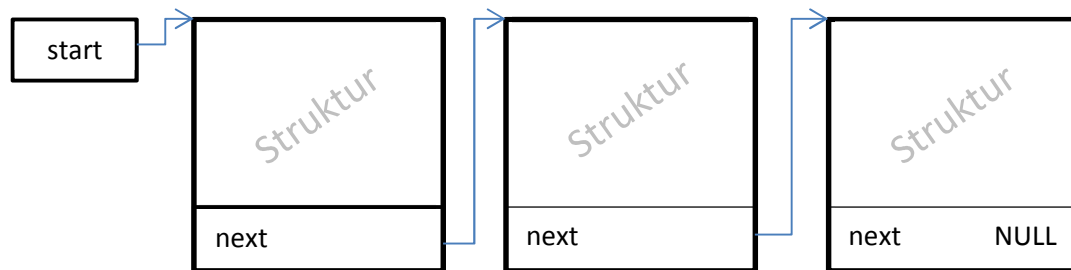
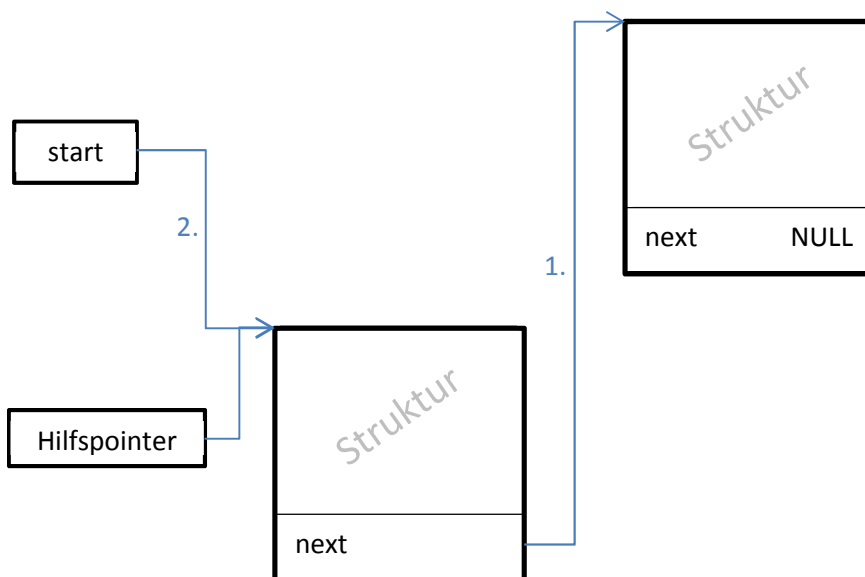


Speicherstrukturen

Einfache verkettete Liste:



Einfügen am Listenanfang:



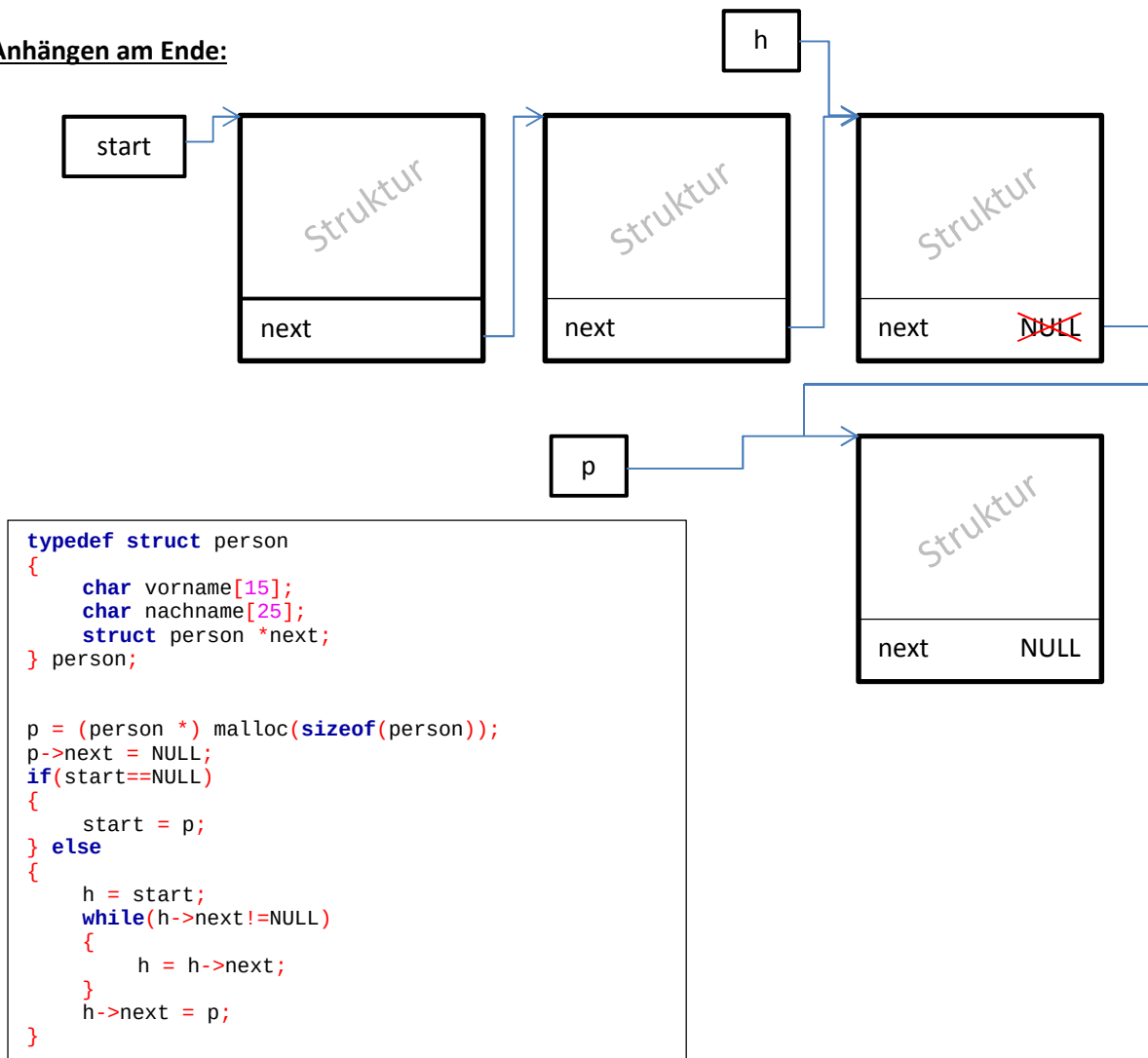
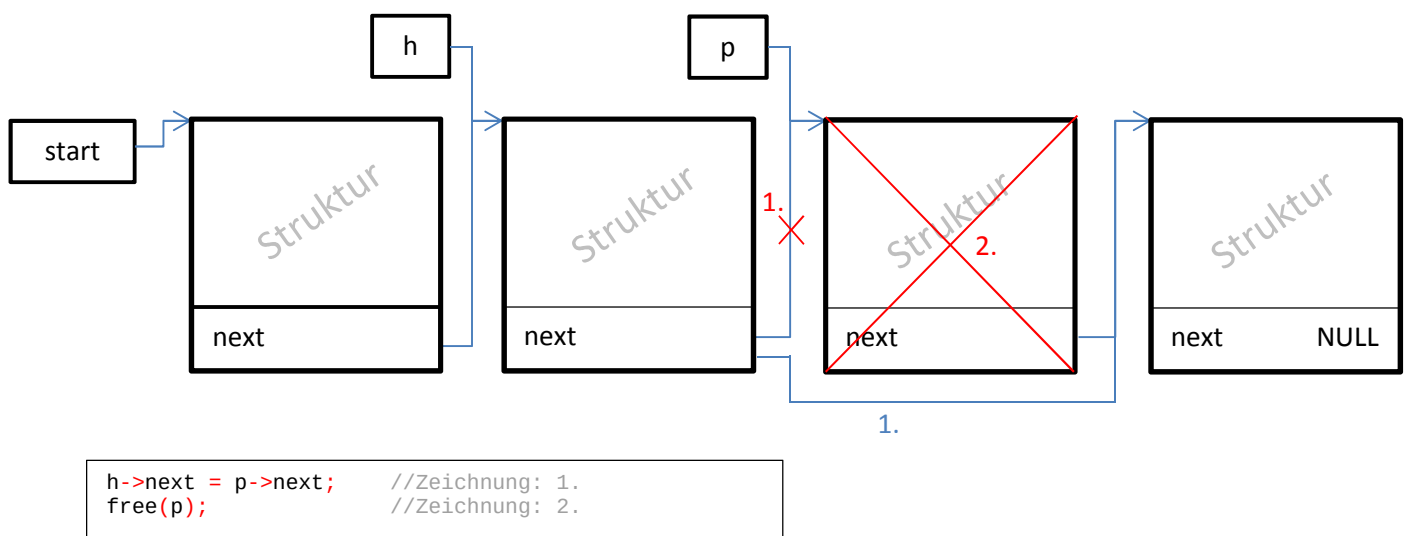
```
typedef struct person
{
    char vorname[15];
    char nachname[25];
    struct person *next;
} person;

int main()
{
    person *start;           //Beginn der Liste
    person *p;               //Hilfspointer
    char auswahl;
    start = NULL;
    do {
        p = (person *) malloc(sizeof(person));
        scanf("%s", (*p).vorname);
        scanf("%s", p->nachname);
        p->next = start;      //Zeichnung: 1.
        start = p;           //Zeichnung: 2.
        auswahl = getche();
    } while(auswahl!='j');
    return 0;
}
```

Aufgabenstellung

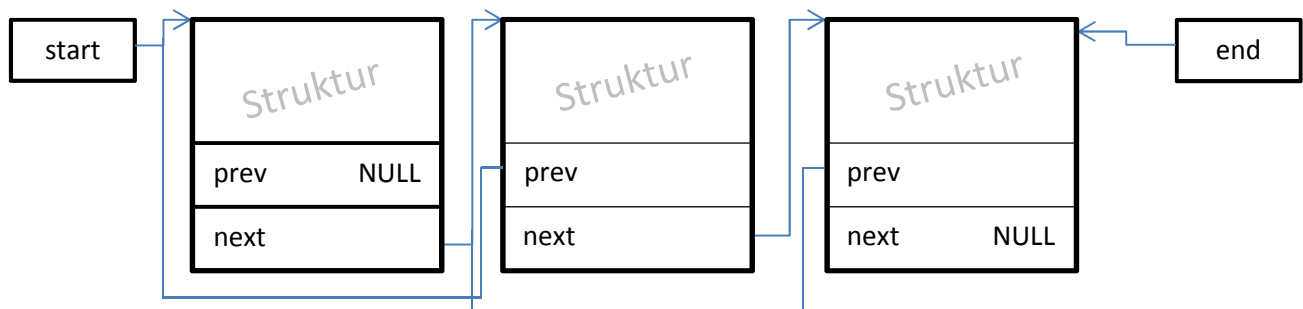
Es soll ein Array vom Typ Test erstellt werden. Dieser Typ enthält Datum, Gegenstand und Punkte. Dem Benutzer wird folgendes Menü angeboten:

1. Test eingeben
2. am Bildschirm ausgeben
3. von Datei einlesen
4. auf Datei speichern
5. Punkteschnitt berechnen

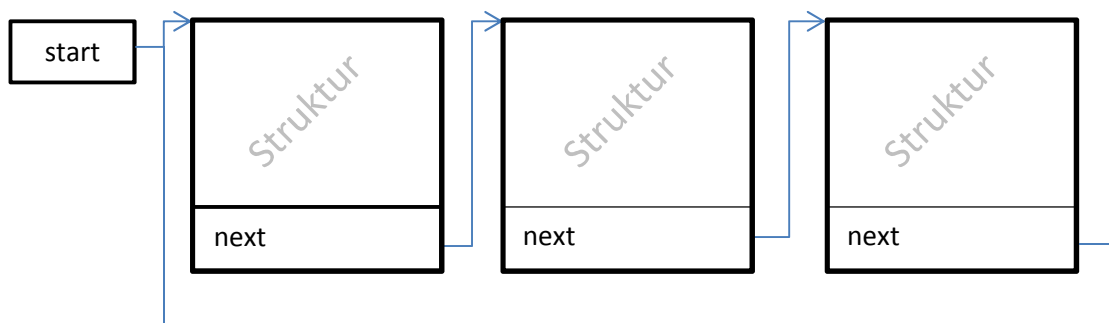
Anhängen am Ende:Löschen eines Elements:

Weitere verkettete Strukturen:

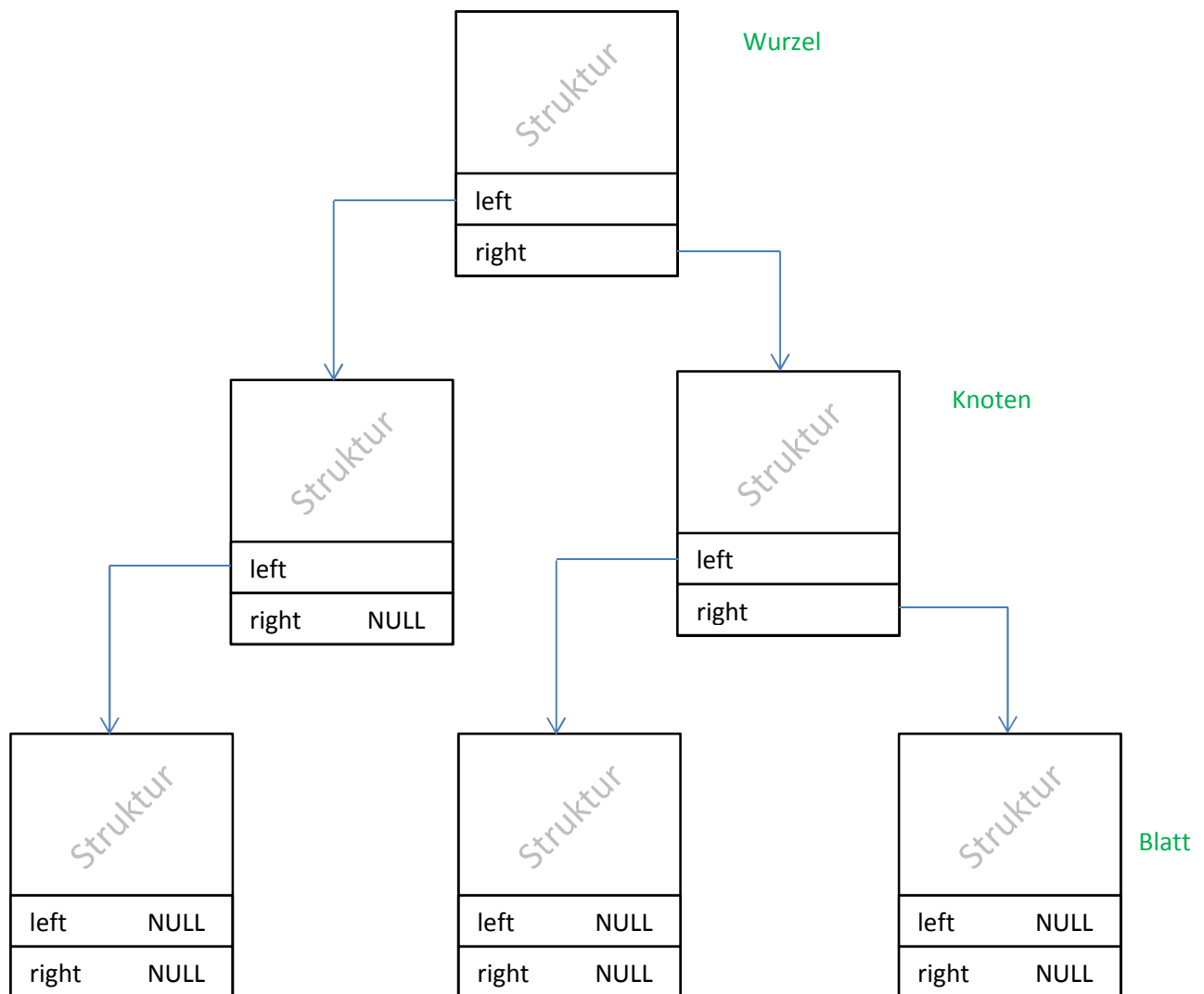
Doppelt verkettete Liste:



einfach verkettete geschlossene Liste:



Binärer Baum:



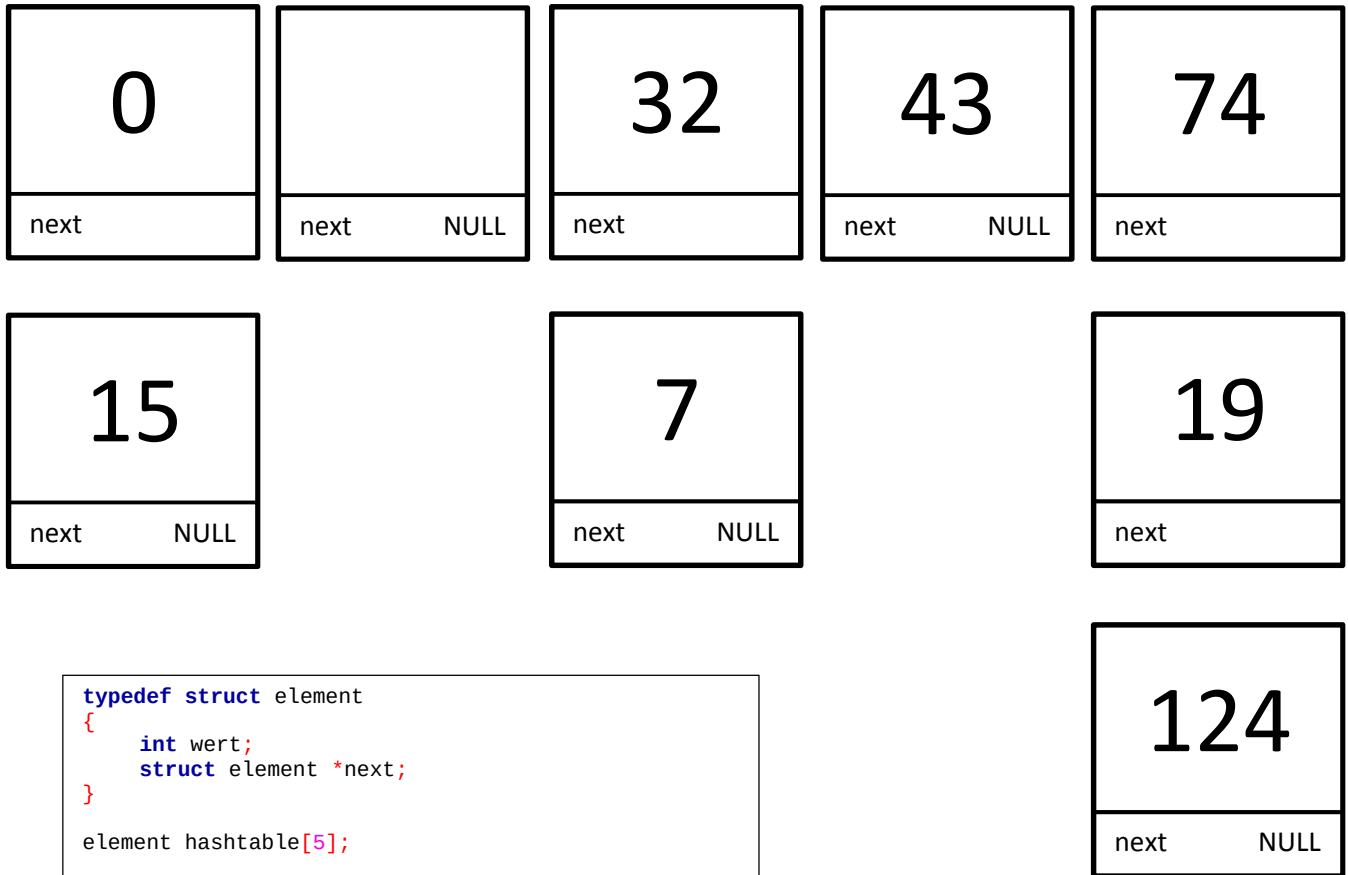
Aufwand für die Suche:

- binärer Baum (ausgeglichener Baum/balanced tree): $\log_2 N$ $N \dots \text{Anzahl}$
- verkettete Liste N

Hashtable:

Eine Hashtable ist ein gewöhnliches Array, aber die Position eines jeden Elements wird aus dem Schlüssel (z.B. Nachname), berechnet. Diese Berechnung wird als Hashfunktion bezeichnet. Bei String kann im einfachsten Fall die Quersumme über alle ASCII-Codes der enthaltenen Buchstaben gebildet werden. Wenn unterschiedliche Einträge den Selben Hashwert erhalten, spricht man von einer Kollision. In diesem Fall kann man an den einzelnen Tabellenplätzen Listen bilden oder man nimmt den nächsten freien Platz in der Tabelle. Eine optimale Hashfunktion sollte möglichst schnell sein, Kollisionen optimal vermeiden und die Schlüssel möglichst gleich verteilt in der Tabelle abbilden.

Beispiel: Hashfunktion $h = \text{wert} \% 5$

0**1****2****3****4**

Präprozessor Direktiven

Jede Quelldatei wird vor der Kompilierung durch den Präprozessor bearbeitet. Jede Präprozessor-Direktive beginnt mit einem Doppelkreuz („number sign“, '#') in einer neuen Zeile. Genügt eine Zeile nicht, wird die Direktive nach einem Backslash am Zeilenende in der nächsten Zeile fortgesetzt.

#define

Mit der define-Direktive wird der Bezeichner durch den darauf folgenden Text ersetzt.

z.B.: `#define TRUE 1` Der Präprozessor ersetzt alle "TRUE " durch "1 "

#include

Die include-Directive kopiert die angegeben Datei in das Programm, wobei die include-Directive durch den Inhalt der Datei ersetzt wird.

z.B.: `#include <Dateiname>` Die Datei wird in den include-Verzeichnissen gesucht.

z.B.: `#include "Dateiname "` Die Datei wird zuerst im aktuellen Verzeichnis gesucht dann in den include-Verzeichnissen, bei Pfadangaben wird nur im entsprechenden Verzeichnis gesucht

#ifdef, #ifndef, #else, #endif

Dient zur Überprüfung, ob ein Symbol definiert ist, in header-Dateien wird dadurch die mehrfach-Einbindung des Dateiinhalts verhindert.

z.B.: `#ifndef STDIO_H`
 `#define STDIO_H`

 `#endif`

Objektorientierte Programmierung (OOP)

bisher: strukturierte Programmierung:

Beschreibt die Programmsteuerung, jedes Programm besteht aus den 3 Grundbausteinen:

1. Anweisungen (Befehle)
2. Verzweigungen (IF und SWITCH)
3. Schleifen (DO, WHILE)

jetzt: Es werden Eigenschaften (Daten) und Methoden (Operationen) zusammengefasst.

Klasse:

Klasse ist der Bauplan für gleichartige Objekte, beschreibt Strukturen und Verhalten. Sie besteht aus:

- Methoden (Operationen, Elementfunktionen)
- Eigenschaften (Daten, Attribute)

Objekt (Instanz):

Wird zur Ausführung (Laufzeit) erstellt und verhält sich entsprechend der Klasse.

Objektorientierte Sprachen müssen 3 Kriterien erfüllen:

- **Datenkapselung** (encapsulation, information hiding):
Auf die Eigenschaften eines Objekts kann von außen nur über Methoden zugegriffen werden.
Der direkte Zugriff ist nicht möglich
- **Vererbung** (inheritance):
Klassen können von anderen Klassen abgeleitet werden und erben damit alle Eigenschaften und Methoden der Oberklasse
- **Polymorphie** (Vielgestaltigkeit):
Objekte verschiedener Klassen reagieren auf den gleichen Methodenaufruf unterschiedlich

Beispiel: Rennwagen

Datei: Rennwagen.h

```
class Rennwagen //Klasse Rennwagen
{
    private: //von außen nicht sichtbar
        int geschwindigkeit; //Eigenschaft
    public: //von außen sichtbar
        Rennwagen() //Konstruktor (gleich Klassenname ohne Typ)
        {
            geschwindigkeit = 0; //Initialisierung
        }
        void beschleunigen(int geschw) //Methode
        {
            geschwindigkeit += geschw;
        }
        void bremsen()
        {
            geschwindigkeit -= 50;
            if(geschwindigkeit < 0)
            {
                geschwindigkeit = 0;
            }
        }
};
```

Datei: Rennwagen.cpp

```
#include <iostream>
#include "Rennwagen.h"

using namespace std;

int main()
{
    Rennwagen Ferrari; //Objekt Ferrari (Instanz)
    Ferrari.beschleunigen(250); //Methode beschleunigen
    Ferrari.bremsen();
    Ferrari.bremsen();
    Ferrari.beschleunigen(50);
    return 0;
}
```

Ein- und Ausgabe in C++:

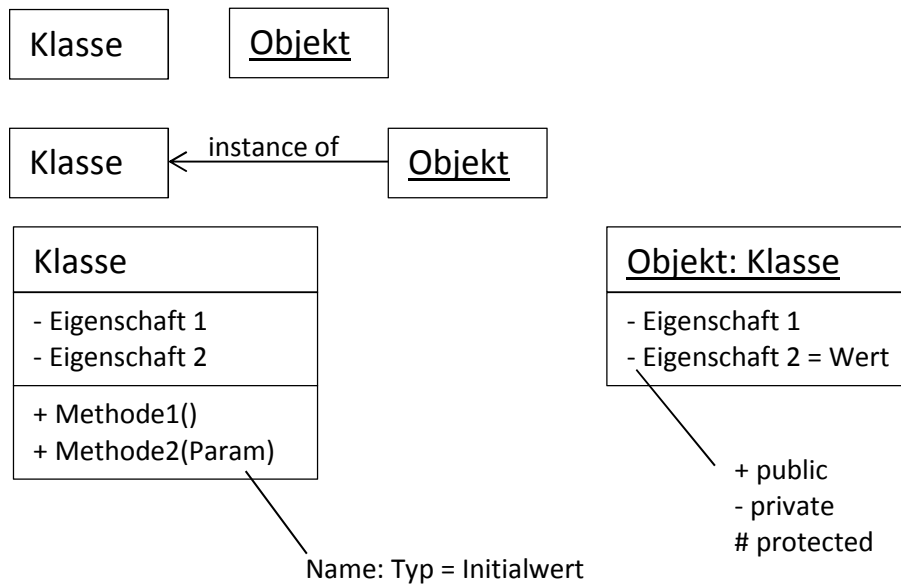
```
cout << "Bitte geben Sie eine Zahl ein: ";
cin >> zahl1;
cout << "Zahl1: " << zahl1 << endl;
```

ohne „using namespace std“:

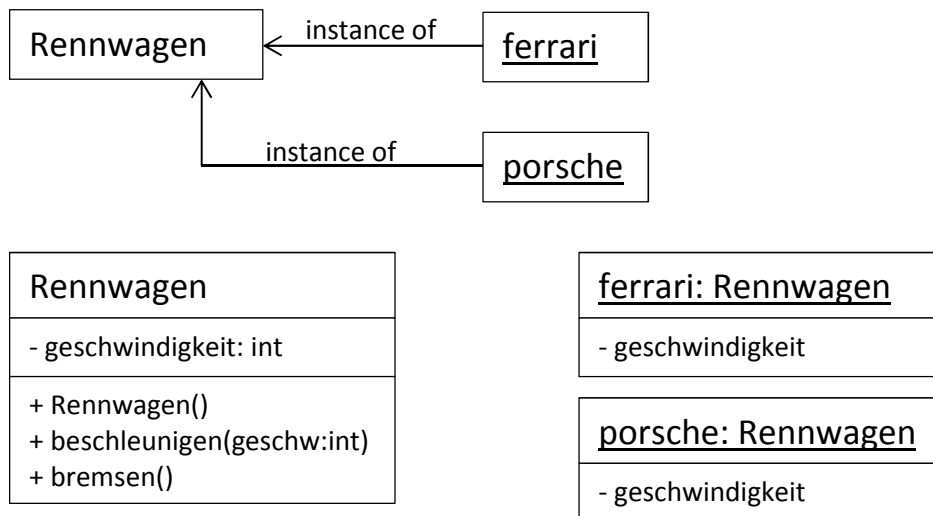
```
std::cout
std::cin
```

UML-Diagramm

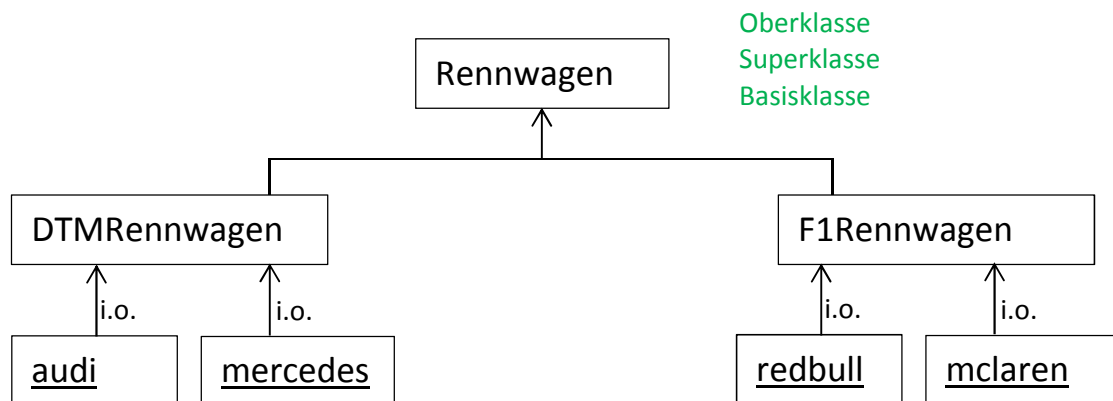
Unified Modeling Language (grafische Modellierungssprache)



Bsp: Rennwagen



Vererbung



Oberklasse
Superklasse
Basisklasse

Beispiel: Rennwagen

Datei: rennwagen.h

```

class Rennwagen
{
    protected:                                //Von außen nicht sichtbar, jedoch für
    int geschwindigkeit;                       //abgeleitete Klassen sichtbar
    public:
        Rennwagen()
        {
            geschwindigkeit = 0;
        }
        void beschleunigen(int geschw)
        {
            geschwindigkeit += geschw;
        }
        void bremsen()
        {
            geschwindigkeit -= 50;
            if(geschwindigkeit < 0)
            {
                geschwindigkeit = 0;
            }
        }
        int getGeschwindigkeit()               //Zugriff auf Eigenschaft
        {
            return geschwindigkeit;
        }
};

class DTMRennwagen : public Rennwagen         //DTMRennwagen erbt von Rennwagen ab
{
    public:
        void bremsen()
        {
            geschwindigkeit -= 40;
            if(geschwindigkeit<0)
            {
                geschwindigkeit = 0;
            }
        }
};

class F1Rennwagen : public Rennwagen          //F1Rennwagen erbt von Rennwagen ab
{
    public:
        void bremsen()
        {
            geschwindigkeit -= 60;
            if(geschwindigkeit<0)
            {
                geschwindigkeit = 0;
            }
        }
};
  
```

Datei: rennwagen.cpp

```

#include <iostream>
#include „rennwagen.h“

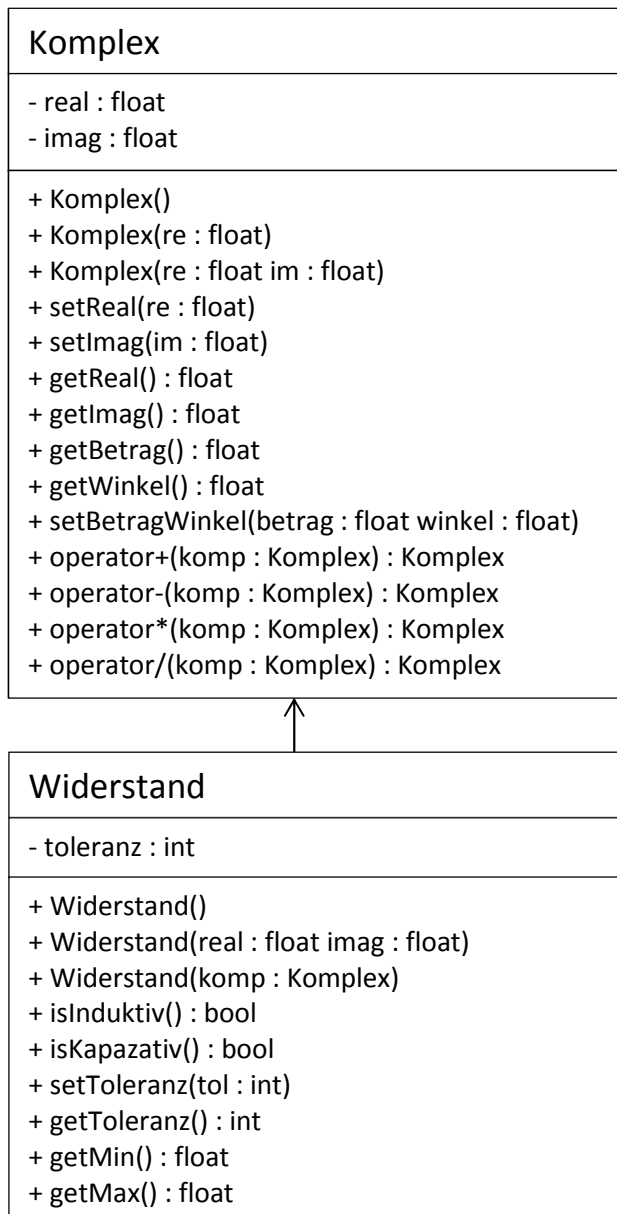
using namespace std;

int main()
{
    DTMRennwagen *audi = new DTMRennwagen;    //dynamische Instanziierung
    DTMRennwagen mercedes;
    F1Rennwagen redbull;
    F1Rennwagen mclaren;

    (*audi).beschleunigen(250);
    cout << "Audi auf " << (*audi).getGeschwindigkeit() << " km/h beschleunigt" << endl;
    audi->bremesen();                          //entspricht (*audi).bremesen();
    redbull.beschleunigen(250);
    redbull.bremesen();
    return 0;
}

```

Bsp.: Komplex



```

class Widerstand : public Komplex
{
    .
    .
    .
    Widerstand() : Komplex()
    {
        toleranz = 0;
    }
    Widerstand(float real, float imag) : Komplex(real, imag)
    {
        toleranz = 0;
    }
    Widerstand(Komplex komp) : Komplex(komp)
    {
        toleranz = 0;
    }
    .
    .
    .
};

```

Überladen von Operatoren

Die Operatoren von C++ haben ein definiertes Verhalten bei bestimmten eingebauten Zyklen. Dieses Verhalten kann auch nicht verändert werden, man kann aber zusätzliches Verhalten für eigene Typen definieren. Wenn man den Operator mit einer Methode überlädt, darf diese Methode nicht statisch sein. Das aufrufende Objekt ist der erste Operand, wird ein 2. Operand benötigt, wird er als Parameter übergeben.

```

class Widerstand : public Komplex
{
    .
    .
    .
    Komplex operator+(Komplex komp)
    {
        Komplex summe;
        summe.setReal(real + komp.getReal());
        summe.setImag(imag + komp.getImag());
        return summe;
    }
    .
    .
    .
};

```

```

Z1 = Z2 + Z3;

Z1 = Z2.add(Z3);

```

Wenn Sie einen Operator mit einer Funktion überladen, die keine Methode der Klasse ist dann werden alle Operanden als Argumente übergeben.

```

std::ostream & operator << (std::ostream & ostr, Komplex komp)
{
    return (ostr << "(" << komp.getReal() << ", " << komp.getImag() << ")");
}

int main()
{
    Komplex Z1(13.5, 17.3);
    cout << "Z1 " << Z1;
    return 0;
}

```

```
std::istream & operator >> (std::istream & is, Komplex & komp)
{
    float eing;
    std::cout << "Realteil: ";
    is >> eing;
    komp.setReal(eing);
    std::cout << "Imaginaerteil: ";
    is >> eing;
    komp.setImag(eing);
    return is;
};

int main()
{
    Komplex Z1;
    cin << Z1;
    return 0;
}
```

Aufgabenstellung

1. Erstellen Sie einen Taschenrechner für komplexe Zahlen (Grundrechnungsarten)
2. Erstellen Sie ein Programm zur Berechnung der Serien & Parallel-Schaltung komplexer Widerstände.
3. Erstellen Sie ein Programm zur Berechnung eines komplexen Spannungsteilers. Neben der Ausgangsspannung U_A soll auch $U_{A \text{ Min}}$ und $U_{A \text{ Max}}$ bestimmt werden ($|Z| \pm 10\%$)

Scope-Operator ::

(scope resolution operator, Bereichsauflösungsoperator, Sichtbarkeitsoperator)

Erlaubt nachträglich auf das Innere einer Klasse zuzugreifen.

```
class Komplex
{
    .
    .
    float getReal();
    .
};

float Komplex::getReal();
{
    return real;
}
```

Referenz

Referenzen müssen bei der Definition initialisiert werden. Man kann sich eine Referenz wie einen konstanten Zeiger vorstellen. Nach der Initialisierung kann die Referenz auf nichts anderes mehr verweisen. Operationen auf der Referenz betreffen die Variable oder das Objekt das referenziert wird.

```
int zahl1 = 20;
int &ref = zahl1;
ref++;           //inkrementiert zahl1
```

Eine Referenz auf eine Klasse kann auch auf ein Objekt einer von dieser Klasse abgeleiteten Klasse zeigen.

```
F1Rennwagen ferrari;
Rennwagen &rennw = ferrari;
```

Referenzen werden häufig als Parameter für Funktionen verwendet. Damit werden Änderungen an einem Parameter ans aufrufende Programm zurückgegeben

```
void aendere(int &zahl1, int &zahl2)
{
    zahl1 = zahl1 * 2;
    zahl2 = zahl1 + 10;
}
```

Referenzen werden oft auch als Returnwerte verwendet.

call by reference

```
void test(int *pointer)           //C
{
    *pointer = 12;
}

void test1(int &pointer)          //C++
{
    pointer = 17;
}

int main()
{
    int zahl = 5;
    test(&zahl);
    test1(zahl);
    return 0;
}
```

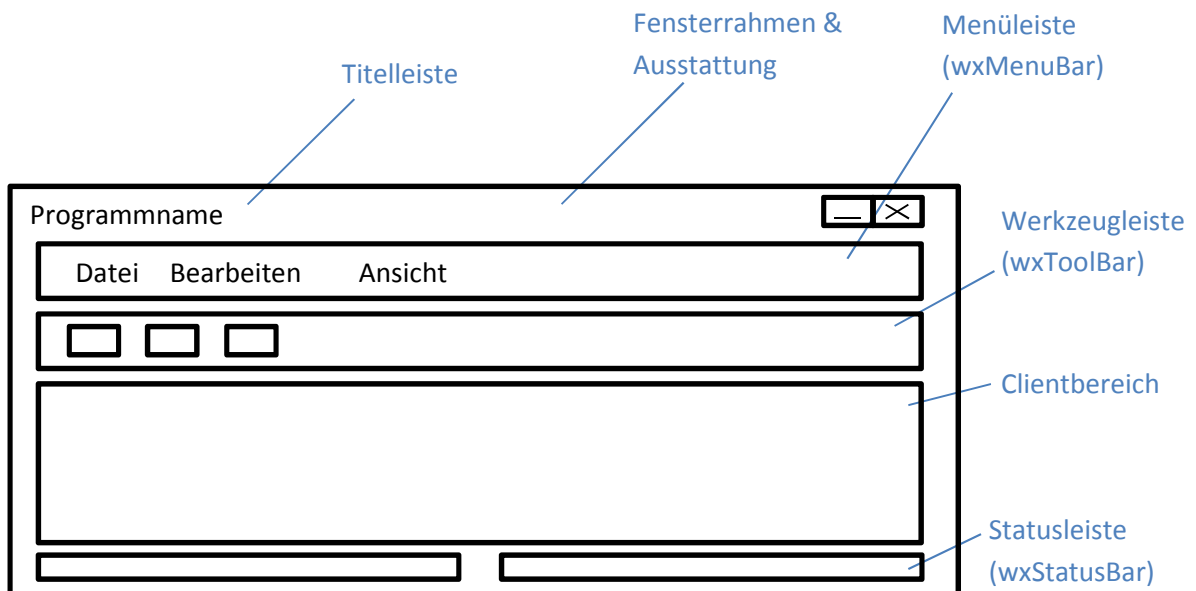
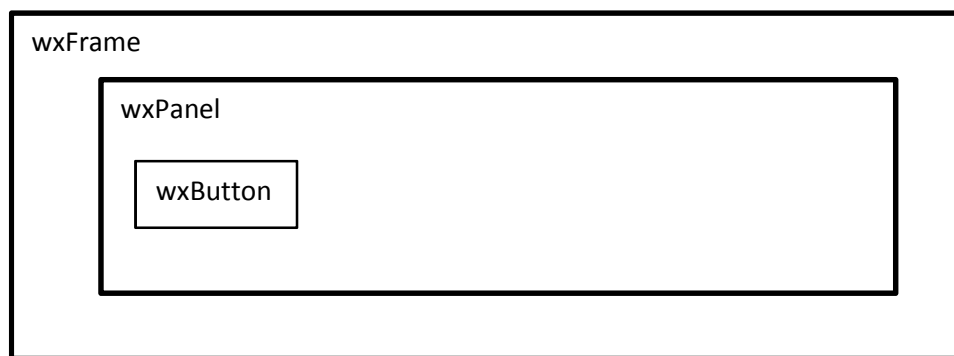
GUI mit wxWidgets

Vorteile:

www.wxwidgets.org

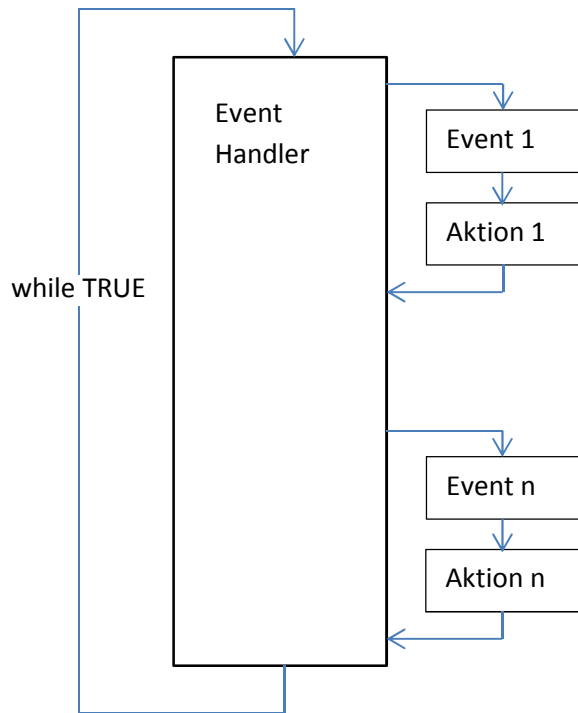
- ziemlich Vollständig
- wird weiterentwickelt
- viele Compiler und Plattformen werden unterstützt
- läuft ohne Änderungen auf verschiedenen Plattformen
- Gute Dokumentation
- für Private und Kommerzielle Anwendung kostenlos
- verwendet Look & Feel des jeweiligen Betriebssystems

Aufbau:



Tool:

RAD-Tool

Rapid Application Development
(wxSmith)**Event Handling**

Benutzeroberflächen werden ereignisgesteuert erstellt (event-driven).

Bei Ereignisgesteuerter Programmierung wird erst eine Aktion gesetzt, wenn eine bestimmte Bedingung erfüllt ist. Sobald der Benutzer ein Steuerelement verwendet, wird ein Event ausgelöst und dadurch kann das Programm entsprechend reagieren. Jener Programmteil, der im Hintergrund die Steuerelemente überwacht, heißt Event-Handler.

```
void wxtestframeFrame::OnButton1Click(wxCommandEvent &event)
{
    TextCtrl1->SetValue(TextCtrl1->GetValue() + "1");
}
```

Konstruktor und Destruktor

Mit einem Konstruktor wird eine Instanz initialisiert. Wenn eine Instanz wieder aus dem Speicher entfernt wird, wird der zugehörige Destruktor aufgerufen. Damit können Abschlussarbeiten für die jeweilige Instanz ausgeführt werden (z.B. Speicher freigeben). Konstruktoren und Destrukturen haben keinen Typ und der Name entspricht dem Klassennamen. Destrukturen haben keine Parameter

```
class Test
{
    private:
        int wert;
    public:
        Test()                //Default Konstruktor
        {
            wert = 0;
        }
        Test(int wrt)         //Konstruktor
        {
            wert = wrt;
        }
        ~Test()              //Destruktor
        {
            ...
        }
};

int main()
{
    Test objekt1;
    Test objekt2(3);
    Test *objekt3;
    objekt3 = new Test(17);
    delete(objekt3);
}
```

C++ Variablen

Variablen können automatisch, statisch oder dynamisch sein.

automatisch:

Variablen werden ohne weitere Angaben in einem Block vereinbart. Für sie wird beim betreten des Blocks Speicherplatz reserviert und beim Verlassen wieder freigegeben. Wird das Unterprogramm rekursiv aufgerufen, wird der Speicherplatz entsprechend oft reserviert.

```
int wert;
Rennwagen rennw;
```

statisch:

Variablen besitzen ihren Speicherplatz vom Start des Hauptprogramms an und behalten ihn bis zum Ende des Programms. Beim Rekursiven Aufruf eines Unterprogrammes wird daher der Speicherplatz nur einmal reserviert.

```
static int wert;
static Rennwagen rennw;
```

dynamisch:

Variablen erhalten ihren Speicherplatz vom Programmierer mit dem Operator „new“ zugewiesen (C: calloc / malloc). Der Speicherplatz muss wieder vom Programmierer freigegeben werden („delete“, C: free)

```
Rennwagen *rennw;  
rennw = new Rennwagen();  
rennw->beschleunigen(100);  
delete(rennw);
```

Polymorphie (Vielgestaltigkeit)

Objekte verschiedener Klassen reagieren auf den gleichen Methodenaufruf unterschiedlich. Erst zu Laufzeit eines Programmes wird die zum jeweiligen Objekt passende Methode ermittelt. (late binding: während der Laufzeit – early binding: beim Übersetzen)

```
class Rennwagen  
{  
    ...  
    virtual void bremsen()    //late binding  
    {  
        geschwindigkeit -= 50;  
    }  
    ...  
}  
  
class F1Rennwagen : public Rennwagen  
{  
    ...  
    virtual void bremsen()  
    {  
        geschwindigkeit -= 60;  
    }  
    ...  
}  
  
class DTMRennwagen : public Rennwagen  
{  
    ...  
    virtual void bremsen()  
    {  
        geschwindigkeit -= 40;  
    }  
    ...  
}  
  
int main()  
{  
    Rennwagen *rennw;  
    ...  
    if(auswahl==1)  
    {  
        rennw = new F1Rennwagen();  
    } else  
    {  
        rennw = new DTMRennwagen();  
    }  
    rennw->bremsen();    //Ohne virtual immer -50 (early binding: Klasse Rennwagen)  
                        //Mit virtual wird zur Laufzeit die Methode der entsprechenden  
                        //Klasse ermittelt (late binding; DTMRennwagen: -40, F1Rennwagen -60)  
    return 0;  
}
```

C++ Strings

```
#include <string>

using namespace std;

int main()
{
    string name;
    name = "Max";
    string nachname("Muster");
    name += " ";           // "Max "
    name = name + nachname; // "Max Muster"
    char x = name [0];     // 'M'
    return 0;
}
```

I/O Streams

Die ANSI-C Bibliotheken...

die jede Art von Ausgabe einheitlich als Datenstrom (Stream) behandeln und die entsprechenden Pufferungen durchführen. Beim Öffnen einer Datei wird ein neuer Stream erzeugt und ein FILE-Pointer angelegt (ein Zeiger auf die Struktur vom Typ File, die Informationen zum Stream enthält). Beim Starten eines Programmes sind bereits standardmäßig 3 Dateien mit folgenden FILE-Pointern geöffnet:

- stdin Tastatur
- stdout } Bildschirm
- stderr }

IO-Streams sind in C++ das bevorzugte Verfahren zur Ein- und Ausgabe:

- cin
- cout
- cerr
- clog

Bsp.:

```
string name;
```

```
name = „Max“;
```

```
cout << name << endl;
```

//Nachschreiben: this-pointer (& anderes?)

Objekte als Wertparameter

1. Wertübergabe als Kopie (call by value)

Nachteil: Es wird ein neues Objekt erstellt, das am Ende wieder zerstört wird

2. Wertübergabe als Zeiger (call by reference)

Vorteil: kein neues Objekt

Nachteil: Original-Objekt wird verwendet (versehentliches Ändern)

3. Wertübergabe als Referenz mit const

Durch das Schlüsselwort const wird ein versehentliches Überschreiben verhindert.

Generische Klassen (Templates)

Eine generische Klasse ist eine Schablone (Template), mit deren Hilfe echte Klassen erzeugt werden können. Die generische Klasse hat formale Parameter die als Platzhalter für die einzusetzenden Daten stehen. Durch einen Generierungsprozess wird aus der Schablone eine echte Klasse erzeugt. Dabei werden die formalen Parameter durch die konkreten, bei der Generierung übergebenen Daten ersetzt.