Informatik I

Michael Van Huffel

1 Zahlen und C++ Notation

1.1 L-Werte und R-Werte

L-Wert (Links vom Zuweisungsoperator − Bsp. Variablename)) :

- Ausdruck mit Adresse
- Wert ist der Inhalt an der Speicheradresse entsprechend dem Typ des Ausdrucks
- L-Wert kann seine Wert ändern (z.B. per Zuweisung).

R-Wert (Rechts vom Zuweisungsoperator − Bsp. Literal 0) :

- Ausdruck der kein L-Wert ist
- Jeder L-Wert kann als R-Wert benutzt werden **(aber nicht umgekehrt)**
- Ein R-Wert kann seinen Wert nicht ändern

1.2 Ganze Zahlen

Wertebereich (`int`) : $(32Bit : 2^{32-1} - 1 = 2147483647)$. Von $\{-2^{B-1}, \ldots, (2^{B-1}-1)\}$, wobei B = Bits (std: $B = 32$).

Wertebereich (`unsigned int`) : $\{0, \ldots, 2^{B-1}\} \rightarrow$ Operatoren : (32Bit: $2^{32} - 1 = 4294967295$). *Se va sotto 0 bisogna aggiungere* 2^{32}.

1 Byte = 8 Bits, 28 mögliche Zustände, 2 Bytes = 16 Bits, 216 mögliche Zustände sizeof(int) gibt Grösse in Bytes.

1.3 Binäre und Hexadecimale Darstellung

Binäre Darstellung : Grössenordnung von 2-Potenzen.
Beachte negative zahlen : $-3 = |-3+1| = (0010)_2 \rightarrow (1101)_2$.
Hexadecimale Darstellung : Zahlen zur Basis 16. Umwandlung siehe Tabelle. *Nützliches :* $16^2 = 256, 16^3 = 4096, 16^4 = 65536, \ldots$

$14_{10} \rightarrow$ 2)14 → 0; 2)7 → 1; 2)3 → 1; 2)1 → 1 $\rightarrow 1110_2$

$937_{10} \rightarrow$ 16)937 → 9; 16)58 → 10; 16)3 → 3 $\rightarrow 3A9_{16}$

Dec	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Bin	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111
Hex	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f

Wozu Hexa? Ein Hex-segno entspricht genau 4 Bits. Die Zahlen 1, 2 ,4, 8 repräsentieren Bit 0, 1, 2, 3. "Kompakte Darstellung von Binär".

1.4 Fliesskommazahlen

Ein Fliesskommazahlensystem ist durch vier natürliche Zahlen definiert: $F(\beta, p, e_{min}, e_{max})$, wobei F enthält die Zahlen $\pm \sum_{i=0}^{p-1} d_i \beta^{-i} \cdot \beta^e$, mit $d_i \in \{0, \ldots, \beta - 1\}, e \in \{e_{min}, \ldots, e_{max}\}$.

$\beta \geq 2$, die Basis — $p \geq 1$, die Präzision

e_{min}, der kleinste Exponent — e_{max}, der grösste Exponent

Normalisierte Darstellung : $\pm d_0 \bullet d_1 \cdots d_{p-1} x \beta^e, d_0 \neq 0$.
Beachte : Die normalisierte Darstellung ist eindeutig und deshalb zu bevorzugen. Die Zahl 0, sowie alle Zahlen kleiner als $\beta^{e_{min}}$, haben keine normalisierte Darstellung. Un FKZS è sempre in Norm. Dar.!

14.25	2^3		2^2	2^1	2^0	•	2^{-1}	2^{-2}
Bin	1		1	1	0	•	0	1
Norm	d_0	•	d_1	d_2	d_3	d_4	d_5	d_6

Chiaramente poi devo aggiustare a seconda della precisione richiesta e moltiplicare per $e^?$.

Rechnen mit Fliesskommazahlen :

1. Exponenten anpassen durch Denormalisieren eine Zahl
2. Binäre Addition der Signifikanden
3. Renormalisierung
4. Runden auf p signifikante Stellen, falls nötig

Wertebereich herausfinden : $F^*(\beta = 2, p = 3, e_{min} = -4, e_{max} = 4) \longrightarrow W = (b^p - b^{p-1}) \cdot$ Range exponent

1. $p = 3 \rightarrow 8$ Binäre Zahlen (111, 100, 101, 110, 000, 001, 010, 011)
2. Fur normalisierte Darstellung alle nicht-normalen Zahlen (011, 010, 001, 000) streichen $\rightarrow$ 4 Zahlen bleiben ubrig
3. Exponent kann 9 Werte $[-4, 4]$ annehmen
4. $4 \cdot 9 = 36$ Zahlen konnen mit F^* dargestellt werden

2 Operatoren Priorität und Variablen

Bemerkung : Variablen, die definiert, aber noch nicht initialisiert wurden, enthalten Zufallswerte.
postfix : x++ (x−−) : Es wird zuerst die Zeile ausgeführt und dann die Variable aktualisiert.
prefix : ++x (−−x) : Es wird zuerst die Variable aktualisiert und dann die Zeile ausgeführt.

```
int x = y++ // int x = y; y = y  + 1 // postfix
int x = ++y // y = y  + 1; int x = y // prefix
```

Priorität	Operator	Richtung
1	: : (scope resolution)	→
2	++ - - (suffix in-/decrement) () (function call) [ ] (array subscripting) . (selection by reference) ->(selection through pointer)	→

3	++ - - (prefix in-/decrement)
	! (logical NOT)
	(`int`) (type cast)
	* (dereference)
	& (address-of)
	`sizeof` (size-of)
	`new`, `new{[]{]}`
	`delete`, `delete{[]{]}`
		←

4	.* ->* (pointer to member)	→
5	* / %	→
6	+ -	→
7	<< >>	→
8	< <= > >=	→
9	== != (Vergleichsoperatoren)	→
10	&& (logical AND)	→
11	| | (logical OR)	→
12	= (direct assignment) += -= *= /= %=	←
13	,	→

Vorgehen: Operatoren paarweise vergleichen, kleinstes Priorität zuerst ausführen. Falls gleiches Priorität, Richtung betrachten.
De Morgan Rule : `!(a && b)=(!a || !b)`, `!(a || b)==(!a && !b)`.

3 Datentypen und Typenumwandlung

Data-type	Function	Memory
`int`	Saves integer numbers	2/4 Byte
`float`	Saves real numbers	4 Byte
`double`	Saves real numbers with higher precision than `float`	8 Byte
`bool`	Saves logical expressions	1 Byte
`char`	Saves characters	1 Byte

```
5.0 // double
6.7f // float (or 2.6F)
9.4l // long double (or 9.4L)
1e1 * 2e2 // double 2000 --> e1 == 10*
```

Gleitkommazahlen werden `double` ausser man verwendet ein suffix f, F, l oder L.

...bei Zuweisungen

Faustregel : Wenn ein (numerischer) Wert einer Variable eines anderen Typs zugewiesen wird, passt sich der Typ des Werts an den Typ der empfangenden Variable an. **Beachte** `unsigned int`!

1. float/double → int : Kommastellen werden wegelassen
2. char → float/double/ int : Nummer des Zeichen im ASCII Code wird als Zahl gespeichert
3. float/double/ int → char : Zahlen bis 255 werden gemass ASCII-Code in Zeichen konvertiert
4. float/double/ int → bool : 0 wird zu false, alle anderen Werte zu true
5. bool → float/double/ int : false wird zu 0, true wird zu 1

...bei arithmetischen Operationen

Faustregel : C++ wandelt Ausdruck in generellsten Datentyp.

```
5.0 / 2f = 2.5, da double / float = double
5.0f / 2 = 2.5, da float / int = float
5.0 / 2 = 2.5, da double / int = double
-3.0 * 2u = -6.0 da double * unsigned int = double
'b' + 'A' - 'B' // char 'a'
17 % 5u // unsigned int 2
int i = 3.2; //i=3
float b = 7/2; //b=3.0
```

...explizit

```
float a = float(3/2 + 4.0f); //a = 5.0
float b = float(3)/2 + 4.0f; //b = 5.5
```

...wichtig

1. Literale 0.1/1.1e1 und 0.1f/1.1e1f haben gleich viele signifikate Stellen. (**F**/T)

2. Es gibt ein Wert a vom Typ float fuer den gilt: a + 1 == a. (**T**))

3. Jede 32-bit/16-bit Zahl vom Typ int kann ohne Wert anderung in den Typ float konvertiert werden. (**F**/T)

4 Funktionen

Der Rückgabewert wird immer zum Rückgabetyp konvertiert. Jede Funktion, die nicht den Rückgabetyp void hat, muss ein return haben Beachte auch auf scope / Gültigkeitsbereich! Eine Variable, die in einer Funktion definiert wurde, kennt im Allgemeinen nur diese Funktion.

4.1 Call by value

Die Werte der übergebenen Variablen werden kopiert. Man arbeitet also inner- halb der Funktion mit Kopien. Die Originalvariaben bleiben unverändert.

```
void swapv(int a, int b) //copy of values
  //changes not visible outside of function
{
int temp = a; a = b; b = temp;
}
int main()
{
  int wallet1 = 300; int wallet2 = 350;
  swapv(wallet1, wallet2); //wallet1 = 300, wallet2 =
       350 return 0;
}
```

4.2 Call by reference : Pointers

Es werden keine Werte u bergeben/kopiert, sondern Adressen. Ueber diese können wir innerhalb der Funktion auf die Originalvariablen zugreifen und sie ändern.

```
void swapp(int *p, int *q) //copy of addresses
   //changes visible outside of function
{
int temp = *p; *p = *q; *q = temp;
}
int main()
{
  int wallet1 = 300; int wallet2 = 350;
  swapp(&wallet1, &wallet2); //Adressen übergeben! //
       wallet1 = 350, wallet2 = 300
  return 0;
}
```

4.3 Call by reference : Reference

Es werden Aliase erzeugt, mit denen wir innerhalb der Funktion auf die Originalwerte zugreifen können.

```
void swapr(int &a, int &b) //changes visible outside of
     function
  //no copy of address is created
{
int temp = a; a = b; b = temp;
}
int main()
{
  int wallet1 = 300; int wallet2 = 350;
  swapr(wallet1, wallet2); //pass reference //wallet1 =
       350, wallet2 = 300
  return 0;
}
```

Achtung : Falls die Funktion nicht nur Werte zurückgeben, sondern auch Originalwerte ändern soll, muss man mit Call by Reference (Pointers/Referenzen) arbeiten!!

4.4 Andere Typ von Funktionen

In C++ kann man Funktionen überladen. Man kann also zwei Funktionen definieren, die den gleichen Namen haben, sich aber durch ihre Argumente unterscheiden. Anhand der übergebenen Argumente kann das Programm zwischen den Funktionen unterscheiden.

Katalogisierung von Befehlen : namespace

```
namespace ifmp { // namespace called ifmp
// POST: "Hi" was written to the terminal
void output func () { // this function is in namespace
     ifmp
        std::cout << "Hi";
}
}
int main () {
ifmp::output func(); // use output func from namespace
     ifmp return 0;
}
```

5 Verzweigungen und Loops

5.1 if, else, if else

```
if (<erste Bedingung>) {
  DoSomething;
}
else if (<zweite Bedingung>) {
  DoSomethingElse;
}
else {
  IfEverythingElseFails; //keine Bedingung
}
```

Beachte : else if wird nur ausgeführt wenn die vorherige Bedingung nicht erfüllt ist! Man kann beliebig viele else if anhängen. Sobald aber ein Block ausgeführt wurde, das heisst sobald die erste Bedingung erfüllt ist, springt das Programm an das Ende der ganzen if-else if-else Struktur.

5.2 switch-Anweisung

Nur int oder char Daten. Wenn break nicht vorkommt, wird alles ausgeführt bis zum nächsten break.

```
int x = 1;
switch(x) {
  case 0:
    // code
    break;
  case 1:
    // code
    break;
  case n: ...
  default:
    // code
}
```

5.3 for-loop

```
for (int i=0; i<4; i++)
{
cout << "i is: " << i;
 }
```

Ablauf : Bedingung wird überprüft; Falls true : Befehle werden ausgeführt. Falls false : Programm springt ans Ende der Klammer / Schleife; Index wird geändert.

5.4 while-loop

```
int i = 0;
while (i < 4)
 {
  cout << "i is: " << i;
   i++;
}
```

Ablauf : Bedingung wird überprüft; Falls true : Befehle werden ausgeführt. Falls false : Programm springt ans Ende der Klammer / Schleife; Index wird geändert (in der Schleife).

5.5 do while-loop

```
int i = 0;
do {
  cout << "i is : " << i;
  i++;
} while (i<4);
```

Ablauf : Befehle werden ausgeführt; Index wird geändert (in der Schleife); Bedingung wird überprüft. Falls false : Programm springt ans Ende der Klammer / Schleife. Falls true : Befehle werden nochmal ausgeführt.

5.6 for vs while

for : für Schleifen mit bekannter Anzahl an Durchgängen; fu r i=x bis y.

while : für Schleifen mit unbekannter Anzahl an Durchgängen; solange Bedingung = true; für komplexe Bedingungen.

5.7 break/continue

- Mit `break` wird die Schleifen unmittelbar verlassen
- `continue` überspringt den Rest des Funktionskörpers der Schleife und geht zur nächsten Auswertung der Bedingung

6 Referenzen

Referenzen können nur Variablen ihrens zugrundeliegenden Typs referenzieren. Sonst gibt es einen Fehler. Referenzen nur mit L-Werten initialisiert werden! Funktionen, bei denen die Argumente Referenztyp haben, können ihre Aufrufargumente ändern.

```
// Usage
int a = 3;
int& b = a; // reference to a
b++;
std::cout << b << "\n"; // Output: b = 4;
std::cout << a << "\n"; // Output: 4;

// Issues
int& c = 3; // Error: 3 is not an lvalue (3 has no
     address)
bool d = false;
int& e = d; // Error: d is bool, e wants to reference an
      int

//Example from old Exam:
std::vector<int> a = {7, 6, 5, 3}
int* x = &a[0] +1 // x punta a 6
```

6.1 const Referenzen

Im Prinzip funktionieren const Referenzen so wie normale Referenzen, bloss dass der Schreibzugriff auf das Ziel der Referenz via diese Referenz verboten ist.
Weiterer Unterschied: const Referenzen R-Werte beinhalten können. Dann wird jeweils ein temporärer Speicher für den R-Wert erstellt, der solange gültig ist, wie die const Referenz selbst. Dies erlaubt beispielsweise, eine Funktion bezüglich Call-by-Reference trotzdem mit R-Werten aufzurufen.
Achtung : man darf keine nicht-const Referenz mit einer const Referenz initialisieren.

```
double a = 3.0;
double& b = a; // non-const reference
const double& c = a; // const reference

c = 4.0; // Error: write-access forbidden
a = 5.0; // this works, a can be changed through itself
b = 6.0; // this works, a can be changed through non-
     const refs

std::cout << c << "\n"; // Output: 6.0, read-access is
     allowed.
double& d = c; // Error: non-const ref from const ref
     not allowed
const double& e = 5.0; // this works for const
     references.
```

7 Rekursion

Eine Funktion ist rekursiv wenn sie sich selbst aufruft.

Beispiel 1 : Fattoriale

```
int factorial(int n) {
  if (n == 0) //Basisfall {
    return 1;
  }
  int result = n * factorial(n-1);
  return result;
}
```

Beispiel 2 : Fibonacci Reihe

```
int fibonacci(int n) {
  if (n <= 1) //Basisfall {
    return 1;
  }
  int result = fibonacci(n-1) + fibonacci(n-2);
  return result;
}
```

Beispiel 3 : Potenzen n^m

```
int power(int n, int m) {
  if (m == 0) {
    return 1;
  }
  return n*power(n,m-1);
}
```

8 Statische Arrays

8.1 1-dimensional (Liste/ "Vektor")

```
//Initialisierung (bei Definition mit {}):

std::vector<int> a = {8, 9, 10, 11};
std::cout << a[0]; // outputs 8
a[3] = 5; // a is 8, 9, 10, 5

int c[5] = {1,2}; //restliche Elemente = 0

int d[] = {11,34,2}; //ohne Groesse --> Elemente muessen
      - initialisiert werden!
```

Achtung : Falls bei einem Aufruf die Grösse des arrays überschritten wird, gibt der Compiler keinen Fehler aus (kein compile time error)! Das Programm wird also kompilieren und fehlerhaft laufen (runtime error). Array ist kein L-Value und darf *nicht* auf der linken Seite einer Zuweisung stehen.
Nicht vergessen: Indizes beginnen bei 0 und nicht 1.

```
int e[3] = {1,2,3};
int zahl = e[4]; //runtime error!
```

8.2 2-dimensional (Tabelle/ Matrix")

```
//Definition:
std::vector<std::vector<int> > my vec (n rows, std::
     vector<int>(n cols, init value))

//Beispiel :
```

```
std::vector<std::vector<int> > my vec (2, std::vector<
    int>(4, 0));
my vec.at(1).at(2) = 3;
// my vec becomes
// 0, 0, 0, 0
// 0, 0, 3, 0
```

Beachte : La grandezza delle colonne deve essere specificata!

9 Strings

string = array von chars = Zeichenkette! Kann Wörter oder Sätze inhalten. *Nuetzliches :* 97 für 'a', 65 für 'A', 48 für '0' und 32 für ' '.

9.1 Initialisierung

1. als array von char's :

```
char name[4] = {'H','a','n','s'}; // kein string
char name[5] = {'H','a','n','s','\0'}; // string
```

2. als Zeichenkette:

```
char name[] = "Hans"; //ohne '\0'
```

3. mit Class string :

```
string name = "Hans";
```

Achtung : '...' um einen char zu initialisieren, "...üm eine Zeichenkette zu initialisieren!

```
char a = 'x'; // valid, char
char b = "y"; // invalid, string
```

9.2 Strings Input und Output

```
char c;
// Version 1: Assume the user enters:
// a b
std::cin >> c; // read a
std::cin >> c; // read b
// Version 2: Assume the user enters again:
// a b
std::cin >> std::noskipws;
std::cin >> c; // read a
std::cin >> c; // read
std::cin >> c; // read b

//Ausgabe:
cout << name; // nur fuer arrays of char's (string)!

int zahlen[2] = {1,2};
cout << zahlen; //invalid!!
```

9.3 Nützliche Funktionen

Datentyp für Zeichen : (in <cstring>)

variable Länge :	std::string my_str (n, 'a'), (n variabel)
Länge abfragen :	my_str.length()
vergleichbar :	text1 == text2
hintereinander hängen:	text1.push_back('blabla')
	text1 += text2

```
std::string my_word (5, a); // initialize my word as
    aaaaa
std::string ref (5, z);
my_word += ref; // append ref to my word.
                // Afterwards my word: aaaaazzzzz
                // Afterwards ref: zzzzz
word.length() << my_word.at(3) = b; // change
if (my_word == ref) { // false
  std::cout << "not output\n";
}
std::cout << my_word << "\n"; //
std::cout << my "\n"; // output: 10 my word to
    aaabazzzzz output whole string at once
```

10 Input/Output von Files

```
#include <iostream> #include <string> #include <fstream>
int decode(std::ifstream & in){
  unsigned char word = 0; std::string buchstabe;
  while(in >> buchstabe)
    if(buchstabe.length() == 8){//else error word length
          for(int i = 0; i < 8; ++i){
              word *= 2;
              if(buchstabe.at(i) == '1')
               word++;}}
    std::cout << word;}
  int main(){
  std::string file; std::cin >> file;
  std::ifstream in(file);
  if(!(bool)(in)){
  std::cout<< "File doesn't exist"; return 0;}
  decode(in);}
```

11 EBNF

Metasprache zur Darstellung kontextfreier Grammatiken. Art Anleitung, welche Kombinationen gultig sind und welche nicht.
Parsen : Testen, ob ein Satz nach EBNF gultig ist.

Alternative (oder) :	...|...
Optionale Wiederholung (0 to ∞), (0 to 1):	{...}, [...]
Gruppierung :	(...)
Aufzahlung (Elenco) :	... , ...
Definition :	··· = ...
Bezeichung terminale Symbole :	"..." '...'

```
Option = ["+"|"-"] 1234; //aquivalent zu
Option = 1234 | +1234 | -1234;

//Beispiel:
// POST: guarda il prossimo carattere senza consumare
char lookahead(std::istream& is) {
    is >> std::ws;             // skip whitespaces
    if (is.eof()) {
        return 0;              // end of stream
    } else {
        return is.peek();      // next character in is
    }
}
// POST: true if match, false otherway
bool consume(std::istream& is, char expected) {
    char actual;
    is >> actual;
    return actual == expected;
}
//Var = "_" Var | Letter {Letter},
//Varlist = Var {"," Var } ":" Type.
bool Var(std::istream& is) {
  if(lookahead(is) == '_'){
    return consume(is, "_") && Var(is);
  }
  else if(Letter(is)){
    while(Letter(is));
    return true;
  }
  else return false;}
bool VarList(std::istream& is) {
  if(Var(is)){
    while(consume(is, ",") && Var(is));
    return consume(is, ":") && Type(is);
  }
  return false;}
```

12 Structs

Ein struct ist ein L-Value und darf auf der linken Seite einer Zuweisung stehen. Nur der Zuweisungsoperator (=) wird automatisch erstellt (und kopiert dann die Member einzeln). Die anderen Operatoren (z.B. ==, !=, ...) muss man selbst passend überladen (siehe operator).
Bei der Default-Initialisierung eines Objekts des Typs str_name werden alle Member einzeln default-initialisiert. Für fundamentale Typen (int, float, usw.) bedeutet das, dass sie uninitialisiert sind, bis man ihnen nachträglich einen Wert zuweist. Das führt zu Problemen, falls man ihren Wert vorher schon ausliest.

Definition :	struct str_name { int mem1; bool mem2; };
Objekt erstellen :	str_name obj1;
mit Startwerten :	str_name obj2 = 3, true, 4;
aus anderem Objekt :	str_name obj3 = obj2;
Zugriff auf Member :	obj1.mem1

Achtung : Die Definition eines Structs hat ein ; am Schluss.

Operator-Uberladung (mittels `operator`) : wird zum Beispiel verwendet, um Operatoren (+, -, *, etc.) auf eigenen Structs zu definieren.

```
//.h
struct Complex {
    double reale;
    double immaginario; // INV: Non c'è
};
// POST : return value is the sum of a and b
Complex operator+(const Complex a, const Complex b);
// POST : return if a equal
bool operator==(const Complex a, const Comple
// POST : return the output
std::ostream& operator<<(std::ostream& out, Complex r);

//.cpp
Complex operator+(const Complex a, const Complex b){
  Complex result;
  result.reale = a.reale + b.reale;
  result.immaginario = a.immaginario + b.immaginario;
  return result;
}
bool operator==(const Complex a, const Complex b){
  return (a.reale == b.reale) && (a.immaginario == b.
      immaginario);
}
std::ostream& operator<<(std::ostream& out, Complex r){
    return out << "[" << r.reale << "," << r.immaginario
         << "]";
}

//main
Complex marsan;
std:cout<< marsan.reale; //Undefined behavior
marsan.reale = 2; marsan.immaginario = 1;
std::cout<< marsan.reale //Problem gone
```

13 Klassen

Eine Klasse besteht aus Daten und Funktionen, genannt Member, und erlaubt deren Kapselung via Zugriffkontrolle: Auf Member im privaten Teil (`private`) einer Klasse kann nur durch die Klasse selbst, d.h., deren Member-Funktionen zugegriffen werden.
Einziger Unterschied gegenu ber structs: Member in structs sind per default o ffentlich (`public`).

```
class my class {
public: // public section
  double some_public_member;
private: // private section
  double some_private_member;
};
...
my class inst;
  inst.some_public_member = 1.0;
  inst.some_private_member = 0.0; // ERROR: cannot
      access private members directly
```

13.1 Memberfunktion

Memberfunktionen ermöglichen den kontrollierten Zugang zu den privaten Daten und privaten Memberfunktionen. Die Deklaration einer Memberfunktion erfolgt immer in der Klassendefinition, die Definition der Memberfunktion ist auch extern möglich (ermöglicht vorkompilierte Libraries). Dann muss allerdings benutzten der ::-Schreibweise.

```
class Insurance {
public:
  void set rate i (const double v) { rate = v; } //
       internal Ruf
  void set rate e (const double v);
...
private:
    double rate;
...
};
void Insurance::set rate e (const double v) {rate = v;}
     // extternal Ruf
```

Der Aufruf einer Memberfunktion ist `obj.mem func(arg1, arg2, ..., argN)`. Der Teil `obj.` kann weggelassen werden, falls wir in der Classe finden.

```
class Insurance {
public:
  double get rate () {
            if (!is_up_to_date) update_rate(); // from
                 inside
      return rate;
  }
private:
bool is_up_to_date;
double rate;
double update_rate () { rate = ...; }
};
...
Insurance insurance;
...
std::cout << insurance.get rate(); // from outside
```

const Memberfunktion : Das const bezieht sich auf `*this`. Es verspricht, dass durch die Funktionsaufruf das implizite Argument nicht im Wert verändert wird.

```
class Insurance {
public:
  double get value() const {
    return value; // same: return (*this).value;
  }
private:
  double value;
};
```

13.2 Konstruktor

Konstruktoren sind spezielle Memberfunktionen einer Klasse, die den Namen der Klasse tragen. Sie werden bei der Variablendeklaration aufgerufen. Damit das funktioniert, muss der Konstruktor öffentlich (`public`) sein.

```
class Insurance {
public:
  Insurance(double v, int r) // general constructor
    : value (v), rate (r) // initialize data members
    { update_rate(); }
  Insurance() // default constructor
    : value (0), rate (0) // initialize data members
    {}
private:
  double value;
  double rate;
  void update_rate();
};
...
// General Constructor
Insurance i1 (10000, 10);
// default-Constructor, direct call
Insurance i3; // identical: Insurance i3 ();
```

Spezielle Konstruktoren sind der Default-Konstruktor (kein Argument), welcher automatisch erzeugt wird, falls eine Klasse keinen Konstuktor definiert, und der Konversions-Konstruktor (genau ein Argument), welcher die Definition benutzerdefinierter Konversionen ermöglicht.

```
class Complex {
public:
  // Conversion Constructor (float --> Complex)
  Complex(const float i) : real (i), imag (0) { }
private:
  float real;
  float imag;
};
```

14 Pointers

Pointer = Variable, die Adressen von Speicherblöcken speichern kann.

Der Wert des Zeigers ist die Speicheradresse des Targets. Will man also das Target via diesen Zeiger verändern, muss man zuerst zu der Adresse gehen. Genau das macht der Dereferenz-Operator *.

Nullpointer : `int *np = nullptr;` Pointer, der auf nichtsßeigt.
Achtung : kann nicht dereferenziert werden (Nullpointer exception)!!

```
<type> *<name>; //Pointervariable "name"
<type>* <name>; //identisch
```

Initialisierung : Um auf die Adresse einer bestimmten Variable zuzugreifen, benutzen wir den Reference Operator &.

```
int a = 2; //int Variable a
int *b; //Pointervariable b, zeigt auf einen int
b = &a; //b = "Adresse von a" / b "zeigt" auf a
```

Dereferenzierung : Um von einer Adresse auf den Wert an der Adresse zu kommen, benutzen wir den dereference operator *. Achtung : Nicht zu verwechseln mit dem * der Definition eines Pointers!

```
int c = *b; //c = "Wert an der Adresse b"
```

Den dereference operator kann man also nur in Kombination mit einer Adresse benutzen.

14.1 Pointer auf Arrays

```
int myArray[5] = {1,2,3,4,5}; //array
int *pArray; //pointer auf Typ int;

pArray = myArray; //pArray = Adresse von myArray[0]
pArray = &myArray[0]; //Same
```

Iterieren :

Zeiger :	`int* ptr = new int[6];`
temporärer Shift :	ptr + 3, ptr - 3
permanenter Shift :	`++ptr, ptr++, --ptr, ptr--, ptr += 3, ptr -= 3`
Distanz bestimmen :	`ptr1 - ptr2`
Position vergleichen :	`ptr1 < ptr2`
Werte der Elemente :	`myArray[n] = *(myArray + n), //Wert n +1 eleme.`

Achtung : Die röten Shifts erzeugen einen neuen (temporären) Zeiger und verschieben `ptr` nicht. Die permanenten Shift verschieben aber `ptr`.

```
// Read 6 values into an array
std::cout << "Enter 6 numbers:";
int* a = new int[6];
int* pTE = a+6;
for (int* i = a; i < pTE; ++i)
  std::cin >> *i; // read into array element
// Output:   a[0]+a[3],  a[1]+a[4],  a[2]+a[5]
for (int* i = a; i < a+3; ++i) {
  assert(i+3 < pTE); // Assert that i+3 stays inside.
  std::cout << (*i + *(i+3)) << ", ";
}
```

14.2 Pointer und const

Es gibt zwei Arten von Konstantheit:

kein Schreibzugriff auf Target :	`const int* a ptr = &a;`
kein Schreibzugriff auf Zeiger :	`int* const a ptr = &a;`

```
int a = 5;
int b = 8;

const int* ptr 1 = &a;
*ptr 1 = 3; // NOT valid (change target)
ptr 1 = &b; // valid (change pointer)

int* const ptr 2 = &a;
*ptr 2 = 3; // valid (change target)
ptr 2 = &b; // NOT valid (change pointer)

const int* const ptr 3 = &a;
*ptr 3 = 3; // NOT valid (change target)
ptr 3 = &b; // NOT valid (change pointer)
```

14.3 Zugriff auf implizites Argument: *this

Memberfunktionen einer Klasse haben ein implizites Argument, das aufrufende Objekt. Und `this` ist ein Zeiger darauf. Via `*this` kann man darauf zugreifen. Man muss es nicht unbedingt explizit angeben (automatisch verwendet).

```
class Human {
public:
  void set (const int a) { age = a; } // or (*this).age
       = a;
  void print1 () const { std::cout << (*this).age; }
  void print2 () const { std::cout << age; } //
       equivalent
private:
    int age;
};
Human me;  me.set(175);
me.print1(); // 175
me.print2(); // 175
```

Achtung : Man muss *this aber mindestens explizit verwenden, falls z.B. eine Referenz auf das implizite Argument zurückgegeben werden soll.

```
class Complex {
public:
  // Note: In most applications
  // a reference should be returned.
  Complex& operator+= (const Complex& b) {
    real += b.real;
     imag += b.imag;
     return *this;
}
... // other members
private:
  float real;
  float imag;
 };
```

15 Dynamische Datentypen

Typ, der dynamischen Speicher verwaltet (z.B. unsere Klasse für Stapel)

- Konstruktoren (siehe Klassen)
- Destruktor
- Copy-Konstruktor
- Zuweisungsoperator

} **Dreierregel :** definiert eine Klasse eines davon, so muss sie auch die anderen zwei definieren!

15.1 Copy-Konstruktor

Der Copy-Konstruktor ist der Konstruktor, dessen Argumenttyp `const My Class&` ist.

```
class avec {
  unsigned int count; // Size of the vector
  tracked* elements; // Vector elements (actually, a
       pointer to the first element)

public:
  // POST: Instantiated a new avec with 'size' elements
  avec(unsigned int size); // Constructor

  // POST: COPY CONSTRUCTOR
avec::avec(const avec& vec): count(vec.count) {
  elements = new tracked[count];
  for(unsigned int i = 0; i < count; ++i){
        elements[i] = vec.elements[i];
   }
}
  // POST: Assign to content of vec, return reference to
        the this-object.
  avec& operator=(const avec& vec); // Vedi 15.2

  // POST: Deallocate the content.
  ~avec(); // Vedi 15.3
//..other functions
};
```

15.2 Kopier-Zuweisung

Eng verwandt mit `operator=` ist der Copy-Konstruktor. Der Unterschied ist, dass der Copy-Konstruktor nur bei der Initialisierung aufgerufen wird, `operator=` hingegen nur nach der Initialisierung.

```
my class a (5, 6), c (4, 4); // Call a general
     constructor
my class b = a; // Call copy-constructor
c = b; // Call operator=
```

`operator=` kann anders als der Copy-Konstruktor implementiert werden müssen. Beispiel: Stack.
`operator=` gibt im Normalfall eine Referenz auf seinen linken Operanden zurück.

Faustregel : Meistens führt `operator=` zuerst die Aufgaben des Destructors, und dann dise Aufgaben des Copy-Konstruktors aus.

```
avec& avec::operator=(const avec& t) {
    avec copy(t);
    std::swap(elements, copy.elements);
    std::swap(count, copy.count);
    return(*this);
}
```

15.3 Destruktor

```
avec::~avec() {
    delete[] elements;
}
```

15.4 new/ delete

new weist benötigten Speicherplatz zu und gibt einen Pointer auf das 1. Element des neu zugewiesenen Speichers zurück.

```
<type> *name = new <type>;
<type> *name = new <type> [number of elements]; //
     dynamic array
//Besipiel:
int *a = new int;
*a = 9;
delete a;
```

Für jedes new muss ein delete (resp. delete[]) für arrays vorkommen. Dynamische Strukturen sollten gelöscht werden, sobald man sie nicht mehr braucht, da man sonst ein Speicherleck hat und das Programm irgendwann abstürzt, weil kein Arbeitsspeicher mehr verfügbar ist. (Aufräumen).

15.5 Dynamic array

Beispiel: array, dessen Grösse durch den Benutzer mit cin eingelesen wird → dynamisch Speicherplatz mit new zuweisen, und wieder mit delete[] leeren.

```
int n; std::cin >> n;
int* range = new int[n];
// Read in values to the range
for (int* i = range; i < range + n; ++i) std::cin >> *i;
delete range; // ERROR: must say: delete[]
delete[] range; // This works
```

15.6 Dynamic struct

```
struct Student{
  char vorname[20];
  string name; //#include <string> int legi;
};
Student *stud1 = new Student; //dynamisches struct

(*stud1).legi = 12123123; //oder :
stud1->legi = 12123123;
strcpy(stud1->vorname, "Max"); //#include <cstring>
     stud1->name = "Mustermann";

delete stud1;
```

Zugriff auf Elemente : ->. Alternativ mit dereference operator *.

16 Iteratoren und Kontainers

16.1 Iteratoren auf Vektoren

Im Folgenden wird nur auf die Unterschiede zum Pointer (auf Array) eingegangen. Die restliche Bedienung erfolgt gleich.
Achtung : Iteratoren erfordern #include<vector>.

Wichtige Befehle (gelte std::vector<int> a (6, 0)):

Definition :	std::vector<int>::iterator itr = ...;
Iterator auf a.at(0) :	a.begin()
Past-the-End-Iterator :	a.end()

```
// Example for vectors.
// To avoid the lengthy lines see entry on typedef.
// Read 6 values into a vector
std::cout << "Enter 6 numbers:\n";
std::vector<int> a (6, 0);
for (std::vector<int>::iterator i = a.begin(); i < a.end
     (); ++i)
  std::cin >> *i; // read into object of iterator

// Output: a.at(0)+a.at(3), a.at(1)+a.at(4), a.at(2)+a.
     at(5)
for (std::vector<int>::iterator i = a.begin(); i < a.
     begin()+3; ++i) {
  assert(i+3 < a.end()); // Assert that i+3 stays inside
       .
  std::cout << (*i + *(i+3)) << ", ";
}
```

16.2 const Iteratoren

Achtung : Einen const-Iterator erzeugt man mittels std::vector<int>::const iterator mups und nicht mittels const std::vector<int>::iterator mups. Die zweite Version erzeugt einen Iterator, den man nicht verschieben kann.

```
std::vector<int> a (6, -8); // a is: -8 -8 -8 -8 -8 -8
std::vector<int>::const iterator itr = a.begin() + 3;
*itr = 4; // NOT valid
itr = a.begin(); // valid (itr now points to a.at(0))
```

16.3 Set

Datentyp für Mengen (jedes Element kommt nur einmal vor und ist in wachsende Ordnung geordnet).
Achtung : erfordert #include<set>.

Definition :	std::set<int> my set (b, e);
	(Initialisiert my_set mit den Werten im Bereich $[b, e)$.)

Die Iteratoren der sets funktionieren wie die Iteratoren der Vektoren, aber:

Keine :	[], +, -, <, >, <=, >=, +=, -=
Zum Verschieben nur :	++..., ...++, --..., ...--, =
Zum Vergleichen nur :	==, !=

```
// Determine All Occurring Numbers
std::cout << "Enter 100 numbers:\n"; std::vector<int>
     nbrs (100);
int number;
for (int i = 0; i < 100; ++i)
  std::cin >> nbrs.at(i); //oppure std::cin>>number;
       nbrs.insert(number);

std::set<int> uniques (nbrs.begin(), nbrs.end());

// Output
typedef std::set<int>::iterator Sit;
for (Sit i = uniques.begin(); i != uniques.end(); ++i)
  std::cout << *i << "  ";
// This does not work:
for (int i = 0; i < uniques.end() - uniques.begin(); ++i
     )
  std::cout << uniques.at(i);
```

16.4 Kontainer nützliches

Kapazität (von std::vector<int> mups) **:**

Vektor grösse :	mups.size()
Resize Vektor somit 'n' element enthält :	mups.resize(n)
Returns true wenn Vektor leer ist	mups.empty()

Element access (to std::vector<int> mups) **:**

Referenze zur g Element :	mups[g], mups.at(g)
Referenze zur erst/letzte Element :	mups.front(), mups.back()

Modifiers (to std::vector<int> mups) **:**

Fügt das Vektor mit x, y mal :	mups.assign(y, x)
Fügt das letzte position mit x :	mups.push_back(x)
Eliminiert das letzte Element :	mups.pop_back();
Fügt x vor y :	mups.insert(y, x);
Fügt x in der Container :	mups.insert(x);
Eliminiert Elemente auf x Position von Vektor .	mups.erase(x);
Swap das inhalt von 2 Vektoren :	mups.swap(mups_2);
Eliminiert alle Elemente von Vektor (size == 0) :	mups.clear();

16.5 Kontainer funktionen

Achtung : diese funktionen erfordern `#include<algorithm>`

std::fill(b, p, val) : Wert `val` in einen Bereich $[b, p)$ einlesen

```
// Goal: Generate vector: 4 4 4 2 2
std::vector<int> vec (5, 4); // vec: 4 4 4 4 4
std::fill(vec.begin()+3, vec.end(), 2); //vec: 44422
```

std::find(b, p, val) : `val` suchen im Bereich [b,p).
Zurückgegeben wird ein Iterator auf das erste gefundene Vorkommnis. Wenn `std::find` nicht fündig wird, gibt es den Past-the-End-Iterator `p` zurück. (Beachte: Past-the-End ist bezüglich Bereich `[b,p)` gemeint.)

```
typedef std::vector<int>::iterator Vit; std::vector<int>
      vec (5, 2);
vec.at(3) = -7
// Goal: Find index of -7 in vec: 2 2 2 -7 2
Vit pos itr = std::find(vec.begin(), vec.end(), -7);
std::cout << (pos itr - vec.begin()); // Output: 3
```

std::sort(b, e) : Bereich [b, e) sortieren.
`std::sort` funktioniert nur, wenn Random-Access Iteratoren für `b` und `e` übergeben werden. Somit funktioniert `std::sort` z.B. für Felder und Vektoren, aber nicht z.B. für Sets.

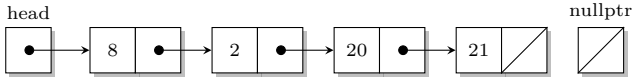
```
std::vector<int> vec = {8, 1, 0, -7, 7};
std::sort(vec.begin(), vec.end()); // vec: -7 0 1 7 8
```

std::min_element(b, p) : Iterator auf Minimum im Bereich [b,p).
Wenn das Minimum nicht eindeutig ist, so wird ein Iterator auf das erste Vorkommnis zurückgegeben.

```
// Goal: Make sure that all inputs are > 0
std::vector<int> vec (10, 0);
for (int i = 0; i < 10; ++i)
std::cin >> vec.at(i);

assert( *std::min element(vec.begin(), vec.end()) > 0 );
// We have to dereference the (r-value-)iterator!!
```

17 Verkettete Liste



Eine Liste ist eine Ansammlung von verketteten Knoten. Jeder Knoten speichert zwei Komponenten : einen Wert und die Adresse (Pointer) des nächsten Knotens. Zu jeder Liste gehört ein head-pointer. Dieser soll auf den ersten Knoten der Liste zeigen (= dessen Adresse speichern).
Der Pointer des letzten Knotens speichert die Adresse `nullptr`, zeigt also auf das Nullelement. Dies zeigt das Ende der Liste an.

17.1 Implementierung der Knoten als struct

```
struct Node{
  int value;
  Node *next;
};
typedef struct Nnode Node; //Node = Nnode (alias)
```

Wenn der Ersteller für eine Klasse der Verketteten Liste erforderlich ist, muss der Zeiger `head` auf den Zeiger `nullptr` initialisiert werden.

17.2 Liste erstellen

Nehmen wir an, wir wollen eine Liste von 3 Knoten erstellen. Wir müssen dafür zuerst die Liste initialisieren und dann mit Hilfe einer Schleife die erstellten Knoten mit den gewünschten Werten füllen :

```
Node *head; //head pointer erstellen
Node *newNode; //pointer auf neuen Knoten erstellen
int num; //Wert
head = nullptr; //head zeigt am Anfang auf Nullelement,
     weil noch kein Knoten existiert

for (int i = 0; i < 3; i++) //3 Knoten {
  cout << "Enter number : ";
  cin >> num;
  newNode= new Node; //oder new Node(value, head)
  newNode->value = num;
  newNode->next = head;
  head = newNode;
}
```

17.3 Liste durchlaufen

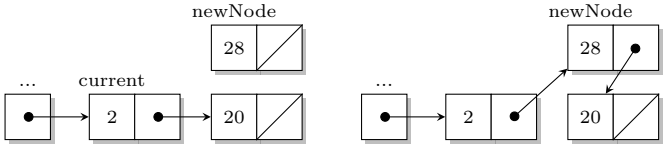
```
Node *current;
current = head; //Start
while (current != nullptr) {
  cout << current->value << " ";
  current = current->next; //naechster Knoten
}
```

17.4 Element finden

```
Node *current;
current = head; //Start
int element = 8; //Element das wir suchen
while (current != nullptr){
  if (current->value == element)
    break; // Schleife wird abgebrochen

  current = current->next ; // naechster Knoten
}
if (current = nullptr)
  std::cout<< "Element not found";
//Resultat : current zeigt auf gefundenes Element
```

17.5 Neuen Knoten einsetzen



Um einen neuen Knoten einzusetzen, benötigen wir einen pointer (`current`) auf den Knoten vor dem einzusetzenden Knoten.
Achtung wenn Liste noch leer ist!

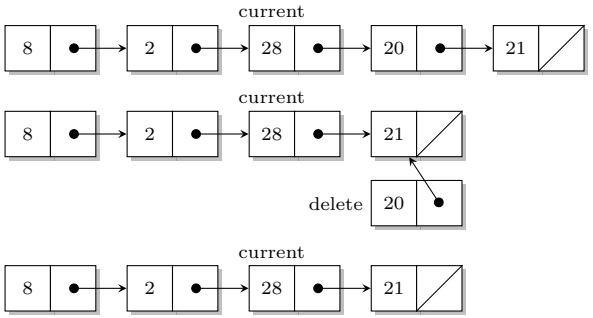
```
//neuer Knoten :
Node *newNode;
newNode = new Node;
newNode->value = 28;

 //Liste noch leer
if (head == nullptr)
{
newNode->next = nullptr;
head = newNode ;
}

//Else einsetzen :
newNode->next = current->next; //Wir haben den Code von
     "17.4 Element finden" benutzt!!
current->next = newNode;
```

17.6 Knoten löschen

Wir brauchen 2 pointer: `current` pointer auf den Knoten vor dem zu löschenden Knoten und ein auf den zu löschenden Knoten selbst.
Beispiel 1 - 2: element 20 löschen - alle elemente loschen




```
Node *deleteNode;
deleteNode = current->next;
current->next = deleteNode->next;
delete deleteNode;
////////////////////////////////////
Node *deleteNode = head;
while(head != nullptr){
  head = deleteNode->next;
  delete deleteNode;}
```