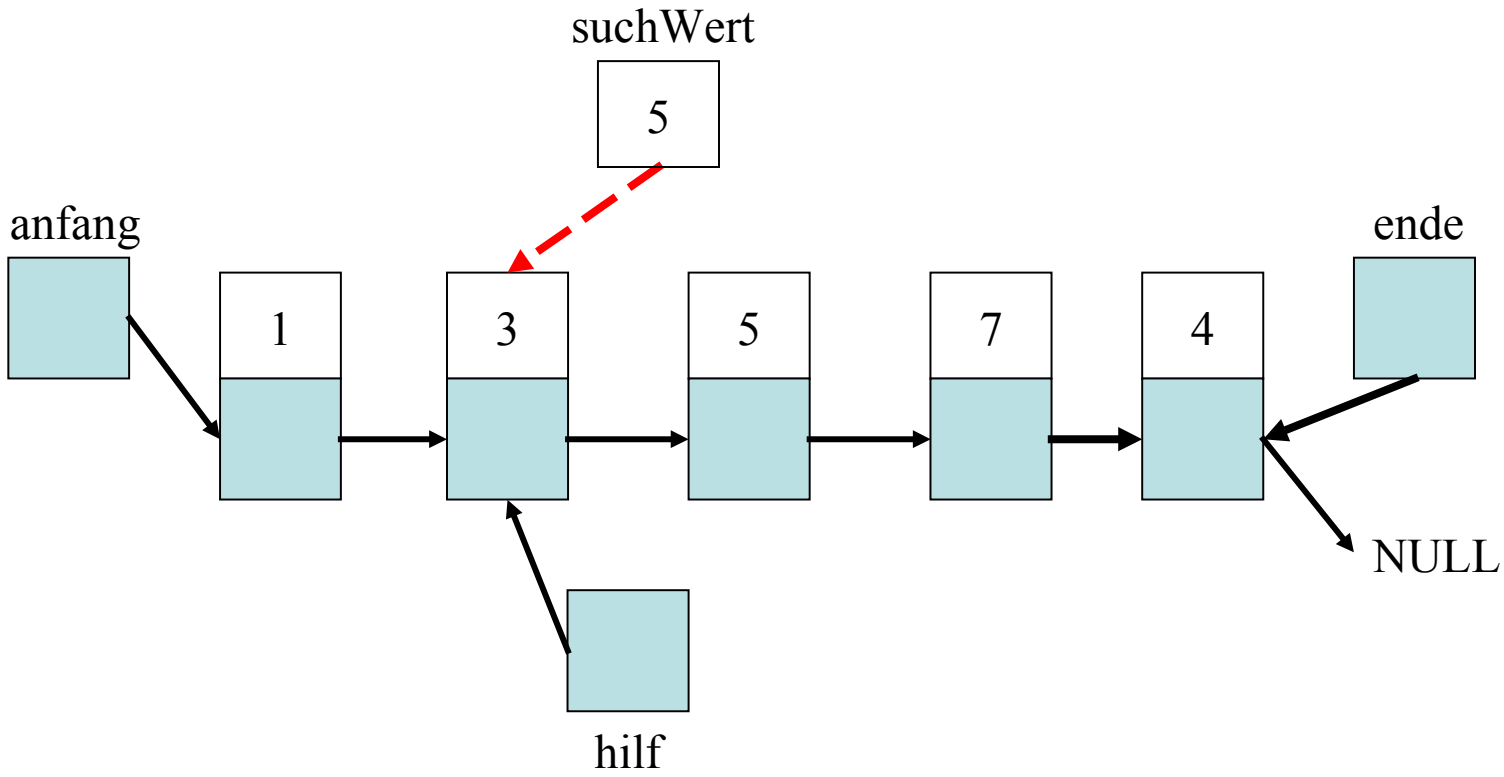


# Verkettete Listen - Ausgeben

```
void ausgeben (knoten *anfang)    // Ausgeben von Anfang
{                                 // bis Ende
  struct knoten *hilf = anfang;   // p zeigt auf aktuellen
                                  // Knoten
  cout << "Liste: ";
  while ( NULL != hilf)
  {
    cout << hilf->info << " ";    // Infoteil des aktuellen
                                  // Knoten ausgeben
    hilf = hilf->next;            // p auf Folgeelement setzen
  }
  cout << endl;
}
```

# Verkettete Listen - Suchen

- Element in der Liste suchen:

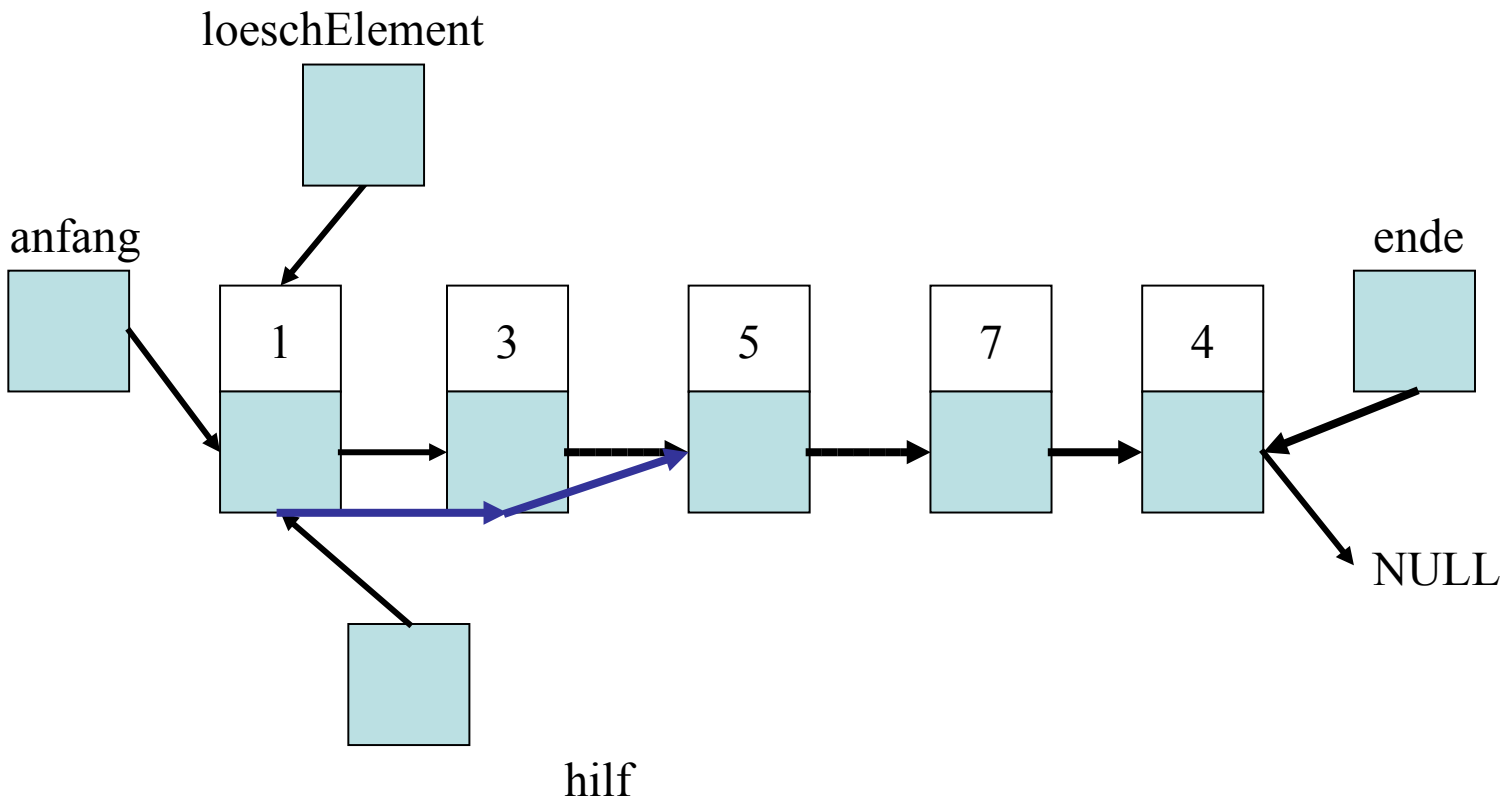


# Verkettete Listen - Suchen

```
struct knoten *suchen (int x, ...) // sequentielles suchen
{
  struct knoten *p = anfang;       // p zeigt auf aktuellen
                                   // Knoten
  while ( NULL != p )
  {
    if ( p->info == x )            // Wert gefunden
      return p;                    // Knoten ausgeben
    else
      p = p->next;                 // p auf Folgeelement setzen
  }
  return NULL;                     // nicht gefunden
}
```

# Verkettete Listen - Löschen

- Element aus der Liste entfernen:



# Verkettete Listen - Löschen

```
void loeschen ( struct knoten *p, ... )
{
  struct knoten *hilf = anfang;  // hilf zeigt auf
                                 // aktuellen Knoten
  struct knoten *loesch = NULL;

  while ( NULL != hilf )
  {
      loesch = hilf;
      hilf = hilf->next;   // hilf auf Folgeelement setzen
      delete loesch;
  }

}
```

# Verkettete Listen - Hauptprogramm

```
int main()
{
  struct knoten *anfang = NULL;                // Leere Liste
  struct knoten *ende = NULL;
  int eingabe = 1;
  while (eingabe > 0)
  {
    cout << "Integer (Abbruch mit negativem Wert): ";
    cin >> eingabe;
    einfuegen(eingabe, …, … );
  }
  ausgeben( … );
  cout << "Integer, der gesucht werden soll: ";
  cin >> eingabe;
  p = suchen(eingabe, … );
  if ( NULL != p )
    cout << eingabe << "in Liste vorhanden" << endl;
  else
    cout << eingabe << "nicht in Liste vorhanden" << endl;
  ausgeben( … );
  loeschen( … );
  return 0;
}
```

# Typedef

- In C gibt es kein Datentyp `string` oder `bool`.
- Diese Datentypen kann der Programmierer selbst definieren. Es existiert die Möglichkeit, Namen zu definieren. Mit
  `typedef` *`bestehenderTyp neuerName`* `;`
  kann dem bestehenden Typ ein neuer Name gegeben werden.
- Es wird - eigentlich - **kein** neuer Typ definiert, sondern nur ein neuer Name eingeführt.

# Typedef

- Beispiele:

```
#include <iostream>
using namespace std;

typedef int BOOL;
#define TRUE  1
#define FALSE 0

int main()
{
  BOOL bool_var = TRUE;
  cout << "bool_var = " << bool_var << endl;

  bool_var = FALSE;
  cout << "bool_var = " << bool_var << endl;

  return 0;
}
```

# Typedef

- Beispiele:

```
#include <iostream>
using namespace std;

typedef double real;

int main()
{
  real x = 1.2;
  cout << "x = " << x << endl;
  return 0;
}
```

oder

```
typedef enum wie {nicht, aufsteigend, absteigend};
```