© 2001 Markus Rigling

Programmieren:

Klassen

2. Auflage

Inhaltsverzeichnis:

1. Unterschied Klassen/Strukturen 3
2. Deklaration 3
3. Definition 3
4. Klassenbenutzung in main 4
5. Sichtbarkeitsoperator :: 4
6. Konstruktor 4
7. Allgemeiner Konstruktor 5
8. Standardkonstruktor 5
9. Kopierkonstruktor 6
10. Typumwandlungskonstruktor 6
11. Initialisierungsliste 7
12. Destruktor 7
13. Zusammenfassung Konstruktoren / Destruktoren 8
14. Benutzung mit Zeigern: -> 8
15. This 9
16. Containerklassen 9
17. Inline Methoden 10
18. Static (Klassenattribute und –methoden) 11
19. Konstante Objekte/Methoden 12
20. Automatisch erzeugte Methoden 12
21. Konversionen (Typumwandlungen) 13
22. Typwandlungsoperator 14
23. Überladen von Funktionen 14
24. friend-Funktionen / Klassen 14
25. Überladen von Operatoren 15
26. Unäre Operatoren 15
27. Binäre Operatoren 15
28. Wertzuweisungsoperator 16
29. Indexoperator 16
30. Operator << 17
31. Übergabe von Parameter an Funktionen 17
32. Operator ( ) 17
33. Iteratoren 17
34. Zusammenfassung Klassen 18

1. Unterschied Klassen/Strukturen

Wenn man die Sichtbarkeit nicht angiebt, ist sie bei einer Klasse `private`, bei einer Struktur `public`.

2. Deklaration

```
#ifndef INTCOUNT_H
#define INTCOUNT_H

class IntegerCount {        //Deklariert neue Klasse IntegerCount
public:                     //in Public sind Methoden, auf die von
     void increment();      //aussen zugegriffen werden kann
     void decrement();
     void reset();
private:                    //in Private sind Attribute, auf die
     int value;             //von aussen nicht zugegriffen werden
     int resetValue;        //kann
};
#endif
```

3. Definition

```
#include "intcount.h"            //Deklaration ist intcount.h
void                             //Methode in Klasse IntegerCount
IntegerCount::increment()
{
     value = value + 1;
}

void
IntegerCount::decrement()
{
     value = value - 1;
}
```

4. Klassenbenutzung in main

#include “intcount.h”

```
int
main()
{
     IntegerCount aCount;     //erstelle Attribut aCount
     aCount.reset();          //Methode reset bei aCount
     return 0;
}
```

5. Sichtbarkeitsoperator ::

Zeigt, welcher Klasse die Methode angehört

```
Klassenname::Komponentenname       //Methode in Klasse

::Komponentenname                  //gleiche Methode ist in und
                                   //ausserhalb der Klasse.
                                   //Jetzt wird auf Methode
                                   //ausserhalb der Klasse
                                   //zugegriffen
```

Scope Resolution Operator (Sichtbarkeitsoperator) ist nicht nötig für:
Aufruf von Methoden für ein Objekt

- Aufruf von Methoden und Zugriff auf Attribute in Methoden der gleichen Klasse

6. Konstruktor

Ein Konstruktor ist eine spezielle Methode, deren Name gleich dem Klassennamen ist. Er hat einen Rückgabewert und kann überladen werden.

Deklaration:
Klassenname();
`Klassenname( Formalparameterliste );`

Definition:
`Klassenname::Klassenname()` *Block*
`Klassenname::Klassenname(` *Formalparameterliste* `)` *Block*

Wird automatisch aufgerufen, wenn ein Objekt der Klasse erzeugt wird

- Wird nicht für ein Objekt aufgerufen, denn er dient zum Erzeugen eines Objekts
- Wird benötigt, wenn beim Erzeugen eines Objekts mehr getan werden soll, als nur normalen Speicherplatz für die Attribute bereitzustellen, d.h. wenn etwa die Attribute initialisiert werden sollen oder wenn der Speicherplatz dynamisch angelegt werden soll.

Natürlich können dem Konstruktor auch Parameter zugefügt werden. Z.B.
`IntegerCount::IntegerCount(int theResetValue)`
So kann man den Reset-Wert z.B. gerade beim Erstellen des Attributs angeben.

7. Allgemeiner Konstruktor

- kann eine Formalparameterliste haben, so dass ihm Parameter mitgegeben werden können. Diese können die Initialisierung oder das dynamische Anlegen beeinflussen.

Klassenname Objektname(Parameterliste);

8. Standardkonstruktor

- Benötigt keine Parameter `IntegerCount();` oder
- Hat für alle Formalparameter Vorgabewerte
- Bei dynamischen Anlegen mit dem Operator `new` ohne Parameterangabe

In der Deklaration:

```
const int stacksize = 0;
Class IntegerCount {
Public:
     IntegerCount( int size = stacksize );   //mit Standartwert
...            //somit braucht es IntegerCount() nicht mehr
};
```

In der Definition:

```
IntegerCount::IntegerCount(int size)    //Standardkonstruktor
{
     resetValue = 0;          //setzt Attribut auf Null
     reset();                 //ruft Methode reset() auf
}
```

9. Kopierkonstruktor

- Falls zu kopierendes Objekt einen Pointer enthält, wird beim Kopieren der Pointer des neuen Objekts auf gleiche Adresse zeigen wie des alten Objekts. Dies kann unerwünscht sein, deshalb muss man einen eigenen Kopierkonstruktor schreiben, der für auf das gezeigte Attribut neuen Speicherplatz beschafft.
- Hat eine Funktion als Rückgabewert ein Objekt einer Klasse, so muss beim Ausführen von `return( Objekt );` ein neues Objekt erzeugt werden. Das neue Objekt wird dem rufenden Programm übergeben =>Kopierkonstruktor wird aufgerufen

Klassenname neuesObjekt = Kopiervorlage;
Hier wird das Objekt erzeugt und initialisiert. Dies hat **nichts** mit einer Wertzuweisung zu tun, denn bei einer Wertzuweisung wird nicht der Kopierkonstruktor verwendet, sondern der Wertzuweisungsoperator!
Deklaration:

```
Klasse( const Klasse& einObjekt);
```

Definition:

```
Klasse::Klasse( const Klasse& ein Objekt){ Block }
```

10. Typumwandlungskonstruktor

- Enthält Parameter eines anderen Typs, die bei der Initialisierung des erzeugten Objekts benutzt werden, um die Attribute mit Werten zu versorgen. (z.B von char zu Stack)

Typumwandlungen einer Klasse können in zwei Richtungen erfolgen:

- eine Variable eines anderen Typs soll in ein Objekt der Klasse gewandelt werden
- ein Objekt der Klasse soll in einen anderen Typ gewandelt werden (geht nur mit Typwandlungsoperator! siehe weiter unten)

```
Stack::Stack( const char string[] )
```

Wandelt ein char-Feld in Stack um. Im Hauptprogramm kann man den Typumwandlungskonstruktor wie folgt verwenden:
Stack meinStack4(„Paul“);

Die Definition könnte dann so umgesetzt werden:

```
Stack::Stack( const char string[] )
{
	maxLength = strlen( string );
	data = new char[maxLength];
	for ( int i=0; i < maxLength; i++) {
		data[i] = string[i];
	}
	topindex = i-1;
}
```

11. Initialisierungsliste

Ohne Initialisierungsliste wird beim Erstellen eines Objekts zuerst die Datenstruktur für die Attribute angelegt und dann ein Wert zugewiesen. Durch die Initialisierungsliste ist es aber möglich, die Attribute direkt beim Anlegen der Datenstruktur initialisieren zu lassen. Dies ist notwendig, wenn ein Attribut als `const` oder als Referenz deklariert ist, denn diese beiden Datentypen kann man nicht in einer Wertzuweisung verwenden.

Definition:
```
Klassenname::Klassenname( Formalparameterliste ) : Initialisierungsliste
{Block}
```

Wobei die Initialisierungsliste durch ein Komma getrennte Folge ist von
Attributname(Wert)

- Die Attribute einer Klasse werden genau in der Reihenfolge initialisiert wie sie in der Klassendeklaration deklariert werden. Die Reihenfolge in der Initialisierungsliste spielt dabei keine Rolle, es wird aber empfohlen, die Reihenfolge wie in der Deklaration beizubehalten.

Der Gebrauch der Initialisierungsliste wird dringend empfohlen, da
- konstante Attribute und Attribute vom Typ Referenz nur initialisiert werden können
- Die Initialisierung effektiver ist, da in bestimmten Fällen Attribute beim Anlegen immer initialisiert werden. Mit einer Wertzuweisung wird diese Initialisierung überschrieben. Wird gleich beim Anlegen (über die Initialisierungsliste) initialisiert, kann diese doppelte Arbeit gespart werden.

12. Destruktor

Ein Destruktor ist eine spezielle Methode, deren Name gleich dem Klassennamen mit vorangestelter ~ (Tilde) ist. Er hat keinen Rückgabewert. Er kann nicht überladen werden (keine Parameter!).

Deklaration:
~Klassenname();

Definition:
```
Klassenname::~Klassenname() {Block}
```

- Wird automatisch aufgerufen, wenn ein Objekt der Klasse vernichtet wird, d.h. wenn Variable ungültig wird oder bei dyn. mit `delete` gelöscht wird.
- Wird benötigt, wenn beim Erzeugen eines Objekts Speicherplatz dynamisch angelegt worden ist, der beim Vernichten wieder freigegeben werden muss.
- In jeder Klasse sollte einen Destruktor deklariert werden: wenn nichts besonderes getan werden muss, dann der Block leer bleiben.

13.Zusammenfassung Konstruktoren / Destruktoren

Konstruktor, usw.	Benutzung	Automatismen
Standardkonstruktor `( )` (Kein Parameter)	*Klasse* `x;` *Klasse* `fx[ 5 ];` *Klasse** `px = new` *Klasse*`[ 5 ];` Feld dynamisch erzeugen	automatisch erzeugt

Kopierkonstruktor
`( const` *Klasse*`& )`
	`foo(` *Klasse* `x ) { ... }`
	`return (` *Objekt* `);`
	Klasse `x = y;`
		automatisch aufgerufen
		automatisch aufgerufen
		automatisch erzeugt

Typwandlungskonstruktor `(` *andererTyp* `)`	*Klasse* `x(` *ParameterAnderenTyps* `);` `foo(` *ParamterAnderenTyps* `);`	automatisch aufgerufen
Allgemeiner Konstruktor `(` *Typ, ...* `)`	*Klasse* `x(` *Parameter,* `... );`	
Destruktor	wenn Objekt ungültig wird `delete px;` von Hand löschen, wenn ich Speicher vorher dynamisch erzeugt habe.	automatisch aufgerufen automatisch erzeugt
`operator=` `( const` *Klasse*`& )`	`y = x;`	automatisch erzeugt

Regel: Wann wird was benötigt?

- Für die Initialisierung von Objekten wird benötigt:
  - Standardkonstruktor und / oder
  - Typwandlungskonstruktor und / oder
  - Allgemeiner Konstruktor
- Wenn dynamische Attribute, d.h. mit `new` angelegte Attribute, verwendet werden, wird benötigt:
  - Standardkonstruktor und
  - Kopierkonstruktor und
  - Destruktor und
  - `operator=`

  (Alle 3 werden benötigt!)

(`operator=` wird später behandelt.)

14.Benutzung mit Zeigern: ->

```
zeigerAufObjekt->methodenname();
```

ist analog zu:

```
(*zeigerAufObjekt).methodenname();
```

15.This

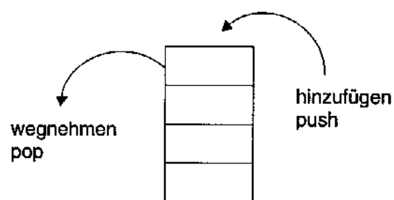
- This ist ein konstanter Pointer auf das Objekt, in dem ich mich befinde, während eine Methode aufgerufen wird
- Enthält als Wert die Adresse des Objekts. Wert kann nicht geändert werden
- `*this` ist das Objekt selbst
- `this->Komponentenname` ist die Komponente (Attribut oder Methode) des Objekts

16.Containerklassen

Beispiele von Containerklassen:

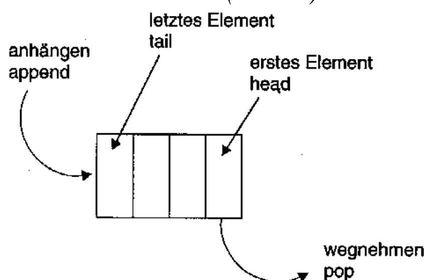
Stack:

Last-In-First-Out (LIFO)

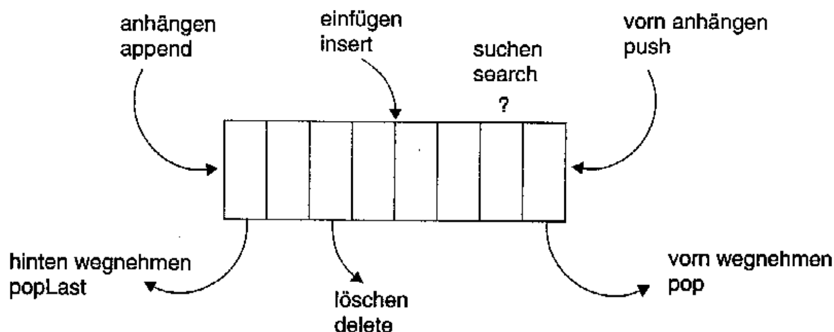


Queue:

First-In-First-Out (FIFO)



Liste:



Spezialfälle der Liste sind:

- **sortierte Liste**
  Bei dieser gibt es nur eine Methode zum Hinzufügern, welche das Element gemäss dem Sortierkriterium in die Liste einfügt. Wegnehmen kann man wie bei der allgemeinen Liste.
- **Zyklinsche Liste**
  Bei dieser ist der Sucessor von last first und umgekehrt
- **Assoziative Liste oder Dictionary**
  Bei dieser Liste enthält jedes Element zwei Objekte. Man kann wie in einem Wörterbuch über das eine der beiden andere finden.

Man kann Containerklassen nach mehreren Strukturkriterien klassifizieren:

Ungeordnet bedeutet, dass es keine definierte Reihenfolge der Elemente gibt. Es gibt kein erstes oder letztes Element, keine Reihenfolge. Zwei ungeordnete Mengen sind dann gleich, wenn jedes Element der einen Menge genau einmal in der anderen Menge vorkommt.

Geordnet bedeutet, dass es eine definierte Reihenfolge der Elemente gibt. Es gibt ein erstes oder letztes Element, Reihenfolge ist auch von Bedeutung. Zwei geordnete Mengen sind dann gleich, wenn jedes Element der einen Menge an der gleichen Stelle in der anderen Menge vorkommt.

Sortiert ist ein Spezialfall von geordnet und bedeutet ebenfalls, dass es eine definierte Reihenfolge der Elemente gibt. Diese Reihenfolge richtet sich jedoch nicht nach der Reihenfolge des Einfügens, sondern nach einem Sortierkriterium.

Einzigartigkeiten der Elemente

Beliebige Duplikate bedeutet, dass ein und dasselbe Objekt mehrmals in der Menge vorkommen darf.

Keine identischen Duplikate bedeutet, dass ein und dasselbe Objekt **nicht** mehrmals in der Menge vorkommen darf, verschiedene Objekte mit dem gleichen Wert jedoch vorkommen dürfen

Keine gleichen Duplikate bedeutet, dass Objekte mit gleichem Wert **nicht** mehrmals in der Menge vorkommen dürfen.

17. Inline Methoden

Da in der Objekt-Orientierten Programmierung alle Attribute als private deklariert werden sollen, ist der Zugriff der Attribute kontrolliert. Damit erfolgt der Zugriff nur über Methoden. Diese sind häufig sehr kurz. Der Aufruf der Methode kostet möglicherweise mehr Zeit als die eigentliche Ausführung. Um effizienter zu sein, bietet C++ die Inline Methoden oder inline functions an.
Sie werden vom Compiler anders als normale Funktionen verarbeitet. Wenn eine normale Funktion aufgerufen wird, so erfolgt ein Sprung in den Code dieser Funktion, am Ende erfolgt ein Rücksprung zum aufrufende Programm. Diese Sprünge sind der genannte Aufwand beim

Aufruf einer Funktion. Inline functions werden vom Compiler durch Einfügen des Funktionscodes an der Aufrufstelle realisiert. Anstelle des Funktionsaufrufs stehen im übersetzten Programm dann die ein oder zwei Zeilen Code der Funktion. Allerdings wird der Code umfangreicher, denn an jeder Aufrufstelle wird der Funktionscode hineinkopiert.

Definition:
inline Rückgabetyp
`Klassenname::Methodenname( Parameterliste ) Block`

Das Schlüsselwort `inline` ist nur eine Empfehlung an den Compiler, dieser kann aber eine zu lange oder zu komplizierte Funktion als normale Funktion bevorzugen, anstatt als inline.

Regeln der Verwendung von Inline Methoden:

- Man verwende für die Definition der Inline Methoden einer Klasse eine eigene Datei mit dem Namen: Klassenname.iC
- In der Deklarationsdatei der Klasse (Klassenname.h) includet man nach der Klassendeklaration diese Datei (vor #endif).
  Damit werden die Inline Methoden in alle Dateien kopiert, die die Klasse verwenden.
- In der definitionsdatei der Klasse (Klassenname.cpp) includet man die Deklarationsdatei und damit auch die Inline Methoden.

18. Static (Klassenattribute und –methoden)

Falls man für alle Objekte einer Klasse ein gemeinsames Attribut wünscht (z.B. Zinssatz beim Bankkonto – alle Kontos haben den gleichen Zinssatz), kann man mit `static` ein gemeinsames Attribut erstellen.
Klassenattribute müssen ausserhalb der Klasse und ausserhalb aller Funktionen vor der Erstellung des 1. Objekts initialisiert werden, am sinnvollsten in der Datei mit den Methodendefinitionen.
Im Beispiel der Bankkonten wäre eine Klassenmethode setzeZinssatz() notwendig, um den Zinssatz aller Sparkonten zu ändern – auch wenn es zur Zeit gar kein Objekt Sparkonto gibt.

Um den Zusammenhang mit der Klasse herzustellen, ist der Klassenname mit dem Sichtbarkeitsoperator dem Attributnamen voranzustellen:

Klassenname::Klassenattribut = Wert;

Eine Klassenmethode kann nur auf Klassenattribute, nicht jedoch auf Instanzattribute zugreifen. Eine Klassenmethode kann `this` nicht verwenden.

Statische Objekte entstehen durch:

- Deklaration ausserhalb von Blöcken: es sind also globale Variable, von deren Verwendung unbedingt abgeraten wird
- statische Variable in einer Funktion: eine solche Variable wird beim ersten Funktionsaufruf angelegt und überlebt den Rücksprung aus der Funktion, ist also beim nächsten Funktionsaufruf noch vorhanden
- Klassenattribute: diese sind ja unabhängig von Objekten der Klasse und werden in der Definitionsdatei der Klasse definiert und initialisiert

19.Konstante Objekte/Methoden

Objekte können wie Variable ebenfalls konstant definiert werden. Sie müssen bei der Definition initialisiert werden. Danach können sie nicht mehr geändert werden.
Methoden, welche die Attribute eines Objekts nicht ändern, sollten immer als konstante Methoden deklariert werden. Dies geschieht durch Verwendung des Schlüsselwortes `const`.
Für konstante Objekte können nur konstante Methoden aufgerufen werden.

Deklaration:
`Rückgabetyp Methodenname( Formalparameterliste )` **`const;`**

Definition:
`Klassenname::Methodenname(Formalparameterliste)` **`const`** {*Block*}

20.Automatisch erzeugte Methoden

Methode	wird automatisch erzeugt, wenn	muss selbst geschrieben werden, wenn
Standard-konstruktor	kein Konstruktor deklariert wird	• Objekte ohne Parameter erzeugt und initialisiert werden sollen, • Felder von Objekten initialisiert werden sollen, • ein anderer Konstruktor existiert und ein Standardkonstruktor benötigt wird (z.B. für Felder von Objekten)
Kopier-konstruktor	kein Kopierkonstruktor deklariert wird	• Klasse dynamisch erzeugte (new) Attribute hat
Destruktor	kein Destruktor deklariert wird	• Klasse dynamisch erzeugte (new) Attribute hat
Wertzuweisungs-operator (s. Kap. 3.8)	kein Wertzuweisungsoperator deklariert wird	• Klasse dynamisch erzeugte (new) Attribute hat

Methode	was die automatische Methode tut:
Standard-konstruktor	• legt Speicher für das Objekt an • ruft für Attribute (sofern diese selbst Objekte sind) deren Standardkonstruktor auf (ggf. den automatischen)
Kopier-konstruktor	• legt Speicher für das Objekt an • kopiert Attribute (sofern diese Objekte sind) mit deren Kopierkonstruktor (ggf. dem automatischen) • kopiert Attribute (sofern diese eingebaute Datentypen haben) bitweise
Destruktor	• ruft für Attribute (sofern diese Objekte sind) deren Destruktor auf (ggf. den automatischen) • gibt den Speicher des Objekt frei
Wertzuweisungs-operator	• ruft für Attribute (sofern diese Objekte sind) deren Wertzuweisungsoperator auf (ggf. den automatischen) • weist Attributen (sofern diese eingebaute Datentypen haben) ihren Wert durch bitweises Kopieren zu

- Wenn notwendig müssen die Methoden selbst geschrieben werden.
- Wenn kein Standardkonstruktor erzeugt werden soll, so muss irgendein Konstruktor geschrieben werden: irgendeinen Konstruktor muss es schliesslich geben!
- Wenn das Kopieren von Objekten oder deren Wertzuweisung verhindert werden soll, so deklariert man den Kopierkonstruktor bzw. Wertzuweisungsoperator als `private`, ohne sie zu definieren.

21. Konversionen (Typumwandlungen)

- werden in vielen Fällen automatisch durchgeführt

Solche Typumwandlungen können so geschrieben werden:

```
(Typname) Ausdruck
Typname( Ausdruck )
```

Beispiel:

```
int i; int j=5; float x = 6;
i = (int) ( (float)j + x );   //Erste Form

i =  int( float(j) + x );     //Zweite Form
```

22. Typwandlungsoperator

Die Wandlung eines Objekts einer Klasse in einen eingebauten Typ (Variable) kann nicht mittels Typwandlungskonstruktor erbracht werden, da das Hinzufügen eines Typwandlungskonstruktors in diesen Fällen nicht möglich ist: eingebaute Typen sind gar keine Klassen und können auch nicht geändert werden, letzteres gilt auch für eingebaute Klassen oder solche aus Klassenbibliotheken.
Die implizite Typumwandlung wird bei Wertzuweisungen, bei Parametern einer Funktion und bei der Rückgabe von Werten von Funktionen benutzt. (z.B. `Stack` in `char`)

Typumwandlungen sollten sparsam verwendet werden, denn sie unterlaufen die eigentlich erwünschte strenge Typprüfung.

23. Überladen von Funktionen

- unterscheiden sich im Typ ihrer Argumente
- Auswahl der richtigen Funktion durch den Compiler geschieht dementsprechend auf Grund dieses Typs
- Es werden, wenn nötig und möglich, Typumwandlungen durchgeführt.

Der Compiler sucht die aufzurufende Funktion nach folgender Regel:

- Benutze eine exakt passende Funktion, wenn vorhanden.
- Benutze die normalen Typkonversionen
- Benutze vom Benutzer definierte Typkonversionen

Die gefundene „beste“ Funktion muss dabei eindeutig sein.

24. friend-Funktionen / Klassen

friend ist ein Mechanismus, der das Verstecken der Attribute durchbricht. Sie gibt einer anderen Klasse Zugriff auf ihre internen Daten.
Es sollten in jedem Fall Alternativen zu friends geprüft werden.
Eine friend-Funktion muss in der Deklaration der Klasse, die den Zugriff gestatten will, erscheinen.

Die Deklaration einer Funktion zur friend-Funktion hat die Form:
friend Rückgabetyp Funktionsname(Parameterliste);

Die Deklaration einer Methode zur friend-funktion hat die Form:

```
friend Rückgabetyp
Klassennname::Funktionsname(Parameterliste);
```

Die Deklaration einer friend-Klasse hat die Form:
friend class Klassenname;

25. Überladen von Operatoren

Darunter versteht man die neue Definition von Operatoren wie +, -, *, / usw. Damit ist dann die gewohnte Schreibweise von Ausdrücken wie a+b auch für die Addition von Objekten einer selbstdefinierten Klasse möglich.

Deklaration:
Rückgabetyp operator@ (Formalparameterliste);
wobei @ für den entsprechenden Operator steht.

Funktionsart	Syntax mit Operator	Tatsächlicher Aufruf
Methode	`x @ y`	`x.operator@( y )`
	`@ x`	`x.operator@( )`
	`x @`	`x.operator@( )`
Funktion	`x @ y`	`operator@( x, y )`
	`@ x`	`operator@( x )`
	`x @`	`operator@( x )`

Vorgehensweise:

1. Was will ich haben?
   Beispiel: Aufruf aufschreiben v4 = -v3;
2. In Operatoren-Tabelle richtigen Eintrag auswählen
3. Deklaration für Operator@ aufschreiben
4. Realisierung für Operator@ aufschreiben

26. Unäre Operatoren

Unäre Operatoren sind Operatoren mit genau einem Argument, z.B. !, ++, - (unär, z.B. –9)

Deklaration:
Rückgabetyp operator@();
Die Parameterliste ist leer, da der Operator auf das Objekt selbst angewendet wird.

27. Binäre Operatoren

Man überlädt sie mittels einer Methode in der Form:

```
Rückgabetyp operator@( Formalparameter);
```

Bei einer Funktion:

```
Rückgabetyp operator@( Formalparameter1, Formalparameter2 );
```

28.Wertzuweisungsoperator

Falls kein eigener Wertzuweisungsoperator geschrieben wurde, verwendet C++ das komponentenweise kopieren, d.h. die Werte aller statischen Attribute werden vom einen Objekt ins andere kopiert.
Falls Attribute dynamisch angelegt werden, muss die Wertzuweisung selbst implementiert werden.
ACHTUNG! Falls einObjekt ein Pointer / Referenz enthält, zeigt das neue Objekt auf gleiche Speicherstelle! Falls ein Objekt geändert/gelöscht wird wirkt sich auch auf das neue Objekt aus!

Deklaration:
const Klassenname& operator=(const Klassenname& einObjekt);

Falls ein Objekt sich selbst zugewiesen wird, darf der Speicher nicht freigegeben und neu angelegt werden, da sonst nicht mehr kopiert werden kann. Dies ist ein häufiger Fehler, der schwer zu entdecken ist, da er unter Umständen erst irgendwann zur Laufzeit auftritt. Mit einer if-Anweisung kann man den Fehler ausschliessen.
Beispiel:

```
const Vektor& Vektor::operator=(const Vektor& v)
{
	if( this != &v ) {
		delete [] p;
		length = v.length;
		…
	}
	return *this;		//ist wichtig!
}
```

29.Indexoperator

Deklaration:
TypDesElements& operator[] (int index);

Deklaration für konstante Objekte:
TypDesElements operator[] (int i) const;

Bei mehrdimensionalen Feldern bietet sich der Operator ( ) an.

30. Operator <<

Der Operator<< wird im main() in folgender Form verwendet: `cout << xy;`

cout ist ein Objekt der Klasse ostream, **xy** ist ein Objekt der selbst erstellten Klasse.
Aus dessen Grund kann der `operator<<` nicht als Methode der auszugebenden Klasse realisiert werden.

Deklaration:
ostream& operator<<(ostream& out, const Klasse& Objektname);

Definition anhand Ausgabemethode print( ):

```
ostream& operator<<(ostream& out, const Vektor& v)
{
     v.print( out );
     return( out );
}
```

31. Übergabe von Parameter an Funktionen

Bisher haben wir immer Referenzen von Objekten der Funktion übergeben. Das Übergeben von ganzen Objekten als Parameter hat zum Nachteil, dass das ganze Objekt mit dem Kopierkonstruktor zuerst kopiert werden muss. Das ganze geht auch mit konstanten Objekten. Die beste Lösung ist die Übergabe von konstanten Referenzen, da diese nicht verändert werden dürfen.
Eingebaute Datentypen (float, int,...) übergibt man weiterhin ohne call-by-reference, da das Kopieren hier effektiver ist.

32. Operator ( )

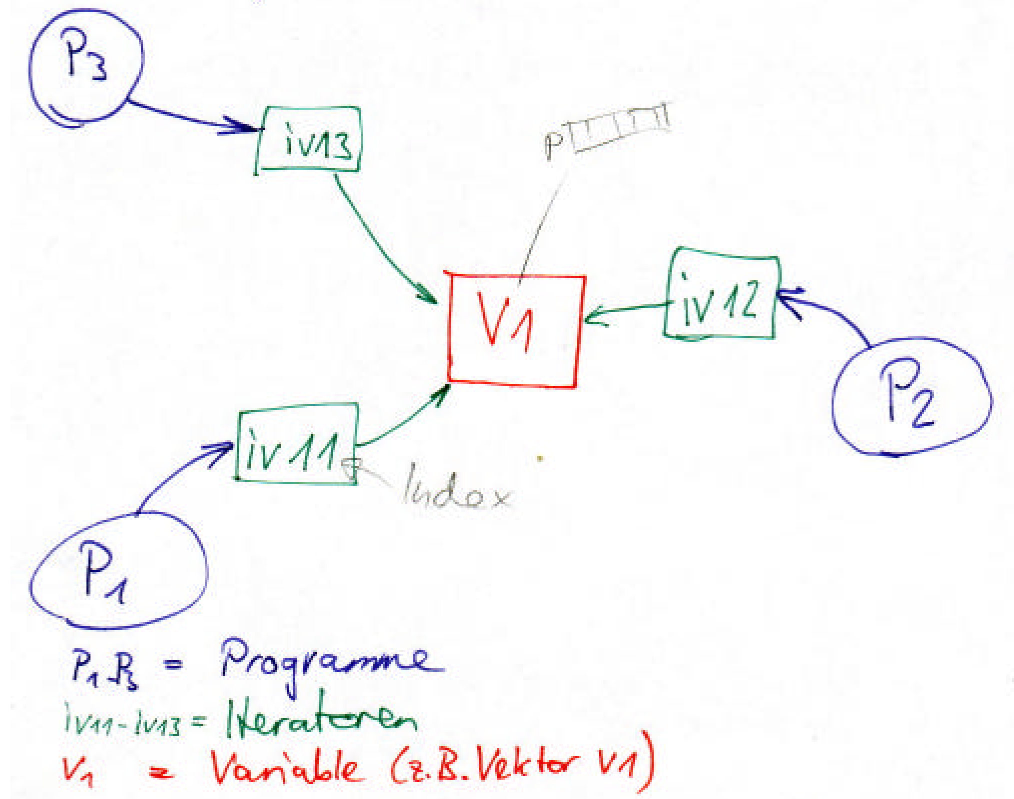
Der Operator ( ) kann ebenfalls überladen werden. Er wird hauptsächlich gebraucht für einen Indexzugriff auf die Elemente eines mehrdimensionales Feldes, da beim Operator [ ] nur ein Parameter erlaubt ist.

33. Iteratoren

Iteratorklasse wird für die Indexverwaltung einer Containerklasse gebraucht. Sie enthält ein Attribut `index`, eine Methode `first()` (welches den Index auf den Anfangswert setzt), `next()` (welche den Index auf das nächste Element setzt), `done()` (welche prüft, ob der Index den gültigen Bereich verlassen hat), und `current()`(welches Zugriff auf das aktuelle Element gibt).

Es wird eine Iterator-Klasse erstellt, damit sie auf viele Objekte der Klasse (z.B. Vektor) zugreifen kann. So wird die Iterator-Klasse als friend der Container-Klasse realisiert.

Mehrere Hauptprogramme (main) greifen auf die gleiche Variable v1 zu



Programme greifen nur indirekt (über den Iterator) auf die Variable zu.

34.Zusammenfassung Klassen

Für eine Klassenbasierte Programmierung sind folgende Elemente notwendig:

- die Verwendung von Klassen mit versteckten Attributen (`private`) und öffentlichen Methoden (`public`)
- das Verwenden von Konstruktoren und dem Destruktor
- das Überladen von Methoden
- das Überladen von Operatoren
- das Verwenden von friend-funktionen