Informatik 8bi Schuljahr 2019/2020 als PDF exportieren

Informatik 8. Klasse - Schuljahr 2019/20

Lehrinhalte

- Lehrplaninhalte

Remote-Zugriff auf Schulserver

Kapitel

- 1) Datenstrukturen
- 2) Datenbanken
- 3) PHP & MySQL
- 4) Unity 3D

Leistungsbeurteilung

- **Schularbeiten (SA)**
  - 2x SA (2h und 3h)
- **Mitarbeit (MA)**
  - Aktive Mitarbeit im Unterricht (aMA)
  - Mündliche Stundenwiederholungen (mMA)
  - Schriftliche Stundenwiederholungen (sMA)
- **Praktische Arbeiten (PA)**
  - 1x praktischer Arbeitsauftrag pro Woche

- Aktueller Leistungsstand

Stoff für die 1. Schularbeit in Informatik – 8BI – Dezember?? (2h)

Stoff für die 2. Schularbeit in Informatik – 8BI – März?? (3h)

Themengebiete RDP

Was wird in der 8. Klasse gemacht?

7. Semester

```
    8. Klasse (3 Stunden,  eine 2- oder 3-stündige Schularbeit)
```

Sicherung der Nachhaltigkeit

- Notwendiges Vorwissens für die Kompetenzbereiche dieses Moduls wiederholen und aktivieren
- Grundlagen für die Kompetenzbereiche dieses Moduls ergänzen und bereitstellen

Gesellschaftliche Aspekte der Informationstechnologie

Berufliche Perspektiven

- Informatikberufe und Einsatzmöglichkeiten der Informatik in verschiedenen Berufsfeldern benennen und einschätzen können.

Verantwortung, Datenschutz und Datensicherheit

- Die Entwicklung der Informatik beschreiben und bewerten können.
- Die Bedeutung von Informatik in der Gesellschaft beschreiben, die Auswirkungen auf die Einzelnen und die Gesellschaft einschätzen und Vor- und Nachteile an konkreten Beispielen abwägen können.
- Maßnahmen und rechtliche Grundlagen im Zusammenhang mit Datensicherheit, Datenschutz und Urheberrecht kennen und anwenden können.

Informatiksysteme – Hardware, Betriebssysteme und Vernetzung

Technische Grundlagen und Funktionsweisen (Hardware)

- Aktualisierungen im Zusammenhang mit der Hardware kennen

Betriebssysteme (Windows, Linux, MacOS, iOS, Android)

- Aktualisierungen im Zusammenhang mit Betriebssystemen kennen

Mensch-Maschine-Schnittstelle

- Maßnahmen für einen barrierefreien zu Zugang Informatik-Systemen angeben können

Algorithmik und Programmierung

Algorithmen und Datenstrukturen

- Algorithmen erklären, entwerfen, darstellen können.
- Datenstruktur Bäume kennen und einsetzen können
- Rekursionen kennen und einsetzen können
- Dynamische Programmierung kennen
- Algorithmen mit Bäumen erstellen können
- Algorithmen mit Rekursionen erstellen können

Programmierung (Objektorientierte visuelle Programmiersprache)

- Algorithmen in einer Programmiersprache implementieren können
- Datenbankanwendungen programmieren können
- Programme mit Bäume erstellen können
- Rekursive Algorithmen erstellen können

Angewandte Informatik, Datenbanksysteme und Internet

Datenmodelle und Datenbanksysteme

- Einen Webserver konfigurieren können
- Internetdienste (Mail-Server, Web-Server, FTP-Server) in ihrer Funktionsweise verstehen und einsetzen können

Web-Techniken (Content-Management-Systeme)

- Content-Management-Systeme installieren können
- Rechte bei Content-Management-Systemen vergeben können
- Oberfläche bei Content-Management-Systemen einstellen und anpassen können
- Die Funktionsweise durch Plugins und Module erweitern können

8. Semester

```
    8. Klasse (3 Stunden,  eine 3- oder 4-stündige Schularbeit)
```

Sicherung der Nachhaltigkeit

Wiederholen, Vertiefen von Fähigkeiten und Vernetzen von Inhalten, um einen umfassenden Überblick über die Zusammenhänge unterschiedlicher informatischer Teilgebiete zu gewinnen.

Inhalt und Umfang der Klausurarbeit im Prüfungsgebiet Informatik

(1) Im Rahmen der Klausurarbeit im Prüfungsgebiet „Informatik“ ist den Prüfungskandidatinnen und Prüfungskandidaten eine Aufgabenstellung mit drei bis fünf voneinander unabhängigen Aufgaben, die in Teilaufgaben gegliedert sein können, aus unterschiedlichen Kompetenzbereichen – Gesellschaftliche Aspekte der Informationstechnologie, Informatiksysteme, Algorithmik und Programmieren sowie Angewandte Informatik, Datenbank und Internet - mit ausgewogenen Anforderungen schriftlich vorzulegen. Mindestens eine Aufgabe hat anwendungsorientierten Charakter zu haben. Für die Bearbeitung zumindest einer Aufgabe muss Computertechnologie eingesetzt werden. (2) Die Arbeitszeit hat 270 Minuten zu betragen.

C++

Zeiger (Pointer)

Zeiger (engl. pointers) sind Variablen, die als Wert die Speicheradresse einer anderen Variable enthalten.

Jede Variable wird in CPP an einer bestimmten Position im Hauptspeicher abgelegt. Diese Position nennt man Speicheradresse (engl. memory address). CPP bietet die Möglichkeit, die Adresse jeder Variable zu ermitteln. Solange eine Variable gültig ist, bleibt sie an ein und derselben Stelle im Speicher.

Am einfachsten vergegenwärtigt man sich dieses Konzept anhand der globalen Variablen. Diese werden außerhalb aller Funktionen und Klassen deklariert und sind überall gültig. Auf sie kann man von jeder Klasse und jeder Funktion aus zugreifen. Über globale Variablen ist bereits zur Kompilierzeit bekannt, wo sie sich innerhalb des Speichers befinden (also kennt das Programm ihre Adresse).

Zeiger sind nichts anderes als normale Variablen. Sie werden deklariert (und definiert), besitzen einen Gültigkeitsbereich, eine Adresse und einen Wert. Dieser Wert, der Inhalt der Zeigervariable, ist aber nicht wie in unseren bisherigen Beispielen eine Zahl, sondern die Adresse einer anderen Variable oder eines Speicherbereichs. Bei der Deklaration einer Zeigervariable wird der Typ der Variable festgelegt, auf den sie verweisen soll.

```
#include <iostream>

int main() {
     int   Wert;      // eine int-Variable
     int *pWert;      // eine Zeigervariable, zeigt auf einen int
     int *pZahl;      // ein weiterer "Zeiger auf int"

     Wert = 10;       // Zuweisung eines Wertes an eine int-Variable

     pWert = &Wert;   // Adressoperator '&' liefert die Adresse einer
Variable
     pZahl = pWert;   // pZahl und pWert zeigen jetzt auf dieselbe Variable
```

Beispielhafte Speicherbelegung des Programms im Hauptspeicher:

Datentyp	Variable	Adresse	Wert
int	Wert	0x0001	10
int *	pWert	0x0005	0x0001
int *	pZahl	0x0009	0x0001
...	...	.	.
...	...	.	.

Der Adressoperator & kann auf jede Variable angewandt werden und liefert deren Adresse, die man einer (dem Variablentyp entsprechenden) Zeigervariablen zuweisen kann. Wie im Beispiel gezeigt, können Zeiger gleichen Typs einander zugewiesen werden. Zeiger verschiedenen Typs bedürfen einer Typumwandlung. Die Zeigervariablen pWert und pZahl sind an verschiedenen Stellen im Speicher abgelegt, nur die Inhalte sind gleich.

Wollen Sie auf den Wert zugreifen, der sich hinter der im Zeiger gespeicherten Adresse verbirgt, so verwenden Sie den Dereferenzierungsoperator *.

```
  *pWert += 5;
  *pZahl += 8;

   std::cout << "Wert = " << Wert << std::endl;
```

Beispielhafte Speicherbelegung des Programms im Hauptspeicher:

Datentyp	Variable	Adresse	Wert
int	Wert	0x0001	10 –> 15 –> 23
int *	pWert	0x0005	0x0001
int *	pZahl	0x0009	0x0001
...	...	.	.
...	...	.	.

Ausgabe:

```
Wert = 23
```

Man nennt das den Zeiger dereferenzieren. Im Beispiel erhalten Sie die Ausgabe Wert = 23, denn pWert und pZahl verweisen ja beide auf die Variable Wert.

Um es noch einmal hervorzuheben: Zeiger auf Integer (int) sind selbst keine Integer. Den Versuch, einer Zeigervariablen eine Zahl zuzuweisen, beantwortet der Compiler mit einer Fehlermeldung oder mindestens einer Warnung. Hier gibt es nur eine Ausnahme: die Zahl 0 darf jedem beliebigen Zeiger zugewiesen werden. Ein solcher Nullzeiger zeigt nirgendwohin. Der Versuch, ihn zu dereferenzieren, führt zu einem Laufzeitfehler.

Bespiel Zeigerübung

```
int main()
{
        int zahl=10;
        int *z=NULL;
        int **zz=NULL;


        cout<< zahl<<endl;            //Wert wird ausgegeben
        cout<<&zahl<<endl;            //adresse wird ausgegeben
        cout<<&zahl+1<<endl;        //adresse cpn zahl+1
        cout<< zahl +1<<endl;        //11


        z=&zahl;
```

```
        cout<< *z<< endl;         //10
        cout<< z<< endl;         //Adresse von zahl= Wert von z
        cout<< &z<< endl;         //Adresse von Zeiger z

        zz=&z;                     //Adresse von Zeiger z wird in Zeiger zz

        cout<< *zz<<endl;         //Wert von z= Adresse von zahl    //&zahl
//z
        cout<< **zz<<endl;         //Wert von zahl                    //zahl
        cout<< zz<<endl;         //Adresse von z =Wert von zz    //&z
//zz
        cout<< &zz<<endl;         //Adresse von Zeiger zz            //&zz

  getch();
  return 0;
}
```

Verschiedene Konventionen bei der Definition von Zeigern

Bei der Definition einer Zeigervariablen muss das Zeichen * nicht unmittelbar auf den Datentyp folgen. Die folgenden vier Definitionen sind gleichwertig:

```
int* i;  // Whitespace (z.B. ein Leerzeichen) nach *
int *i;  // Whitespace vor *
int * i; // Whitespace vor * und nach *
int*i;   // Kein whitespace vor * und nach *
```

Versuchen wir nun, diese vier Definitionen nach demselben Schema wie eine Definition von „gewöhnlichen“ Variablen zu interpretieren. Bei einer solchen Definition bedeutet

```
T v;
```

dass eine Variable v definiert wird, die den Datentyp T hat. Für die vier gleichwertigen Definitionen ergeben sich verschiedene Interpretationen, die als Kommentar angegeben sind:

```
int* i;   // Definition der Variablen i des Datentyps int*
int *i;   // Irreführend: Es wird kein "*i" definiert,
          // obwohl die dereferenzierte Variable *i heißt.
int * i;  // Datentyp int oder int* oder was?
int*i;    // Datentyp int oder int* oder was?
```

Offensichtlich passen nur die ersten beiden in dieses Schema. Die erste führt dabei zu einer richtigen und die zweite zu einer falschen Interpretation.

Allerdings passt die erste Schreibweise nur bei der Definitionen einer einzelnen Zeigervariablen in dieses Schema, da sich der * bei einer Definition nur auf die Variable unmittelbar rechts vom *

bezieht:

```
int* i,j,k; // definiert int* i, int j, int k
            // und nicht: int* j, int* k
```

Zur Vermeidung solcher Missverständnisse sind zwei Schreibweisen verbreitet:

- C-Programmierer verwenden oft die zweite Schreibweise von oben, obwohl sie nicht in das Schema der Definition von „gewöhnlichen“ Variablen passt. Diese Schreibweise wird auch in Turbo CPP verwendet.

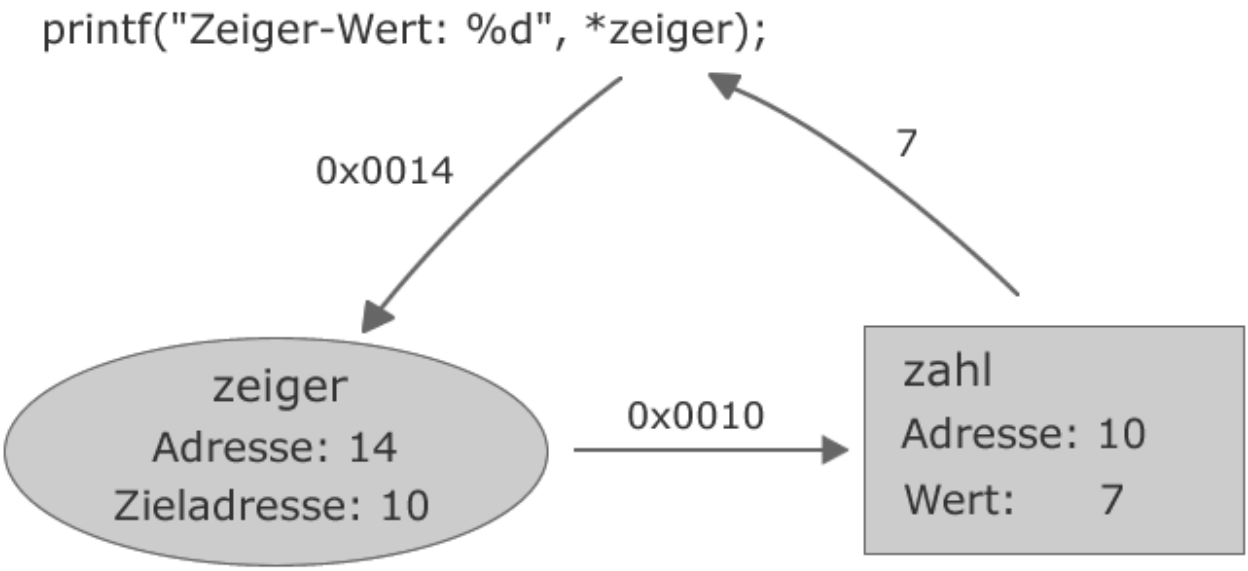
```
int *i,*j,*k; // definiert int* i, int* j, int* k
```

- Bjarne Stroustrup (einer der C bzw. CPP-Entwickler) empfiehlt, auf Mehrfachdefinitionen zu verzichten. Er schreibt den * wie in „int* pi;“ immer unmittelbar nach dem Datentyp.

Ein weiteres Beispiel....

```
int zahl = 7;
int *zeiger;
zeiger = &zahl;
printf("Zeiger-Wert: %d\n", *zeiger);
```

Ein **Zeiger repräsentiert eine Adresse** und nicht wie eine Variable einen Wert. Will man auf den Wert der Adresse zugreifen, auf die ein Zeiger zeigt, muss der Stern * vor den Namen gesetzt werden.



Zeiger auf Zeiger

Zeiger zeigen auf Adressen. Sie können nicht nur auf die Adressen von Variablen, sondern auch auf die Adressen von Zeigern verweisen. Dies erreicht man mit dem doppelten Stern-Operator * * .

```
int zahl=7;
```

```
int *zeiger = &zahl;
int **zeigerAufZeiger = &zeiger;

cout << "Wert von zeigerAufZeiger -> zeiger -> zahl:" << **zeigerAufZeiger;
```

Ausgabe

```
Wert von zeigerAufZeiger -> zeiger -> zahl: 7
```

Aufgabe 1.1.1

Gib nach jeden Programmierbefehl alle Adressen und Inhalte der einzelnen Variablen aus!

```
int a=2, b=5, *c=&a, *d=&b;

a = *c * *d;
*d -= 3;
b = a * b;
c = d;
b = 7;
a = *c + *d;
```

Aufgabe 1.1.2

Gib nach jeden Programmierbefehl alle Adressen und Inhalte der einzelnen Variablen aus!

```
int a=2, b=5, *c=&a, *d=&b;
int **zz=NULL;

a = *c + *d;
zz=&d;
**zz=*zz-10;
*c *= 3;
b = a * *c;
c = d;
a = 7-*d;
b = *c * *d;
*c = *c + **zz;
```

Aufgabe 1.1.3

```
char   *a=NULL, *b=NULL, *c=NULL;
char   d;

a = new char;
b = new char;

*a = 'S';

*b = 'T';

 c = b;

cout << *a << endl;
cout << *b << endl;
```

```
cout << *c << endl;

 d = 'U';

*c = 'G';

 b = &d;


cout << d << endl;
cout << *b << endl;

 d = 'H';

 b = a;

 a = c;

 c = &d;

*b = 'I';

cout << *c << *b << *a << d;
```

Struktur - eine zusammengesetzter Datentyp

Mit Arrays können Variablen gleichen Typs zusammengestellt werden. In der realen Welt gehören aber meist Daten unterschiedlichen Typs zusammen. So hat ein Auto einen Markennamen und eine Typbezeichnung, die als Zeichenkette unterzubringen ist. Dagegen eignet sich für Kilometerzahl und Leistung eher der Typ Integer. Für den Preis bietet sich der Typ float an. Bei bestimmten Autohändlern könnte auch double erforderlich sein. Alles zusammen beschreibt ein Auto.

Modell

Vielleicht werden Sie einwerfen, dass ein Auto noch mehr Bestandteile hat. Da gibt es Bremsscheiben, Turbolader und Scheibenwischer. Das ist in der Realität richtig. Ein Programm interessiert sich aber immer nur für bestimmte Eigenschaften, die der Programmierer mit dem Kunden zusammen festlegt. Unser Beispiel würde für einen kleinen Autohändler vielleicht schon reichen. Eine Autovermietung interessiert sich vielleicht überhaupt nicht für den Wert des Autos, aber möchte festhalten, ob es für Nichtraucher reserviert ist. Eine Werkstatt dagegen könnte sich tatsächlich für alle Teile interessieren. Ein Programm, das die Verteilung der Firmenfahrzeuge verwaltet, interessiert sich vielleicht nur für das Kennzeichen. Es entsteht also ein Modell eines Autos, das bestimmte Bestandteile enthält und andere vernachlässigt, je nachdem was das Programm benötigt. Bereits in C gab es für solche Zwecke die Struktur, die mehrere Variablen zu einer zusammenfasst. Das Schlüsselwort für die Bezeichnung solch zusammengesetzter Variablen lautet struct. Nach diesem Schlüsselwort folgt der Name des neuen Typen. In dem folgenden geschweiften Klammernblock werden die Bestandteile der neuen Struktur aufgezählt. Diese unterscheiden sich nicht von der bekannten Variablendefinition. Den Abschluss bildet ein Semikolon.

struct

Um ein Auto zu modellieren, wird ein neuer Variablentyp namens TAutoTyp geschaffen, der ein Verbund mehrerer Elemente ist.

```
struct TAutoTyp // Definiere den Typ
{
    char Marke[MaxMarke];
    char Modell[MaxModell];
    long km;
    int kW;
    float Preis;
};  // Hier vergisst man leicht das Semikolon!
```

Syntaxbeschreibung

Das Schlüsselwort struct leitet die Typdefinition ein. Es folgt der Name des neu geschaffenen Typs, hier TAutoTyp. In dem nachfolgenden geschweiften Klammerpaar werden alle Bestandteile der

Struktur nacheinander aufgeführt. Am Ende steht ein Semikolon, das man selbst als erfahrener Programmierer immer wieder einmal vergisst. Variablendefinition Damit haben wir den Datentyp TAutoTyp geschaffen. Er kann in vieler Hinsicht verwendet werden wie der Datentyp int. Sie können beispielsweise eine Variable von diesem Datentyp anlegen. Ja, Sie können sogar ein Array und einen Zeiger von diesem Datentyp definieren.

```
TAutoTyp MeinRostSammler; // Variable anlegen
TAutoTyp Fuhrpark[100];   // Array von Autos
TAutoTyp *ParkhausKarte;  // Zeiger auf ein Auto
```

Elementzugriff

Die Variable MeinRostSammler enthält nun alle Informationen, die in der Deklaration von TAutoTyp festgelegt sind. Um von der Variablen auf die Einzelteile zu kommen, wird an den Variablenname ein Punkt und daran der Name des Bestandteils gehängt.

```
// Auf die Details zugreifen
MeinRostSammler.km = 128000;
MeinRostSammler.kW = 25;
MeinRostSammler.Preis = 25000.00;
```

Zeigerzeichen

Wenn Sie über einen Zeiger auf ein Strukturelement zugreifen wollten, müssten Sie über den Stern referenzieren und dann über den Punkt auf das Element zugreifen. Da aber der Punkt vor dem Stern ausgewertet wird, müssen Sie eine Klammer um den Stern und den Zeigernamen legen.

```
TAutoTyp *ParkhausKarte = 0;      // Erst einmal keine Zuordnung
ParkhausKarte = &MeinRostSammler; // Nun zeigt sie auf ein Auto
(*ParkhausKarte).Preis = 12500;   // Preis für MeinRostSammler
```

Das mag zwar logisch sein, aber es ist weder elegant noch leicht zu merken. Zum Glück gibt es in C und C++ eine etwas hübschere Variante, über einen Zeiger auf Strukturelemente zuzugreifen. Dazu wird aus Minuszeichen und Größer-Zeichen ein Symbol zusammengesetzt, das an einen Pfeil erinnert.

```
ParkhausKarte->Preis = 12500;
```

L-Value

Strukturen sind L-Values. Sie können also auf der linken Seite einer Zuweisung stehen. Andere Strukturen des gleichen Typs können ihnen zugewiesen werden. Dabei wird die Quellvariable Bit für Bit der Zielvariable zugewiesen. TAutoTyp MeinNaechstesAuto, MeinTraumAuto; MeinNaechstesAuto = MeinTraumAuto;

Trotzdem die beiden Strukturvariablen nach dieser Operation ganz offensichtlich gleich sind, kann

man dies nicht einfach durch eine Anwendung des doppelten Gleichheitszeichen nachprüfen. Sie können bei Strukturen die Typdeklaration und die Variablendefinition zusammenfassen, indem der Name der Variablen direkt nach der geschweiften Klammer eingetragen wird.

```
struct // hier wird kein Typ namentlich festgelegt
{
    char Marke[MaxMarke];
    char Modell[MaxModell];
    long km;
    int kW;
    float Preis;
} MeinErstesAuto, MeinTraumAuto;
```

Hier werden im Beispiel die Variablen MeinErstesAuto und MeinTraumAuto gleich mit ihrer Struktur definiert. Werden auf diese Weise gleich Variablen dieser Struktur gebildet, muss ein Name für den Typ nicht unbedingt angegeben werden. Damit ist dann natürlich keine spätere Erzeugung von Variablen dieses Typs möglich. Initialisierung Auch Strukturen lassen sich initialisieren. Dazu werden wie bei den Arrays geschweifte Klammern verwendet. Auch hier werden die Werte durch Kommata getrennt.

```
TAutoTyp JB = {"Aston Martin", "DB5", 12000, 90, 12.95};
TAutoTyp GWB = {0};
```

Lineare Listen (=Einfach verkettete Listen)

Einfach **verkettete Listen** oder **linked lists** sind eine **fundamentale Datenstruktur**, die ich hier anhand von Code-Beispielen und Grafiken erklären will.

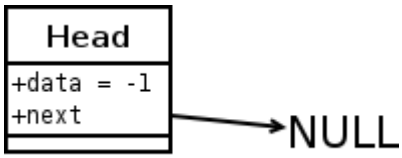
Einfach verkettete Listen zeichnen sich dadurch aus, dass man besonders einfach Elemente einfügen kann, wodurch sie sich besonders gut für Insertion Sort eignen.

Knoten

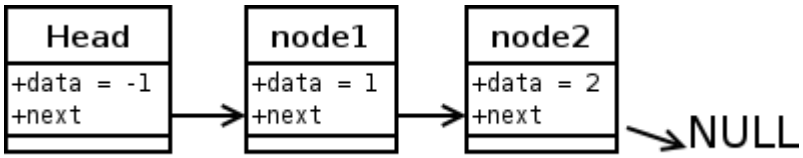
Eine einfach verkettete Liste besteht aus **Knoten, Englisch nodes**, die einen **Zeiger auf das nächste Element** und Daten beinhalten.

```
typedef struct listnode{

    int data;
    listnode *next;

};
```

Eine **leere Liste** besteht aus einem Kopf (Head) und nichts sonst:



Wenn man mehrere Elemente einfügt, sieht das so aus:



Eine einfach verkettete Liste mit einem Kopf und zwei Knoten.

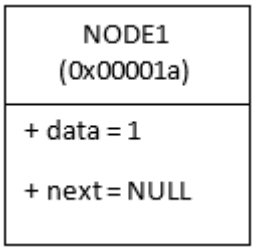
Knoten befüllen & einfügen

Wenn man einen Zeiger auf ein Element der Liste hat, ist es einfach, ein Element dahinter einzufügen.

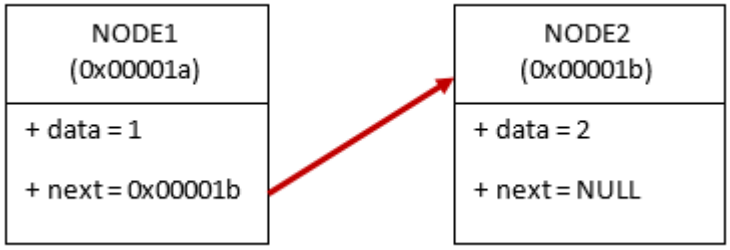
Dazu muss man den next-Zeiger der Liste auf das neue Element setzen, und den next-Zeiger des neuen Element auf den alten Wert des next-Zeigers der Liste:

```
    //1. Knoten erstellen
    listnode *node1;
```

```
    //1. Knoten befüllen
    node1->data=1;    //oder (*node1).data=1;
    node1->next=NULL;
```

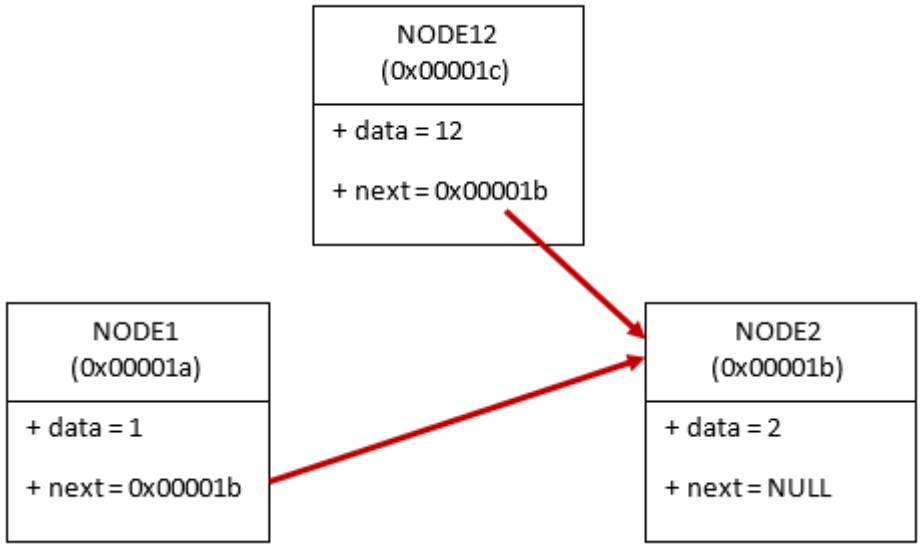



```
    //2. Knoten erstellen
    listnode *node2;
    node2->data=2;
    node2->next=NULL;
    //1. Knoten zeigt auf 2. Knoten
    node1->next=node2;
```

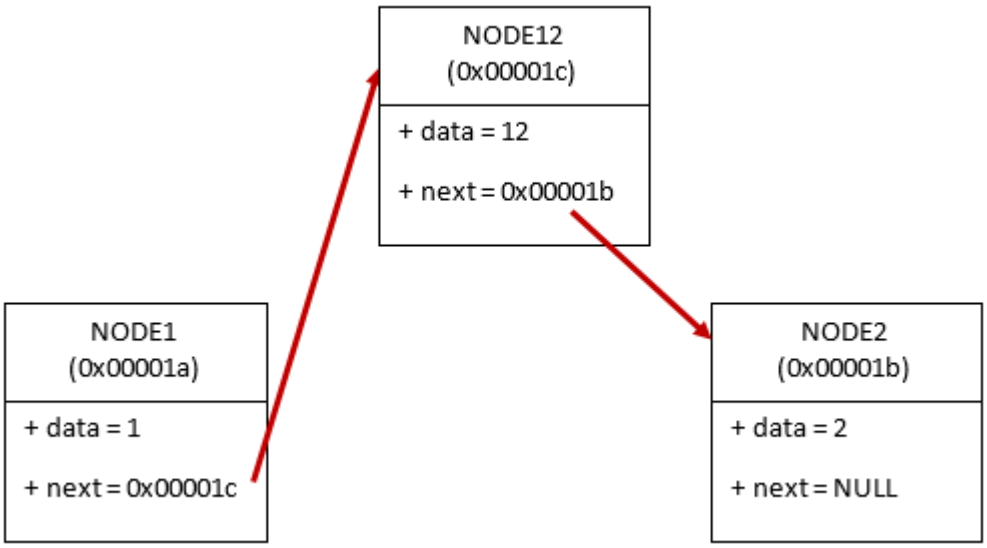



```
    //Knoten 12 zwischen Knoten 1 und 2 einfügen
    listnode *node12;
    node12->data=12;
    node12->next=node2;
    node1->next=node12;
```

Schritt 1



Schritt 2



Automatisiertes Hinzufügen von Knoten

Ein neuer Listenknoten wird durch Aufruf von new erzeugt. Dabei muss darauf geachtet werden, dass der Zeiger next gleich korrekt gesetzt wird. Wenn Sie nicht sofort den Nachfolger einhängen können, setzen Sie den Zeiger auf NULL.

```
#include <iostream>


using namespace std;

typedef struct listnode{
```

```
    int data;
    listnode *next;

};

int main(int argc, char** argv) {


    //Kopf der Liste
    listnode *head=NULL;


    //10 Knoten hinzufügen
    for(int i=1; i<10; i++)
    {
        listnode *node = new listnode;
        node->data=i;
        node->next=head;
        head=node;
    }


    //Knoten ausgeben
    listnode *help=NULL;
    help=head;

    while(help!=NULL)        //Solange Hilfszeiger nicht auf NULL zeigt
    {
        cout << help->data << endl;
        help=help->next;
    }

        return 0;
}
```

- 1.3.1) Musterbeispiel

```
#include <iostream>
using namespace std;
/* run this program using the console pauser or add your own getch,
system("pause") or input loop */

typedef struct listnode{

    int data;
    listnode *next;
};

listnode *head=NULL;

void vorneeinhaengen(int n)
{
    for(int i=0;i<n;i++)
    {
        listnode *node = new listnode;    //Anlegen eines neuen Knotens
        cout << "Wert eingeben: ";
        cin >> node->data;                //Daten in den Knoten übernehmen
        node->next=head;                 //Zeiger node->next zeigt auf ersten
Knoten
        head=node;                        //Listenkopf ist der neue Kneten
    }

}


void listeausgeben()
{
    listnode *h=NULL;        //Hilfszeiger
    h=head;                    //der auf den Kopf der Liste zeigt.

    while (h!=NULL)        //Solange Hilfszeiger nicht auf NULL zeig
    {
        cout << h->data << " -> ";
        h=h->next;
    }
}

int anzahlelemente()
{
    int anzahl=0;
    listnode *h=NULL;
    h=head;
    while(h!=NULL)
    {
        anzahl++;
        h=h->next;
    }
    return anzahl;
```

```
}

int main(int argc, char** argv) {

   int anzahl=0;
   cout << "Einfach verkettete Liste\n";
   cout << "Geben Sie an, wie viele Elemente Sie einhaengen wollen: ";
   cin >> anzahl;
   vorneeinhaengen(anzahl);
   cout << "Ausgabe der Liste\n";
   listeausgeben();
   cout << "Die Anzahl aller Elemente betraegt: " << anzahlelemente();

    return 0;


}
```