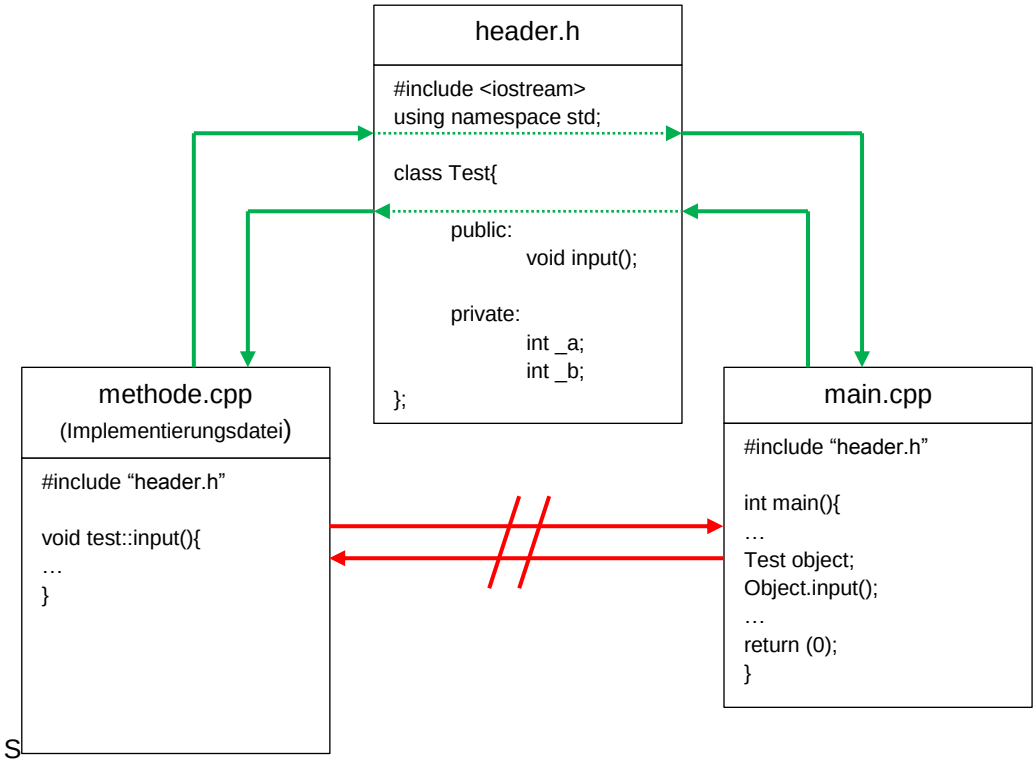
Objektorientiertes Programmieren C++

Link: http://de.cppreference.com/w/
Link: http://www.cplusplus.com/

INHALTSVERZEICHNIS

1. Ablauf beim Programm ausführen

Alles läuft direkt über die header.h Datei. Die main.cpp und die methode.cpp Datei können nie selbst untereinander Daten austauschen.



S

header.h

Im header werden die Klassen deklariert, Library’s integriert und das “using namespace std;“.

Da die header.h Datei mit dem Befehl *#include “header.h“ in der metode.cpp und main.cpp* die Schnittstelle für die Implementierung (methode.cpp) und das Programm (main.cpp) ist, kennen die zwei .cpp Dateien automatisch die integrierten library’s und das “using namespace std;“ der header.h Datei.

In der Klasse wiederum werden die Methoden (Funktionen), Konstruktoren, Destruktoren und lokalen Attribute (Variablen) deklariert

Wichtig: Eine header-Datei enthält niemals Code, somit kann die Datei nicht compiliert werden!

#ifndef CHESSBOARD_H
#define CHESSBOARD_H

Die zwei Flags #indef und @define sind um dem Main mitzuteilen, ob da Header Files bereits geladen wurd. Auf diese Weise kann die Perfomance erhöt werden, wenn das Programm nicht immer wieder die Header Files „lesen“ muss.

methode.cpp

Die Methoden (Funktionen), Konstruktoren und Destruktoren werden in einer

eigenen Implementierungsdatei implementiert.

main.cpp

In den Main- und anderen Programmdateien werden Objekte aus den Klassen erstellt und die dazugehörigen Methoden aufgerufen.

a = char
„Hallo“ = char*
„Hallo1“, „Hallo2“, … = char**

```
int main(int argc, char **argv) {
if (argc !3 3)
cout<<”Invalid number of PAs”<<endl;
return 0;

cout<<”argc: “<<argc<<endl;
for(int I = 0; I < argc; i++){
        cout<<argv[i]<<endl;

  return 0;
}
```

file.exe, para1, para2
argc = 3

argv	“file.exe“	argc1	“file.exe“
	“para1“	argc1	“para1“
	“para2“	argc1	“para2“

Im Parameter **argc** wird die Anzahl der Parameter gespeichert; der Name steht für **Argument Count**.

Im Parameter **argv** werden die einzelnen übergebenen Werte in einem **char**-Array gespeichert; der Name steht für **Argument Values**.

Nachdem man ein Programm kompiliert hat, kann man es über die Shell oder Eingabeaufforderung mit Parametern starten.

Objekt

Ein Objekt (Klasse) ist ein komplexer Datentyp gleich wie auch ein Integer ein Datentyp ist.
Komplex Datentyp weil er aus mehreren Datentypen beseht!

Datenkapselung

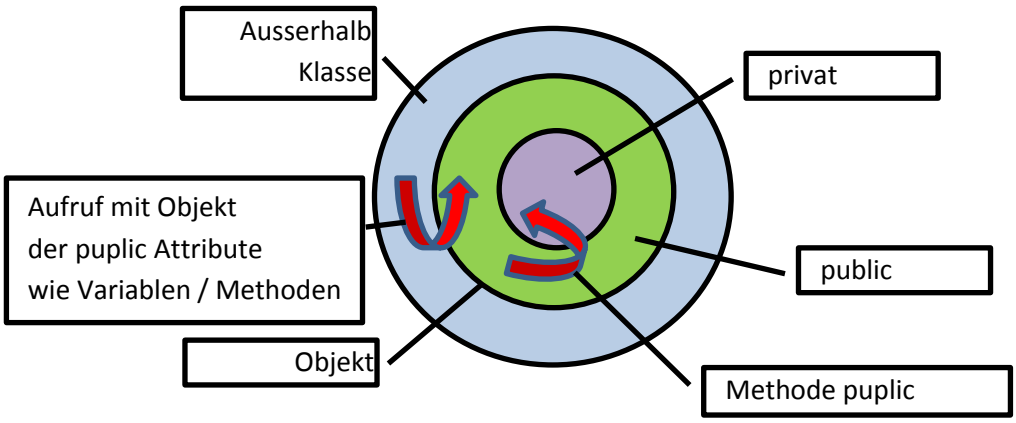
Die Datenkapselung beschreibt die drei folgenden Stufen / Levels welche bestehen:

- private
  Der Zugriff auf diese Attribute ist nur möglich wenn man sich während der Laufzeit in dem Objekt befindet! Dazu werden Public Methoden (=Funktionen) programmiert welche den Zugriff übernehmen.

- public
  Zugriff von ausserhalb der Klasse (aus z.Bsp. Main) möglich

- Ausserhalb der Klasse
  Grundsätzlich im Main oder einem anderen Code Teil der kein Objekt ist.

Eselsbrücke:	Mexikanerhut von oben gesehen



Konstruktoren

Mittels Konstruktoren werden die Attribute der Klasse entsprechend dem Konstruktor erstellt und initialisiert sobald im Hauptprogramm ein neues Objekt erstellt wird.

Wichtig: Ein Konstruktor immer wird automatisch ausgeführt, der Programmierer kann keinen Konstruktor aufrufen.

Merkmale:
- kann nicht selbst aufgerufen werden, wird vom System gemacht
- Methodenname = Klassennamen
- kein Rückgabewert

System Default Konstruktor:
Wenn der Programmierer selbst keinen Konstruktor erstellt, erstellt das System einen Konstruktor welcher die Attribute erstellt, dies macht das System selbst. Mit dem System Konstruktor werden die Attribute nicht mit einem Wert initialisiert (z. Bsp. x=0;) sondern es steht irgendein Wert im Speicherplatz, je nachdem was die Bits für einen Zustand haben.

Achtung: Sobald man selbst einen User Konstruktor macht wird der System Default Konstruktor nicht mehr ausgeführt. Wenn der User Konstruktor mit z.Bsp. zwei Variablen erstellt wird und man danach ein Objekt erzeugt ohne das Attribute übergeben werden, dann muss der „System Default Konstruktor" durch einen eigenen „User Default Konstruktor" ersetzt werden.

User Default Konstruktor:
Der Default Konstruktor wird vom Programmierer erstellt um die Attribute der Klasse zu erstellen und zu initialisieren. Sobald ein eigener Konstruktor erstellt wird, wird der „System Default Konstruktor" nicht mehr aufgerufen und man muss den „"User Default Konstruktor" selbst erstellen.
Sobald man ein Objekt erstellen muss, bei welchem die Attribute nicht initialisiert werden. Normalerweise wird im Default Konstruktor mit dem Wert 0 initialisiert.

Beispiel: Wenn ein neues Objekt „Bankkonto“ für einen neuen Kunden erstellt wird muss dieses leer sein.
Wenn ein neues Objekt „Bankkonto“ für einen bestehenden Kunde erstellt wird so muss diese mit einer Überweisung aus einem bestehenden anderen Konto direkt mit einem Betrag eröffnet werden.

User Konstruktor:
Wenn der User ein Objekt mit einem eigenen Konstruktor erstellen will so wird dieser Konstruktor entsprechend den Bedürfnissen definiert. Hierzu wird wie beim überladen von Funktionen z. Bsp. ein Konstruktor mit (double, double) erstellt und ein zweiter mit (int, int) für das Initialisieren der Variablen im neu erstellten Object.

Kopierkonstruktor:
Wenn ein neues Objekt als Kopie von einem bestehenden Objekt erstellt wird, erhält das neue kopierte Objekt die gleichen Attribute mit den gleichen Werten wie sie die Vorlage zu diesem Zeitpunkt hat.

Beispiel:

Zeitpunkt	**Objekt 1 (Vorlage)**	**Objekt 3 (Kopie von Vorlage)**

t1	*Objekt 1 wird erstellt*
	init a = 0;
	init b = 0;
		Objekt 3 existiert noch nicht

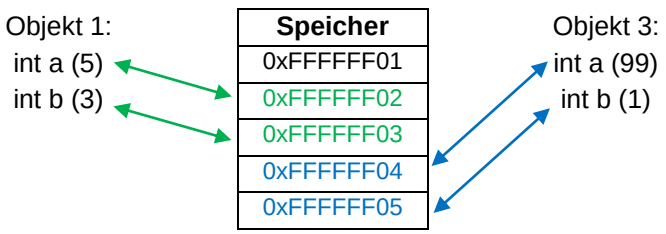
t2	*Objekt 1 hat folgende Werte:*
	int a = 5;
	int b = 3;
		Objekt 3 wird als Kopie von Objekt 1 erstellt:
		int a = 5;
		int b = 3;

t3	*Objekt 1 hat folgende Werte:*
	int a = 1;
	int b = 2;
		Objekt 3 hat folgende Werte:
		int a = 99;
		int b = 5;

t1 Objekt 1 wird mit Default Konstruktor erstellt, Objekt 3 gibt es noch nicht
t2 Objekt 3 wird als Kopie von Objekt 1 erstellt und die Attribute haben dieselben Werte
t3 Objekt 1 und Objekt 3 drei sind unabhängig voneinander bearbeitbar

Wichtig: Zu dem Zeitpunkt wo die Kopie erstellt wird, gibt es die Attribute Objekt 3 [int a, int b] noch nicht, heisst der Speicherplatz muss zuerst reserviert werden und dann initialisiert mit dem Wert von Objekt 1 [int a, int b].
Um dies zu erreichen wird mittels einer Referenz zuerst der Datentyp kopiert und erstellt und danach der aktuelle Wert dereferenziert aus dem Original gelesen und in die Kopie geschrieben.

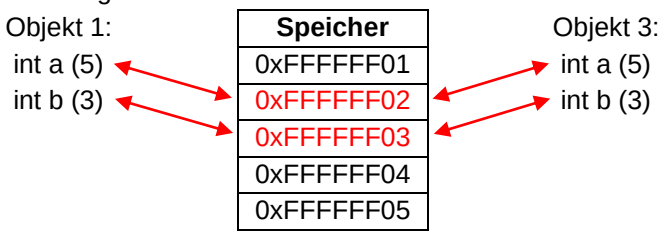
Kopie



Wichtig: Wenn nicht dereferenziert auf das Original zugegriffen wird, erstellt man einen Siamesischen Zwilling. Das heisst Objekt 1 und Objekt 3 teilen sich dieselben Attribute und beide lesen und schreiben die Attribute.

Die zwei Objekte haben immer dieselben Attributwerte! (Zwei Autos mit demselben Tank)

Siamesischer Zwilling



Dekonstruktor

Sobald Pointer in irgen einer Weise in einer Klasse vorkommen, so müssen diesen Speicher mit dem Syntax „new“ reserviert werden. Wenn das Objekt jetzt am Ende gelöscht wird muss auch dieser Speicher wieder freigegeben werden, was normalserweise mit „delete“ gemacht wird. Bei Klassen wird dies mit dem Dekonstruktor automatisch vom System gemacht.

Wichtig: Ein Konstruktor wird immer automatisch ausgeführt, der Programmierer kann keinen Konstruktor aufrufen.

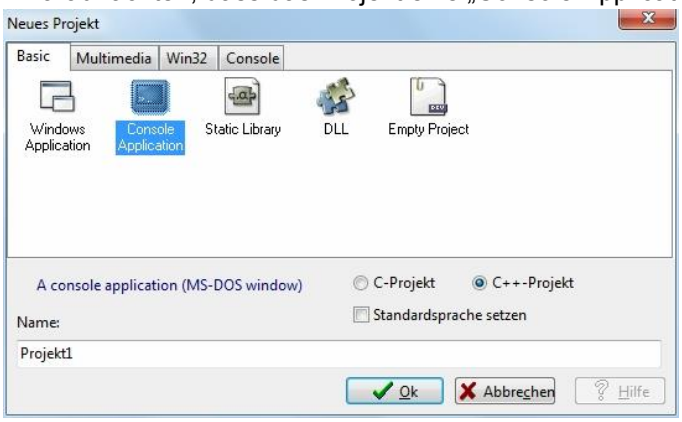
Merkmale:
- kann nicht selbst aufgerufen werden, wird vom System gemacht
- Methodenname = Klassennamen
- kein Rückgabewert
- enthält keine Parameter!

Wird dies nicht gemacht entstehen Speicherlöcher, sprich reservierter Speicher aber man „kennt“ die Adresse des Speichers nicht mehr weil der Pointer mit dem Objekt gelöscht wurde.

2. Erstellen eines neuen Projektes im Dev C++

1. Neues Projekt im Menü „Datei/Neu/Projekt“ auswählen.

2. Darauf achten, dass das Projekt eine „Console Application“ ist mit C++ Sprache.



3. Es müssen mindestens die folgenden drei Dateien Quelldateien gespeichert werden:
   main.cpp
   methode.cpp
   header.h

3. Beispiel Code

main.cpp

```
#include "header.h"

int main() {
int x (2), y (3), result (0);

Calc Objekt1;
   Objekt1.setValue(x, y);
   Objekt1.addValue();
   result = Objekt1.returnValue();
   cout<<result;

Calc Objekt 2(5,8);
   Objekt2.addValue();
   result = Objekt2.returnValue();
   cout<<result;

return 0;
}
```

header.h

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

class Calc {
     public:
          Calc();
          Calc(int a, int b);
          void setValue(int a, int b);
          void addValue();
          int returnValue();

     private:
          int _a;
          int _b;
          int _result;
};
```

method.cpp

```
#include "header.h"

// Default Konstruktor
Calc::Calc(){
   _a = 0;
   _b = 0;
   _result = 0;}

// User Konstruktor (int a, int b)
Calc::Calc(int a, int b){
          _a = a;
          _b = b;}

// Methode setValue
void Calc::setValue(int a, int b){
          _a = a;
          _b = b;}

// Methode addValue
void Calc::addValue(){
          _result = _a + _b;
          cout<<"a + b = "<<_result;}

// Methode Return
int Calc::returnValue(){
          return _result;}
```

Legende zu Quellcode

header.h

#include<iostream> using namespace std;	Alle Library's und "using namespace std" für das Projekt müssen in der header.h Datei implementiert werden.
#include "header.h"	Alle Quelldateien des Projekt habe über das implementieren der header.h Datei Zugriff auf alle Library's und „using namespace std"
class Calc {…}	class deklarieren der Klasse Clac Name der Klasse { } Reifenrumpf der Klasse
public:	Deklariert die öffentlichen Daten, heisst die Daten auf welche vom main.cpp etwas übergeben oder zurückgegeben werden kann. Wichtig: Um Daten vom main.cpp an private Variablen übergeben zu können werden öffentliche Funktionen deklariert.
Calc();	Deklarieren vom Default Konstruktor () heisst keine Daten werden für die Initialisierung (Nullen) beim Erstellen des Objekts übergeben
Calc(int a, int b);	Deklarieren vom User Konstruktor (int a, int b) heisst es werden Daten vom main.cpp für die Initialisierung (Grundwert stellen) beim Erstellen des Objekts übergeben
void setValue(int a, int b);	Deklarieren einer Funktion mit dem Namen „setValue" void heisst ohne Rückgabe (int a, int b) heisst es werden zwei Werte von Variablen aus der main.cpp an die Funktion übergeben
void addValue();	Deklarieren einer Funktion mit dem Namen „addValue" void heisst ohne Rückgabe () heisst es werden keine Werte von Variablen aus der main.cpp an die Funktion übergeben
int returnValue();	Deklarieren einer Funktion mit dem Namen „returnValue" int heisst es wird ein Wert vom Datentyp Integer zurückgegeben () heisst es werden keine Werte von Variablen aus der main.cpp an die Funktion übergeben
private:	Deklariert die privaten Daten, heisst die Daten auf welche nur innerhalb der Klasse etwas übergeben oder zurückgegeben werden kann.
double _a;	Deklarieren einer privaten Variabel
;	Ende der header.h Quelltext Wichtig: ohne das Semikolon wird ein Compiler Fehler generiert!
method.cpp	(Implementierungsdatei)
#include "header.h"	Alle Library's in der *header.h" sind automatisch durch den Verweis #include "header.h" automatisch auch in der methode.cpp verfügbar.

Calc::Calc(){
_a = 0;
_b = 0;
_result = 0;}
	Default Konstruktor um Werte mit 0 zu initialisieren.

Calc::Calc(int a, int b){
_a = a;
_b = b;}
	User Konstruktor um beim Erstellen eines neuen Objektes mit den Variablen (int a, int b) zu konstruieren.

void Calc::setValue(int a, int b){
_a = a;
_b = b;}
	Methode „setValue“ mit zwei Attribute die aus dem Main übergeben werden und ohne Rückgabe (void)

void Calc::addValue(){
_result = _a + _b;
cout<<"a + b = "<<_result;}
	Methode „addValue“ ohne Attribute die aus dem Main übergeben werden und ohne Rückgabe (void)

int Calc::returnValue(){ return _result;}	Methode „returnValue“ ohne Attribute die aus dem Main übergeben werden aber mit einer Rückgabe integer

main.cpp

#include "header.h"	Alle Library’s in der *header.h“ sind automatisch durch den Verweis #include "header.h" automatisch auch in der methode.cpp verfügbar.
Calc Objekt1;	Erzeugen vom Objekt “Objekt1” nach der Vorlage “Calc”
Objekt1.setValue(x, y);	Methode (Funktion) “setValue” ausführen für das Objekt “Objekt1”
Objekt1.addValue();	Methode (Funktion) “addValue” ausführen für das Objekt “Objekt1”
result = Objekt1.returnValue();	Methode (Funktion) “returnValue” ausführen für das Objekt “Objekt1” und die Variable „result“ mit dem Rüchgabewert aus der Methode Return initialisieren.

4. Zugriff auf Klasse aus Main

Es kann nicht direkt vom Main auf die Attribute vom Objekt zugegriffen werden, weil diese als „privat“ deklariert sind. Um trotzdem auf die Objekte zugreifen zu können werden Methoden (Funktionen) als „puplic“ implementiert welche innerhalb der Klasse dann auf die „privat“ Attribute zugreifen können.

In der folgenden Tabelle sind ein paar Beispiele in rot wie diese Methoden aussehen können:

main.cpp Zugriff	**header.h** Implementierung & Initialisierung	**methode.cpp** Methode	**Bemerkung**
main () { int a (0), b (0);	class Bsp { puplic:		

Obj1.setPoint (x, y);	void setPoint (int q, int r);	void Bsp::setPoint (int q, int r) {
		…
		}	

Obj2.showPoint ();	void showPoint ();	void Bsp::showPoint () {
		…
		}	

Obj1.getValue ();	void getValue ();	void Bsp::getValue () {
		…
		}	

Obj4.setValue (a, b);	void setValue (int q, int r);	void Bsp::setValue (int q, int r) {
		…
		}	

Obj7.addValue (5,6);	void addValue ();	void Bsp::addValue () {
		…
		}
			Die Zahl 5 und sechs werden an die Methode übergeben.

Obj99.returnValue ();	int returnValue ();	int Bsp::returnValue () {
		…
		return x; // int Variable
		}	

Obj81.statusValue (a, b);	bit statusValue (q, r);	bit Bsp::stausValue int q, int r) {
		…
		return b1; // bool Variable
		}	

Obj5.refValue (a, b) ;	void refValue (int &q, int &r);	void Bsp::refValue (int &q, int &r({
		_q = q;
		_r = r;
		}	

Obj5.poiValue (a, b) ;	void poiValue (int x, int y);	void Bsp::poiValue (int x, int y) {
		*_x = x ; // Wert aus x deref. in _x
		*_y = y ; // Wert aus y deref. in _y
		}
			Die Pointer _x und _y müssen beim Erstellen des Objektes „Obj5“ mit dem Default Konstruktor zuerst wie folgt erstellt und initialisiert damit der Code links funktioniert:
			_x = new int (0);
			_y = new int (0);

Obj33.getPointerx ();	int getPointerx () ;	int Bsp::getPointerx (){
		return *_x;
		}	

return 0; }	private: int _q; int _r ; int *_x; int *_y; };		

5. Default & User Konstruktoren

main.cpp

```
#include "header.h"

int main() {
int x (5), y (0);
double z (5);
// Obj.1 mit Default Konstruktor erstellen
testPointer Object1;
    // Wichtig: Wenn Objekt1 mit Default
    Konstruktor erstellt werden soll, dann
    darf es am Ende KEINE Klammern
    haben!

// Obj.2 mit User Konstruktor erstellen
testPointer Object2 (x, y);



// Obj.3 mit User Konstruktor erstellen
testPointer Object3 (z, x);


}
```

header.h

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

class testPointer{

    public:
        // User Default Konstruktor
        testPointer ();


        // User Konstruktor (int a, int b)
        testPointer (int a, int b);


        // User Konstruktor (double a, int b)
        testPointer (double a, int b);


        //  Methode für Objekte welche mit
        dem Default Konstruktor () erstellt
        wurden:
        void methDefault (int a, int b);


        Methode für Objekte welche mit
        dem. User Konst. (int, int) und
         (double, int) erstellt   wurden:
        void methUser ();


    private:
        int _a;
        int _b;
};
```

method.cpp

```
#include "header.h"



// User Default Konstruktor
testPointer:: testPointer (){
    _a = 0; // erstellen und initialisieren
    _b = 0: // erstellen und initialisieren
}


// User Konstruktor (int a, int b)
testPointer:: testPointer (int a, int b){
    _a= a; // main x in Object2._a schreiben
    _b= b; // main y in Object2._b schreiben
}


// User Konstruktor (double a, int b)
testPointer:: testPointer (double a, int b){
    _a= (int) a; // main z in Object3._a casten
    _b= b; // main x in Object3._b schreiben
}


//  Methode für Default Konstruktor
void testPointer::methDefault (int a, int b){
    _a = a;
    _b = b;
}


//  Methode für User Konstruktor
void testPointer::methUser(){
    _b = _a  * 2;
    cout<< _b<<" = "<<_a<<" x 2";
}
```

System Default Konstruktor = deklarieren aller Variablen aus der Klasse aber ohne initialisieren
wird vom System zur Verfügung gestellt aber nur solange vom User kein eigener Konstruktor deklariert wird!

User Default Konstruktor = deklarieren aller Variablen aus der Klasse mit initialisieren
muss durch den User erstellt werden sobald ein Konstruktor selbst gemacht wird. Es werden keine Werte aus dem Main übergeben sondern normalerweise nur intern die Variablen aus der Klasse mit 0 / NULL initialisiert.

6. Kopierkonstruktor

Das System stellt einen Default Kopierkonstruktor zur Verfügung aber sobald mit dynamischen Attributen oder mit Zähler in Objekten (Klassenvariablen) gearbeitet wird, muss der Kopierkonstruktor selbst aufgebaut werden.

Mittels einer Referenzvariabel welche die Adresse des zu Kopierenden Objektes „Objekt3“ enthält, wird es möglich im Kopierkonstruktoren direkt auf Werte des zu kopierenden Objektes „Objekt1) zu zugreifen. Diese Referenz ist als konstant zu deklarieren und gehört zur aufzurufenden Klasse vom Objekte. Dies wird mittels des Punktoperators gemacht.

Kopierkonstruktor OHNE Pointer

main.cpp

```
#include "header.h"

int main() {
int x (5), y (1)

// erstellen Object1 mit Wert 5 und 1
testKonst Object1(x, y);



// erstellen Object3, initialisiert mit Wert 5
und 1 von Object 1
testKonst Object3(Object1);


return 0;
}
```

header.h

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

class testKonst {

      public:
            // Deklaration Default Konst.
            testKonst (int a, int b);



            // Deklaration Copy  Konst.
            testKonst (const testKonst &Ref);



      private:
            int _a;
            int _b;
};
```

method.cpp

```
#include "header.h"



// Default Konstruktor
testKonst:: testKonst (int a, int b){
      _a = a;
      _b = b;
}

// Kopier Konstruktor OHNE Pointer
testKonst:: testKonst (const testKonst &Ref){
      _a = Ref._a;   //_a aus Object1 in _a
                     // Object3 kopieren
      _b = Ref._b;   //_b aus Object1 in _b
                     // Object3 kopieren
}
```

Kopierkonstruktor MIT Pointer

main.cpp

```
#include "header.h"

int main() {
int x (5), y (1)

// erstellen Object1 mit Wert 5 und 1
testKonst Object1(x, y);



// erstellen Object3, initialisiert mit Wert 5
und 1 von Object 1
testKonst Object3(Object1)

return 0;
}
```

header.h

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

class testKonst {

      public:
            // Deklaration Default Konst.
            testKonst (int a, int b);



            // Deklaration Copy  Konst. Pointer
            testKonst (const testKonst &Ref);



      private:
            int _a;
            int *_c;
};
```

method.cpp

```
#include "header.h"



// Default Konstruktor
testKonst:: testKonst (int a, int b){
      _a = a;
      _c = new int (); // Speicher reservieren
      *_c = 0; // deref. in Speicher schreiben
}

// Kopier Konstruktor MIT Pointer
testKonst:: testKonst (const testKonst &Ref){
      _a = Ref._a;   //_a aus Object1 in _a
                     // Object3 kopieren
      _c = new ();   // Speicher reservieren
      *_c = *Ref._c    //
}
```

7. Referenzen Objektorientiert

Mittels einer Referenz wird beim Aufruf einer Methode die Adresse zu der Variabel aus dem Main erstellt. So wird es möglich mit einer Referenz direkt die oder mehrere Variabel(n) aus dem Main mit einer Methode zu lesen oder zu schreiben.

Wert mit Referenzen aus Objekt in Main schreiben:

main.cpp	header.h	methode.cpp

#include "header2.h"
 	
main (){
  int dummy = 69;
 	
  // Object erzeugen
  test Objekt1;
 	
  // Wert per Ref. an Main
  Objekt1.methode2main(dummy);
  cout<<dummy;
 	
return 0;
}
	#include <iostream>
	
	using namespace std;
	
	class test{
	  public:
	  test ();
	  void methode2main (int &r_variable);
	
	  private:
	  int _a;
	};
		#include "header2.h"
		
		// Default Konstruktor
		test::test (){
		  _a = 0;
		}
		
		// Methode
		void test:: methode2main (int &r_variable){
		  r_variable = _a;
		}

1. „r_variable" referenziert von Variabel „dummy".
2. Somit wird Zufallszahl aus der Operation rand()% 100 …
3. … direkt in Main-Variable „dummy" geschrieben.

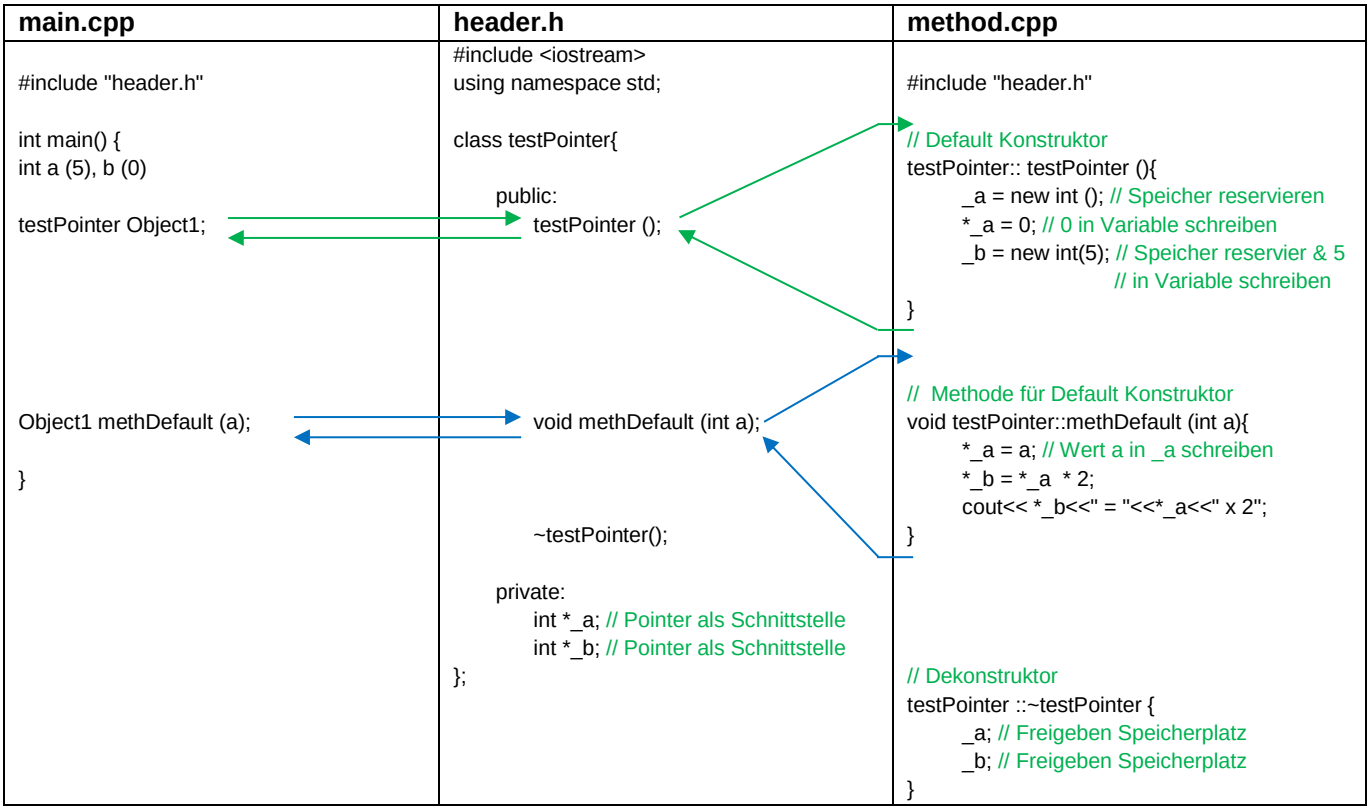
Wert mit Referenzen aus Main in Objekt schreiben:

main.cpp	header.h	methode.cpp

#include "header2.h"
 	
main (){
  int dummy = 69;
 	
  // Object erzeugen
  test Objekt1;
 	
  // Wert per Ref. an Methode
  Objekt1.main2methode(dummy);
 	
return 0;
}
	#include <iostream>
	
	using namespace std;
	
	class test{
	  public:
	  test ();
	  void main2methode (int &r_variable);
	
	  private:
	  int _a;
	};
		#include "header2.h"
		
		// Default Konstruktor
		test::test (){
		  _a = 0;
		}
		
		// Methode
		void test:: main2methode (int &r_variable){
		  _a = r_variable;
		}

1. „r_variable" referenziert auf Adresse von Variabel „dummy".
2. Somit wird Wert 69 aus der Main-Variabel „dummy" …
3. … direkt in Objektvariabel „_a" von „Objekt1" geschrieben.

8. Pointer Objektorientiert



Wenn das Object1 im Main erstellt wird, wird der Default Konstruktor aufgerufen und die Variablen wie folgt beschrieben erstellt und initialisiert:

Beispiel 1:

```
_a = new int ();    // Es wird eine Variable mit dem Namen _a erstellt mit der dazugehörigen Adresse
*_a = 0;            // Der Wert 0 wird dereferenziert im Speicherplatz der Variable _a gespeichert
```

Beispiel 2:

```
_b = new int (5);   // Es wird eine Variable mit dem Namen _b erstellt mit der dazugehörigen Adresse und
                    // gleichzeitig wird der Wert 5 in den Speicherplatz geschrieben
```

Wichtig: Die Pointer _a und _b zeigen auf den Speicherplatz welcher ihnen durch den Befehl new zugeteilt wurde. Wenn man Wert aus den zwei lesen oder schreiben will muss dies dereferenziert geschehen!

Resultat = *_a + *_b

9. Array Objektorientiert (dynamischer Speicher)

Beim Erstellen eines dynamischen Array's müssen immer folgende Elemente vorhanden sein:

- Pointer Variable
- Konstruktor
- Dekonstruktor

main.cpp

```
#include "header.h"

main () {
int sizeArray (5), valueArray (0), fieldArray (0);

// Objekt erstellen
dynArray Objekt1 (5); // Array mit 5 Felder


// Lesen Array Feld im Objekt
?

//Schreiben Array Feld im Objekt
?

return 0;
}
```

header.h

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

class dynArray{

    public:
        // Deklaration Default Konst.
        dynArray(int size);

        // Deklaration Dekonst.
        ~dynArray();

        // Methode lesen Array Felder
        ?

        // Methode schreiben Array Felder
        ?

     private:
        int *_pa; // Pointer für Array
        const int sizeArray;
};
```

method.cpp

```
#include "header.h"

// Default Konstruktor mit Initialisierungsliste
dynArray:: dynArray (int size): sizeArray (size){

        // reservieren Speicher und mit size init.
        _pa = new int[size];

        // Alle Felder nullen
         for(int i = 0; i < size; i++){
                _pa[i] = 0;
         }
}

// Dekonstruktor
dynArray::~ dynArray (){
        delete [ ] _pa;
}

// Methode lesen Array Felder
?

// Methode schreiben Array Felder
?
```

Die Grösse eines statischen Array kann man wie folgt ermitteln:

```
int testArray [99];
int sizeArray =  sizeof(testArray)/sizeof (testArray [0]);
```

Bei einem dynamischen Array ist dies NICHT möglich, den die Berechnung gibt nur die Grösse der Pointer Variable (= 4Byte) zurück.

Wenn diese Funktion gebraucht wird sollte mit Vektoren gearbeitet werden, dort wird bereits eine Funktion für die Grösse zur Verfügung gestellt.

10. Pointer auf Klasse

(dynamisches Objekt)

main.cpp

```
#include “header.h”

main () {
int z1(0),z2(0);

// Erstellen eines neuen Objektes
pointerKlasse *a1;
a1 = new pointerKlasse ();
a1 -> setValues (z1,z2); // Werte 0 werden
                            gesetzt
(*a1). setValues (11,69); // Werte 11 und
                            69 werden
                            gesetzt


// Erstellen mehreren neuen Objekten
pointerKlasse *a2;
a2 = new pointerKlasse [2];
a2[1]. setValues (z1,z2); // Werte 0 vom
                            Objekt in
                            Feld 2
                            werden
                            gesetzt


return 0;
}
```

header.h

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

class pointerKlasse {

      public:
            void setValues(int x, int y);
            void showValues();

      private:
            int _a;
            int _b;
};
```

method.cpp

```
#include “header.h”

void pointerKlasse::setValues(int x, int y){
  _a = x;
  _b = y;
}
```

Analyse zum Code:

Code	Erklärung
*pointerKlasse *a1*	*Speicherplatz reservieren für ein Objekt mit dem Namen a1 der Klasse „pointerKlasse“, Der benötigte Speicherplatz ist so gross, dass alle „Infos“ für eine komplette Adressierung des ganzen Objektes mit allen Attributen Platz hat.*
a1 = new pointerKlasse ();	*Erstellen des Objekt a1 aus der Klasse „pointerKlasse“. Zu diesem Zeitpunkt werden alle zusätzlichen Links im Hintergrund vom System für alle Attribute des Objekt a1 erzeugt*
a1 -> setValues (z1,z2);	*Zugriff dereferenziert auf das Objekt mit Pfeiloperator*
*(*a1). setValues (z1,z2);*	*Zugriff dereferenziert auf das Objekt „nach alter Art“ mit Punktoperator*
*pointerKlasse *a2*	*Speicherplatz reservieren für ein Objekt mit dem Namen a2 der Klasse „pointerKlasse“, Der benötigte Speicherplatz ist so gross, dass alle „Infos“ für eine komplette Adressierung des ganzen Objektes mit allen Attributen Platz hat.*
a2 = new pointerKlasse [2];	*Erstellen eines Array welches zwei Objekte a2[0] und a2[1] aus der Klasse „pointerKlasse“.enthält Zu diesem Zeitpunkt werden alle zusätzlichen Links im Hintergrund vom System zu den Attributen des Objekt a2 (= Array mit 2 Feldern) erzeugt.*
a2[1]. setValues (z1,z2);	*Zugriff dereferenziert auf das Objekt mit Pfeiloperator*

11. Initialisierungsliste

(Skript 6.4 und S.82)

Ein konstanter Werte und eine Referenzen müssen beim Zeitpunkt bei dem sie erstellt werden (deklarieren) gleichzeitig auch initialisiert werden, ansonsten gibt es einen Compilerfehler.
Damit dies beim Erstellen einer neuen Klassen möglich wird, muss beim User Default Konstruktor vor dem Eintritt in den Konstruktor mittels der Initialisierungsliste die Konstanten und Referenzen deklariert und initialisiert werden!

Beispiel Code:

initList:: initList()	Normaler Aufruf des User Default Konstruktor im der Methoden- / Implementierung Datei
: _c (0), _d (a)	Initialisierungsliste
{	Starten des Konstruktor
….	Deklarieren und initialisieren aller anderen Variablen des neuen Objektes.
}	Beenden des Konstruktor

main.cpp	**header.h**	**method.cpp**

#include “header.h”
 	
main () {
int a= 99
 	
initList neuesObjekt1 (a);
 	
return 0;
}
	#include<iostream>
	using namespace std;
	
	class initList {
	
	public:
	initList (int x);
	~ initList ();
	
	private:
	const double _c; // Konstanter Wert
	int &_d; // Referenz
	int *_a; // Pointer
	int _b; // Ganzzahl
	};
		#include “header.h”
		
		// User Konstruktor
		initList:: initList (int x)
		: _c (3.1459), _d (x) {
		_a = new int (77);
		_b = 66;
		}
		
		// Dekonstruktor
		testClass::~testClass(){
		delete _a;
		}

Wichtig: Die Werte werden entsprechend der Deklarierungsreihenfolge in Klasse übergeben.
Beispiel:

main.cpp	header.h	method.cpp
initList neuesObjekt1 (9);	private: int _a; int _b;	initList:: initList (int a) : _b (a), _a (a) {…..}

Es wird zuerst die Variable _a mit 9 initialisiert und dann die Variable _b mit 9!

Analyse zum Code in Datei „methode.cpp“:

Code	Erklärung
initList:: initList (int a)	Aufrufen des User Konstruktor mit dem Wert 99 der von Main übergeben wurde
:	Start Initialisierungsliste, heisst Variablen nach : werden deklariert und initialisiert
_c (3.1459)	Konstanter Wert wird deklariert und mit Pi initialisiert
,_d (x)	Die Refernzvariabel _d im neuenObjekt1 wird mit der Adresse von Variabel int a aus Main initialisiert. Achtung: Wenn int x in einer weiteren Methode neu geschrieben wird, so ändert sich auch der Wert der mit cout<<_d; ausgegeben wird, weil die Adresse von _d auf int x zeigt.
{	Ende Initialisierungsliste, Starten des Konstruktors
_a = new int (77);	Speicherplatz für Pointer reservieren und mit dem Wert 77 initialisieren
_b = 66;	Variabel 66 mit dem Wert 66 initialisieren
}	Beenden des Konstruktor

12. Initialisierungsliste bei Kopierkonstruktoren

main.cpp	header.h	method.cpp

#include “header.h”
 	
main () {
int a = 99
 	
initList Objekt1 (a);
initList Objekt2(Objekt1, a):
 	
return 0;
}
	#include<iostream>
	using namespace std;
	
	class initList {
	
	public:
	initList (int x);
	//initList (const initList &rCopy);
	initList (const initList &rCopy, int x);
	~ initList ();
	
	private:
	const double _c; // Konstanter Wert
	int &_d; // Referenz
	int *_a; // Pointer
	int _b; // Ganzzahl
	};
		#include “header.h”
		
		// User Konstruktor
		initList:: initList (int x)
		: _c (3.1459), _d (x) {
		_a = new int (77);
		_b = 66;
		}
		
		// Kopierkonstruktor
		initList:: initList (const testClass &rCopy, int x)
		//: _c (rCopy._c), _d (rCopy._d){
		//: _c (rCopy._c), _d(x){
		_a = new int ();
		*_a = *rCopy._a
		_b = rCopy._b;
		}
		
		// Dekonstruktor
		testClass::~testClass(){
		delete _a;
		}

Analyse zum Code in Datei „methode.cpp“:

Code	Erklärung
(const testClass &rCopy)	Übergeben der Adresse vom Objekt1 in der Referenzvariabel rCopy
: _c (rCopy._c), _d (rCopy._d)	Weil es sich bei diesem Beispiel um eine Referenz auf eine Referenz handelt funktioniert das nicht. Es wird zwar der Speicherbereich zugeordnet aber der Wert der bestehenden Referenz wird nicht in die neue Variabel übertragen, dies muss selbst programmiert werden. Grund kann sein, dass es nicht wirklich Sinn macht eine Referenz in einem Objekt zu haben und diese zu kopieren.
:	Start Initialisierungsliste, heisst Variablen nach : werden deklariert und initialisiert
_c (rCopy._c)	Die Referenz rCopy enthält die Adresse von Objekt 1. Über Referenz rCopy kann somit auf Objekt1, Variable _c zugegriffen werden und der Wert in Objekt2, Variabel _c kopiert werden.
, _d (x)	Der Referenz Variabel _d wird mit der Adresse von int a von Main initialisiert.
{	Ende Initialisierungsliste, Starten des Kopierkonstruktor
_a = new int ();	Neuen Speicherplatz für Objekt2, Variable Pointer _a reservieren
*_a = *rCopy._a	Die Referenz rCopy enthält die Adresse von Objekt 1. Über Referenz rCopy kann somit auf Objekt1, Variable _a zugegriffen werden und der Wert in Objekt2, Variabel _a kopiert werden. Achtung: Da es sich um einen Point handelt muss dereferenziert werden.
_b = rCopy._b;	Die Referenz rCopy enthält die Adresse von Objekt 1. Über Referenz rCopy kann somit auf Objekt1, Variable _b zugegriffen werden und der Wert in Objekt2, Variabel _b kopiert werden.
}	Beenden des Kopierkonstruktor

13. “This” Pointer

(Skript S.85 & 88)

Ein „this“ Zeiger enthält die Adresse vom Objekt das aufgerufen wird. Somit ist es möglich mit dem „this“ Zeiger auf die Adresse eines Objektes zu zeigen und damit wird es möglich zweimal denselben Variablennamen zu verwenden. Im Main wird dann jeweils das Objekt 1 bis n aufgerufen und der „this“ Zeiger aus der Methode zeigt auf die Adresse des aufzurufenden Objektes 1 bis n. Somit referenziert die Methode immer auf die private Variabel vom Objekt welches die Methode aufruft.

main.cpp

```
#include “header.h”

main () {

thisPointer Objekt1 (99);

return 0;
}
```

header.h

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

class thisPointer {

      public:
          thisPointer (int); //Konstruktor


       private:
            _a // alt
            a // neu

};
```

method.cpp

```
#include “header.h”

//alt
thisPointer::thisPointer (int a){
        _a = a;
}

//neu
thisPointer::thisPointer (int a){
        this->a = a;
}
```

14. Konstante Methoden

(Skript 6.5)

Wenn ein Objekt1 bis n eine Konstante Methode aufruft, kann die aufgerufene Konstante Methode die Attribute des Objektes nicht verändern sondern nur lesen.
Dieser Umstand kann verwendet werden wenn Bsp. eine getMethode definiert wird denn somit wird verhindert dass die Attribute des Objektes mit einer getMethode verändert werden kann.

Gleichzeitig ist es ein Performance Vorteil.

main.cpp

```
#include “header.h”

main () {

constMethode Objekt1;

Objekt1.setValue (5);

Objekt1.getValue();

return 0;
}
```

header.h

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


class constMethode {

      public:
            constMethode ();
            void setValue(int);
            …
            void getValue() const;

      private:
            a;
};
```

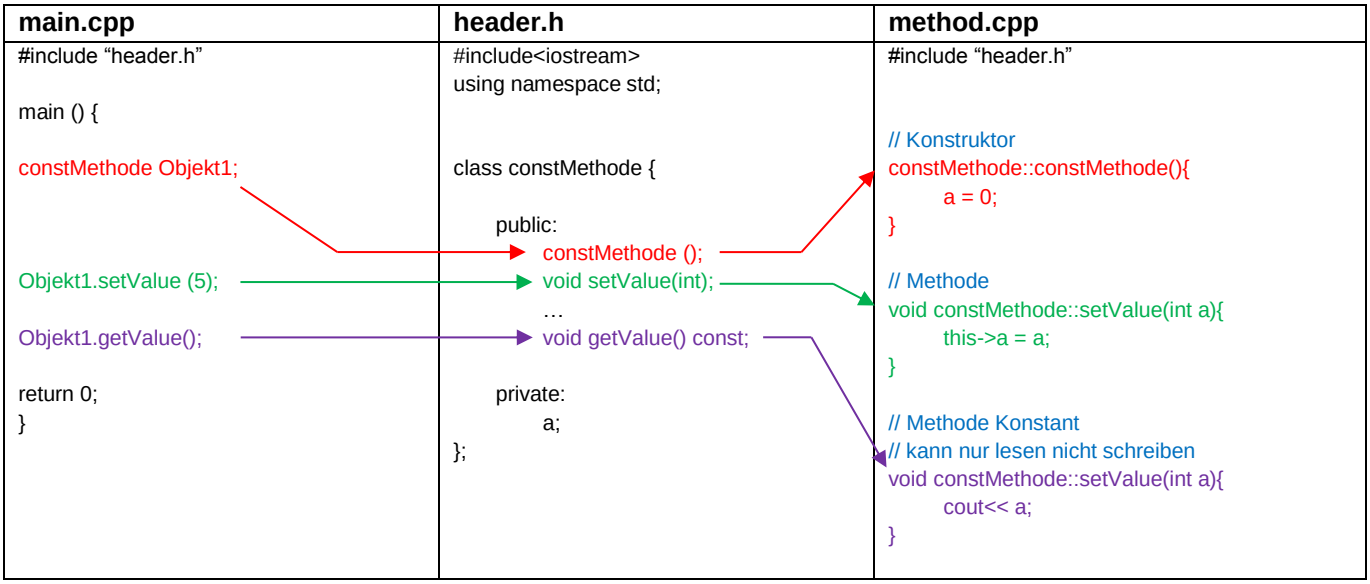
method.cpp

```
#include “header.h”


// Konstruktor
constMethode::constMethode(){
      a = 0;
}

// Methode
void constMethode::setValue(int a){
      this->a = a;
}

// Methode Konstant
// kann nur lesen nicht schreiben
void constMethode::setValue(int a){
      cout<< a;
}
```



15. Klassenvariablen

(*Skript 6.9)*

Es gibt Informationen die in jedem Objekt keinen Sinn machen weil dies in der Klasse nur einmal vorhanden sein muss.

Beispiel:
Klasse Tram, Anzahl Trams in Basel = 365. Diese Information muss nicht in jedem Tram vorhanden sein sondern es reicht wenn sie nur einmal vorhanden ist. Ebenfalls braucht es viele Ressourcen wenn ein neues Tram dazu kommt, weil alle 365 bestehenden Trams diese Info aktualisieren müssten.
Wenn es um eine grosse Anzahl von Objekten geht die aktualisiert werden muss, kann es dazu führen dass das Objekt 1 bereits aktualisiert ist und Objekt 1‘000‘000‘000 noch nicht, heisst es gibt unterschiedliche Werte bei den Objekten während der Laufzeit des Programmes.

Auf Klassenvariablen können alle zugreifen aber nur die Klassen hat diese Variabel. Dieses Attribut teilen sich alle Objekte, Besitzer ist aber die Klasse.

Dies wird durch den Syntax „static“ bei einem „private:“ Attribut in der Klasse erreicht.

Die Klassenvariablen müssen mit einem Konstruktor initialisiert werden. Dies wird jeweils in dem File gemacht wo die entsprechende Klasse auch den Default Konstrukor hat.

Wichtig: Auf das Attribut „static“ als Klassenvariable kann auf zwei Wegen zugegriffen werden!

1. Über das Objekt

Dieser Weg sollte nicht genommen werden denn es ist nicht ersichtlich, dass die Klassenvariabel geschrieben wird. Das Objekt Test2 schreibt auch für das Objekt Test1 die Klassenvariabel.

main.cpp

```
#include “header.h”

main () {

testClass Test1 (1), Test2 (0);
// erzeugen von 2 Objekten

Test2.set(1);
// Objekt wird geöffnet und Methode
// set wird ausgeführt wobei der Wert
// 1 übergeben wird.

return 0;
}
```

header.h

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

class testClass {

puplic:     // Klassenfunktionen & Konstruktoren
…..         // Konstruktor

static void set (int); // Klassenfunktion weil static.
                       // Wird gebraucht um Klassen-
                       // variablen zu initialisieren

puplic:     // Methoden (2te mal optional möglich
            // um zu visualisieren ab hier geht es um
            // die Objekt Methoden
void show (); // Methoden für Objekte
…..

private: //Objekattribute
int a;
…..

private:    //Klassenvariabeln (2te mal optional
            // möglich um zu visualisieren ab hier
            // geht es um die Klassenvariabeln)
static int y;
```

method.cpp

```
#include “header.h”

int testClass::y = 23;
// Initialisierung der Klassenvariabel (ist eine
// Art Konstruktor)

testClass::set (int a){
y = a;
// ich kann hier das private Attribute y den Wert
// aus dem Main zuweisen
}
```

	};	

2. Über die Klasse

Dieser Weg ist der richtige weil ersichtlich ist, dass die private Klassenvariable geschrieben wird.

main.cpp	header.h	method.cpp

#include “header.h”
 	
main () {
 	
testClass Test1 (1), Test2 (0);
// erzeugen von 2 Objekten
 	
testClass::set(9);
// Objekt wird geöffnet und Methode
// set wird ausgeführt wobei der Wert
// 1 an übergeben wird.
 	
return 0;
}
	#include<iostream>
	using namespace std;
	
	class testClass {
	
	puplic: // Klassenfunktionen & Konstruktoren
	….. // Konstruktor
	
	static void set (int); // Klassenfunktion weil static.
	// Wird gebraucht um Klassen-
	// variablen zu initialisieren
	
	puplic: // Methoden (2te mal optional möglich
	// um zu visualisieren ab hier geht es um
	// die Objekt Methoden
	void show (); // Methoden für Objekte
	…..
	
	private: //Objekattribute
	int a;
	…..
	
	private: //Klassenvariabeln (2te mal optional
	// möglich um zu visualisieren ab hier
	// geht es um die Klassenvariabeln)
	static int y;
	};
		#include “header.h”
		
		int testClass::y = 23;
		// Initialisierung der Klassenvariabel (ist eine
		// Art Konstruktor)
		
		testClass::set (int a){
		y = a;
		// ich kann hier das private Attribute y den Wert
		// aus dem Main zuweisen
		}

16. Sortierverfahren Bubble Sort

Bubble Sort ist ein ineffizientes Verfahren da es immer mindesten zwei Schritte braucht um zwei Zahlen auszutauschen.

Der 1.te Durchlauf erfolgt nach folgendem Prinzip:

varArray	[0]	[1]	[2]	[3]	[4]	[5]
Wert	3	9	7	6	8	4

Schritt	Prüfung	Aktion
1	*Prüfen ist varArray[0] = 3 grösser als varArray[1] = 9*	*Keine*

varArray	[0]	[1]	[2]	[3]	[4]	[5]
Wert	3	9	7	6	8	4

Schritt	Prüfung	Aktion
2	*Prüfen ist varArray[1] = 9 grösser als varArray[2] = 7*	*Ja, austauschen der Werte*

varArray	[0]	[1]	[2]	[3]	[4]	[5]
Wert	3	7	9	6	8	4

Schritt	Prüfung	Aktion
3	*Prüfen ist varArray[2] = 9 grösser als varArray[3] = 6*	*Ja, austauschen der Werte*

varArray	[0]	[1]	[2]	[3]	[4]	[5]
Wert	3	7	6	9	8	4

Schritt	Prüfung	Aktion
4	*Prüfen ist varArray[3] = 9 grösser als varArray[4] = 8*	*Ja, austauschen der Werte*

varArray	[0]	[1]	[2]	[3]	[4]	[5]
Wert	3	7	6	8	9	4

Schritt	Prüfung	Aktion
5	*Prüfen ist varArray[4] = 9 grösser als varArray[5] = 4*	*Ja, austauschen der Werte*

varArray	[0]	[1]	[2]	[3]	[4]	[5]
Wert	3	7	6	8	4	9

ENDE 1.ter Durchlauf! Die höchste Zahl ist jetzt an der letzten Stelle!

Der 2.te Durchlauf erfolgt genau nach demselben Prinzip aber diesmal nur bis zur Position [4] weil die höchste Zahl bereits an der letzten Stelle ist.

Der 3.te Durchlauf erfolgt genau nach demselben Prinzip aber diesmal nur bis zur Position [3] weil die zweithöchste Zahl bereits an der letzten Stelle ist.

Der 4.te Durchlauf erfolgt genau nach demselben Prinzip aber diesmal nur bis zur Position [2] weil die zweithöchste Zahl bereits an der letzten Stelle ist.

Der 5.te Durchlauf erfolgt genau nach demselben Prinzip aber diesmal nur bis zur Position [1] weil die zweithöchste Zahl bereits an der letzten Stelle ist.

Fertig!

Erkenntnis:	**Es braucht also $n_{Durchlauf}$ = [n] Durchläufe um auf diese Art zu sortieren!**

17. Sortierverfahren Selection Sort

Effizienteres Verfahren um Daten zu sortieren

Der 1.te Durchlauf erfolgt nach folgendem Prinzip:

varArray	[0]	[1]	[2]	[3]	[4]	[5]
Wert	3	9	7	6	8	4

Schritt	Prüfung	Aktion
1.1	kleinste Zahl von Array[0] bis Array[5] suchen und Nummer vom Arrayfeld speichern	*Nummer 0 vom Arrayfeld speichern*

varArray	[0]	[1]	[2]	[3]	[4]	[5]
Wert	3	9	7	6	8	4

Schritt	Prüfung	Aktion
1.2	Swap Wert von Array[0]auf Array[0]	*Swap*

Der 2.te Durchlauf erfolgt nach folgendem Prinzip:

varArray	[0]	[1]	[2]	[3]	[4]	[5]
Wert	3	9	7	6	8	4

Schritt	Prüfung	Aktion
2.1	kleinste Zahl von Array[1] bis Array[5] suchen und Nummer vom Arrayfeld speichern	*Nummer 5 vom Arrayfeld speichern*

varArray	[0]	[1]	[2]	[3]	[4]	[5]
Wert	3	4	7	6	8	9

Schritt	Prüfung	Aktion
2.2	Swap Wert von Array[5]auf Array[1]	*Swap*

Der3.te Durchlauf erfolgt nach folgendem Prinzip:

varArray	[0]	[1]	[2]	[3]	[4]	[5]
Wert	3	4	7	6	8	9

Schritt	Prüfung	Aktion
3.1	kleinste Zahl von Array[2] bis Array[5] suchen und Nummer vom Arrayfeld speichern	*Nummer 3 vom Arrayfeld speichern*

varArray	[0]	[1]	[2]	[3]	[4]	[5]
Wert	3	4	6	7	8	9

Schritt	Prüfung	Aktion
3.2	Swap Wert von Array[3]auf Array[2]	*Swap*

Der4.te Durchlauf erfolgt nach folgendem Prinzip:

varArray	[0]	[1]	[2]	[3]	[4]	[5]
Wert	3	4	7	7	8	9

Schritt	Prüfung	Aktion
4.1	kleinste Zahl von Array[3] bis Array[5] suchen und Nummer vom Arrayfeld speichern	*Nummer 3 vom Arrayfeld speichern*

varArray	[0]	[1]	[2]	[3]	[4]	[5]
Wert	3	4	6	7	8	9

Schritt	Prüfung	Aktion
4.2	Swap Wert von Array[3]auf Array[3]	*Swap*

und so weiter… bis man bei der letzten zwei Zahlen angekommen ist! Fertig!

18. Sortierverfahren Insertion Sort

(Aufgabe 25.3)?

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

int main(){
  int i(0), j(0), v(0);
  int a[] = {13, 3, 23, 43, 18, 2, 67, 11, 53};

  for (i = 1; i<9; i++){
    v = a[i];
    j = i;
    while ((a[j-1] > v) && (j > 0)){
      a[j] = a[j-1];
      j--;
    }
    a[j] = v;
    cout<<"a["<<j<<"] = "<<a[j]<<endl;
  }

  for (i = 0; i<9; i++){
    cout << "Element " << i << " = " << a[i] << endl;
  }
  return 0;
}
```

19. Vererbung

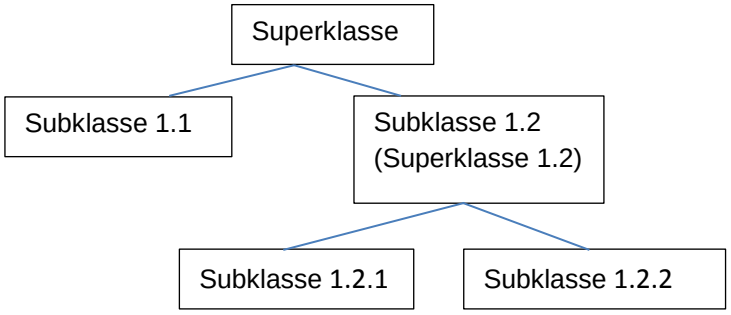
(Script 6.11)

Beispiel:
Es gibt drei verschiedene Vielecke die alle eine Fläche haben und alle vier Kanten aber alle sind unterschiedlich. Zu diesem Zeitpunkt wäre es Sinnvoll eine Klasse zu haben, welche alle generellen Attribute aufweist und dann in einer spezifischer Klasse die spezifischen Attribute enthält.

Generelle Attribute: (Basisklasse)
Kanten(-anzahl), Fläche

Spezifische Attribute (Subklasse)
Anzahl der Kanten, Länge der Kanten, Formel um die Fläche zu berechnen

Wichtig: Basisklassen kennen ihre Subklassen nicht, Subklassen kennen ihre höheren gelegenen Klassen



Enthalten Namen der Klasse, Attribute und Methoden

Deklarieren Vererbung:

Class Superklass 1 { …};		
Class Subklass 1.1 : public Superklass { …};	Class Subklass 1.2 : public Superklass 1 { …};	
	Class Subklass 1.2.1 : public Subklass 1.1 { …};	Class Subklass 1.2.2 : public Subklass 1.1 { …};

Beim Aufrufen im Code der Klasse 1.2 wird:
1. zuerst die Basisklassen 1 aufgerufen (nicht die Subklasse 1.2)
   Grund ist, dass immer zuerst die Grund Attribute erstellt (Konstruktor Basisklasse 1) werden müssen weil die Sub Attribute ja anhand der Grund Attribute erstellt werden.
2. wenn Basisklasse 1 komplett aufgebaut ist wird die Subklasse 1.2 aufgebaut (Konstruktor Subklasse 1.2)

Zugriffsarten:
Zu „private“ und „public“ kommt „protected“
Protected: vollzugriff (lesend und schreiben zugreifen). Subklasse 1.1 & Subklasse 1.2 kann auf Superklasse 1 protected Attribute zugreifen, die Subklasse 1.2.1 & Subklasse 1.2.2 können ebenfalls auf die Superklasse 1 & Subklasse 1.2 zugreifen.

Konstruktoren**:** Jede Klasse, also jede Basisklasse und jede Subklasse muss einen eigenen Konstruktor besitzen. Dem Subklassen-Konstruktor muss mitgeteilt werden welcher Superklassen-Konstruktor beim Erzeugen eines Objektes verwenden muss!

Wichtig!

Konstruktor:
Es wird immer zuerst der Konstruktor der Superklasse aufgerufen und danach erst der Konstruktor der Subklasse.

Dekonstruktor:
Es wird immer zuerst der Dekonstruktor der Subklasse aufgerufen und danach erst der Dekonstruktor der Superklasse.

Beispiel:

main.cpp

```
#include “header.h”

main () {


subclass Objekt1;




return 0;
}
```

header.h

```
#include<iostream>
using namespace std;
//Superklasse
class superClass {
      public:
            superClass();        //Konst. Superkl.
      protected:
            …
      private:
            …
};

//Subklasse
class subClass: public superClass{
      public:
            subClass();          //Konst. Subkl.
      protected:
            …
      private:
            …
```

method.cpp

```
#include “header.h”

superClass:: superClass(){
_a = 0;
}




subclass::subclass(): superClass(){
_b = 0;
}
…
```

Basisinitialisierung: Beim Erstellen des Objektes über die Subklasse werden aus dem Main alle notwendigen Werte mitgegeben. Damit die Werte zur richtigen Variabel in der Super- oder Subklasse gehen müssen diese mittels der Basisinitialisierung beim Konstruktor der Subklasse zugewiesen werden.

Die Basisinitialisierung ist eigentlich eine Regel welche Variabel zu welchem Konstruktor gehört.

Beispiel: Superklasse 2 x double Variable
Supklasse 1 x integer Variable

main.cpp

```
#include “header.h”

main () {

subclass Objekt2(9 ,7, 5.3);

return 0;
}
```

header.h

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

//Superklasse
class superClass {
      public:
            superClass(int a, int b); //Konst. Superkl.
      protected:
            …
      private:
            …
};

//Subklasse
class subClass: public superClass{
      public:
            subClass (double c);      //Konst. Subkl.
      protected:
            …
      private:
            …

};
```

method.cpp

```
#include “header.h”

superClass:: superClass(int a, int b){
_a = a;
_b = b;
}


subclass::subclass(int a, int b, double c):
superClass(a, b){
_c = c;
}
```

UML (Unified Modeling Language) Diagramme:

Über UML ist es möglich den Aufbau graphisch zu definieren und mit speziellen Editoren den Code für die Klassen mit allen Superklassen und Subklassen zu generieren.

UML zeigt folgendes auf:

- Was für Klassen sind enthalten
- Was für einen Standard (Attribute) haben die Klassen
-Welches sind Subklassen, welches sind Superklassen

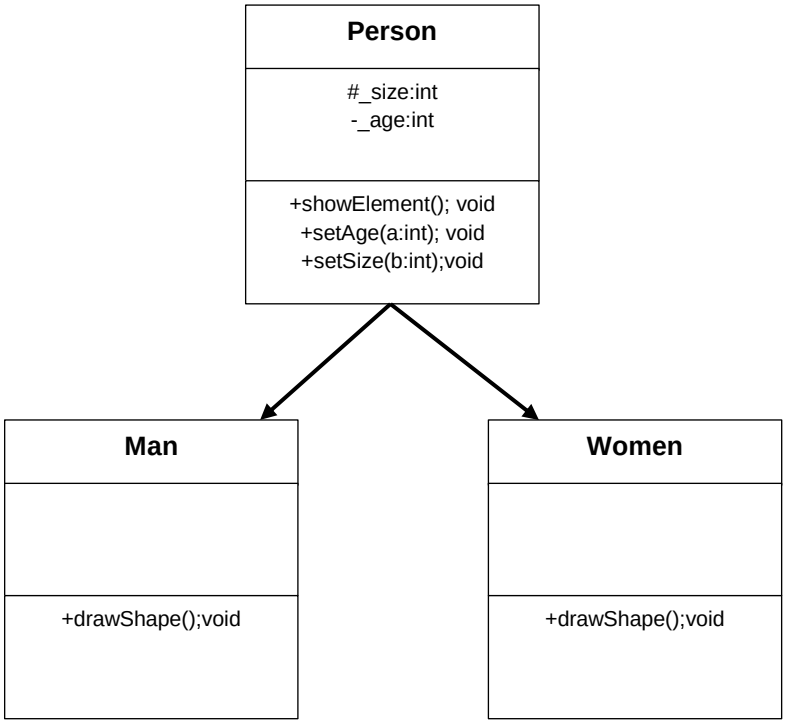
Zeichen & Bedeutung:

+ Level public deklariert (Zugriffmodifizierer)
- Level private deklariert (Zugriffmodifizierer)
Level proteced deklariert (Zugriffmodifizierer)
x Unterstrichen heisst es handelt sich um eine „static“ Variable (siehe 15 Klassenvariablen)

Aufbau:

Name Klasse	TestClass	Syntax
Attribute	+a: int -b: float #c: double	Level & Name : Datentyp
Methoden	+getValue(); int -setValue(a: int); void #copyValue();bool	Level & Name() ; Rückgabe Datentyp oder Level & Name(Name : Übergabe Datentyp) ; Rückgabe Datentyp

Beispiel zu Aufgabe 28.1



Ausgabe in C++:

header.h

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

class Person{
        public:         void showElements();
                        void setAge(int);
                        void setSize(int);
        protected:      int _size;
        private:        int _age;
} ;

class Man:public Person{
        public:         void drawShape();
        };

class Women:public Person{
        public:         void drawShape(); /
};
```

method.cpp

```
#include "header.h"

void Person::showElements(){
}

void Person::setAge(int age){
}

void Person::setSize(int size){
}

void Man::drawShape(){
}

void Women::drawShape(){
}
```