Skript zur vhb-Vorlesung

Programmierung in C++
Teil 2

Prof. Dr. Herbert Fischer
Technische Hochschule Deggendorf

Inhaltsverzeichnis

Primär-Literatur:

Breymann Ulrich
Derr C++ Programmierer, Hanser, 2. Auflage, 2011
ISBN 3-4464-2691-7

Louis Dirk
C++, Hanser Verlag, 1. Auflage, 2014
ISBN: 3-446-44069-2

Kirch-Prinz Ulla, Kirch Peter
C++ Lernen und professionell anwenden, mitp, 2.Auflage, 2002
ISBN 3—89842-171-6

May Dietrich
Grundkurs Softwareentwicklung mit C++, vieweg, 2.Auflage, 2006
ISBN 3-8348-0125-9

Einsenecker Ulrich
C++: Der Einstieg in die Programmierung, W3L GmbH, 1.Auflage, 2008
ISBN 3-9371-3712-4

Sekundär-Literatur:

André Willms
C++-Programmierung lernen
ISBN: 978-3827326744, Addison-Wesley Verlag, 1. Auflage, 2008

Ute Claussen
Objektorientiertes Programmieren
ISBN: 978-3540579373, Springer-Verlag, 2. Auflage, 2013

Oliver Böhm
C++ echt einfach
ISBN: 978-3772374104, Franzis Verlag, 3. Auflage, 2006

Thomas Strasser
C++ Programmieren mit Stil
ISBN: 978-3898642217, dpunkt.Verlag, 2003

John R. Hubbard
C++ Programmierung
ISBN: 978-3826609107, mitp-Verlag, 2003

Arnold Willemer
Einstieg in C++
ISBN 978-3836213851, Rheinwerk-Verlag, 4.Auflage, 2009

Tools:

Code::Blocks für Windows, Linux, Mac OS (kostenlose Software): **http://codeblocks.org/downloads/26**

Alternativen:
CodeLite für Windows, Linux, Mac OS: **http://downloads.codelite.org**
KDevelop für Windows, Linux: **https://www.kdevelop.org/download**
Dev-C++ für Windows: **http://www.bloodshed.net/dev/**
XCode für Mac OS: **https://itunes.apple.com/de/app/xcode/id497799835**

Empfohlene Webseiten:
http://www.c-plusplus.de/cms
http://community.borland.com/cpp/0,1419,2,00.html
http://www.willemer.de/informatik/cpp/index.htm

1 Dateiverarbeitung & Fehlerbehandlung

Wie wir alle wissen gehen mit der Beendigung eines Programms die im Hauptspeicher gehaltenen Daten des Programms verloren. Was können wir dagegen tun? Die Antwort ist sehr einfach. Um Daten permanent zu speichern, müssen diese in einer Datei abgelegt werden.

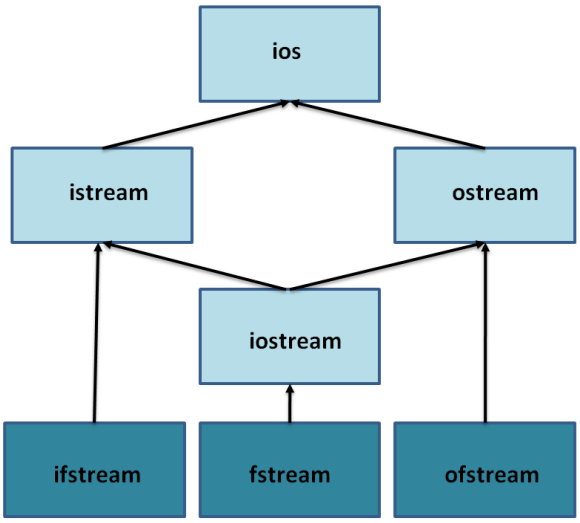
1.1 Dateioperationen

Zeichenfolgen oder einzelne Zeichen können so wie auf dem Bildschirm auch in eine Textdatei geschrieben werden. Es werden jedoch häufig Datensätze in einer Datei gespeichert. Was ist aber ein Datensatz? Ein Datensatz enthält Daten, die logisch zusammengehören, beispielsweise Personendaten oder Kontodaten. Beim Schreiben wird so ein Datensatz in die Datei übertragen, das bedeutet, dass der Datensatz in der Datei aktualisiert oder neu hinzugefügt wird. Beim Lesen dagegen wird dieser Datensatz aus der Datei in eine Datenstruktur des Programms kopiert. Auf diese Weise werden Objekte permanent gespeichert.

Aus der Sicht eines C++ Programms ist eine Datei ein langer „Byte-Vektor“. Die Strukturierung der Datensätze gehört zu den Aufgaben des Programmierers. Dieses bietet den Vorteil maximaler Flexibilität.

Jedes einzelne Zeichen in einer Datei hat eine Byte-Position. Das erste hat die Position 0, das zweite 1 usw. Die aktuelle Position in der Datei ist die Position des Bytes, das als nächstes gelesen oder geschrieben wird. Beim *sequenziellen Zugriff* auf die Datei werden die Daten immer nacheinander gelesen oder geschrieben. Die erste Leseoperation beginnt immer am Anfang der Datei. Wenn eine bestimmte Information aus der Datei geholt werden soll, so muss der Dateiinhalt vom Dateianfang der Reihe nach durchsucht werden. Beim Schreiben in die Datei, kann eine neue oder eine vorhandene überschrieben werden. Es können auch neue Daten am Ende einer vorhandenen Datei angefügt werden.
Im Gegensatz zum sequenziellen Zugriff gibt es auch den *wahlfreien Dateizugriff*. Dieser ermöglicht die aktuelle Dateiposition beliebig zu setzen.

Für die Dateiverarbeitung stellt C++ verschiedene Standardklassen zur Verfügung. Die komfortable Handhabung von Dateien ermöglichen die sogenannten *File-Stream-Klassen*. Diese kümmern sich vor allem um die Verwaltung von Dateipuffern und um systemspezifische Details.



Die obige Klassenhierarchie zeigt, dass die File-Stream-Klassen die bereits uns bekannten Stream-Klassen als Basisklassen besitzen:

- die Klasse ifstream ist von der Klasse istream abgeleitet und ermöglicht das Lesen von Dateien
- die Klasse ofstream ist von der Klasse ostream abgeleitet und ermöglicht das Schreiben in Dateien
- die Klasse fstream ist von der Klasse iostream abgeleitet und ermöglicht sowohl das Lesen als auch Schreiben von Dateien.

1.1.1 Datei öffnen, schreiben und schließen

Zum Öffnen einer Datei benötigen wird ein Objekt der Klasse fstream. Danach wird durch Aufruf der Funktion open() die Datei geöffnet bzw. neu angelegt. Die Datei wird durch den Aufruf der Funktion close() wieder geschlossen.

Um eine Datei bearbeiten zu können, muss diese geöffnet werden. Dabei wird:
- der Dateiname angegeben, die einen Pfad enthalten kann und
- ein sogenannter Eröffnungsmodus

festgelegt.

Wenn es keine Pfadangabe gibt, dann muss sich die Datei im aktuellen Verzeichnis befinden. Der Eröffnungsmodus bestimmt insbesondere, ob die Datei zum Lesen oder zum Schreiben geöffnet wird, oder beides.

```
#include <fstream>
#include <cstdlib>

using namespace std;

int main()
{
   fstream f;
   f.open("test.txt", ios::out); // Datei zum Schreiben öffnen
   f << "Dieser Text wird in die Datei geschrieben" << endl; // Zeichenkette in Datei schreiben
   f.close(); // Datei schließen

   system("pause");
   return 0;
}
```

Eröffnungsmodi:

Konstante	Bedeutung
ios::in	Eine bereits existierende Datei zum Lesen öffnen
ios::out	Zum Schreiben
ios::trunc	Datei wird beim Öffnen geleert
ios::app	Geschriebene Daten ans Ende anhängen. Vor jeder Schreiboperation wird auf das Dateiende positioniert.
ios::ate	Positionszeiger ans Ende setzen
ios::binary	Schreib-/Leseoperationen im Binärmodus durchführen

Verschiedene Eröffnungsmodi können miteinander mit dem Bitoperator | kombiniert werden.

Beachte: der Konstruktor und die Methode open() der File-Stream-Klassen ifstream und ofstream benutzen folgende Default-Werte:

ifstream	ios::in
ofstream	ios::out | ios::trunc

1.1.2 Datei lesen

Die Stream-Objekte für Dateien lassen sich mit cin und cout verwalten.
Um Dateien auszulesen wird der Eingabeoperator >> verwendet, der mit fstream-Objekten in der gleichen Weise wie mit cin arbeitet. Die Datei wird so gelesen, als würden Sie den Inhalt auf der Tastatur eintippen. Um aus den Dateien Leerzeichen auslesen zu können, müssen Sie Elementfunktion getline() verwenden. Als erster Parameter wird ein Zeiger auf char übergeben, als zweiter die maximale Anzahl von Zeichen, die in den Puffer passt.

Beispiel: Auslesen der Datei test.txt

```
#include <fstream>
#include <iostream>
#include <cstdlib>

using namespace std;

int main()
{
   fstream f;
   char cstring[256];
   f.open("test.txt", ios::in);                  // Datei öffnen
   while(!f.eof()) {                             // Solange nicht das Ende (End-of-file) der Datei erreicht wurde
         f.getline(cstring, sizeof(cstring));    // Einzelne Zeile Auslesen und in cstring abspeichern
         cout << cstring << endl;
   }

   f.close();

   system("pause");
   return 0;
}
```

Es gibt aber auch noch eine globale Funktion namens getline(), die nicht fstream, sondern ein Objekt vom Typ ifstream als ersten Parameter hat. Diese Funktion arbeitet auch mit der Standardklasse string und akzeptiert auch Objekte vom Typ string als zweiten Parameter. Beispiel von oben würde so aussehen:

```
#include <fstream>
#include <iostream>
#include <cstdlib>
#include <string>

using namespace std;

int main()
{
   ifstream f;
   string s;
   f.open("test.txt", ios::in);          // Öffne Datei
   while(!f.eof()) {                     // Solange noch Daten vorliegen
         getline(f,s);                   // Lese eine Zeile
         cout << s << endl;              // Zeige sie auf dem Bildschirm
   }
   f.close();                            // Datei schließen

   system("pause");
   return 0;
}
```

1.1.3 Benutzerdefinierte Datei

Das nächste Beispiel soll zeigen, wie man benutzerdefinierte Dateien mit einem strukturierten Aufbau erstellt und die gespeicherten Werte ausliest.

Beispiel:

```
#include <iostream>
#include <cstdlib>
#include <fstream>

using namespace std;

int main()
{
   ofstream einlesenDatei("mitarbeiter.txt");
   ifstream auslesenDatei("mitarbeiter.txt");

   cout << "Mitarbeiter Nr, Name, Einkommen" << endl;

   int nr;
   string name;
   double einkommen;
   bool weiter;

   do
   {
      cin >> nr >> name >> einkommen;       //Nr, Name, Einkommen eingeben. Durch Leerzeichen getrennt

      einlesenDatei << nr << ' ' << name << ' ' << einkommen << endl;  // Benutzerdefinierte Ausgabe mit den Werten
                                                                       // in Datei mitarbeiter.txt speichern
      cout << "Weiter? [1] JA [0] NEIN: ";
      cin >> weiter;
   }while(weiter == 1);                                                // Solange weiter gleich 1 ist, gib Daten ein

   einlesenDatei.close();                                              // Datei schließen

   cout << endl << "--- AUSLESEN ---" << endl;

   while(auslesenDatei >> nr >> name >> einkommen)          // Datei mitarbeiter.txt auslesen
   {                                                        // Jeder Spaltenwert einer Zeile in die Variable speichern
      cout << nr << ", " << name << ", " << einkommen << endl;          // Ausgabe der einzelnen Zeilen mit den Werten
   }

   system("pause");
   return 0;
}
```

Erläuterung:
Wir haben einen *ofstream* zum Erstellen und Speichern der eingegebenen Daten und einen *ifstream* zum Auslesen der Daten. Der Benutzer wird aufgefordert eine Nummer, einen Namen und ein Einkommen einzugeben. Dies wird solange durchgeführt, bis der Benutzer bei der Abfrage „Weiter?“ den Wert 0 eingibt. Da bei *cin* drei Variablen stehen, kann man diese direkt durch ein Leerzeichen getrennt auf einmal eingegeben. Beispiel: 1 Max 1999.90
Jede eingelesene Zeile wird dann in das ofstream-Objekt geschoben und die Zeilen werden in der Datei mitarbeiter.txt abgespeichert. Beim Verlassen der Schleife wird die Datei geschlossen.
Bei der Ausgabe wird ebenfalls eine Schleife erstellt. Dort wird gleich das ifstream-Objekt verwendet. Da wir wissen, dass unsere Datei auf drei Spalten aufgebaut ist, können wir dem ifstream-Objekt sagen, dass er die ausgelesenen Werte in die Variable *nr*, *name* und *einkommen* hinterlegen soll. Die Ausgabe der Werte erfolgt innerhalb der Schleife. Dies erfolgt solange, bis wir das Dateiende erreicht haben. Hierfür wird kein *close()* benötigt, da das ifstream-Objekt weiß, dass es das Ende erreicht hat und deswegen die Datei selbstständig schließt.

1.2 Fehlerbehandlung

1.2.1 Try, Catch, Throw

C++ bietet einen Mechanismus, der es ermöglicht, einen Block von Anweisungen gegen Abstürze zu sichern. Alle darin auftretenden Ausnahmefehler werden einem Behandlungsblock zugeleitet. Der gesicherte Anweisungsblock wird durch das Schlüsselwort *try* (engl. versuchen) eingeleitet. Der Fehlerbehandlungsblock beginnt mit *catch* (engl. fangen). Im Anweisungsblock wird also versucht, das Programm fehlerfrei abzuarbeiten. Der Fehlerbehandlungsblock fängt die Abstürze ab wie ein Netz.

Beispiel: Try, Catch

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main()
{
   Try                                  // Anweisung, wo ein Fehler auftreten könnte
   {
      int divisor = 0;
      int dividend = 1;
      int quotient = dividend/divisor; // Fehler Tritt auf, Programm würde abstürzen!
   }
   catch (...)                          // Allgemein Fehler abfangen und Fehlermeldung werfen
   {
      cerr <<  "Problem erkannt..." << endl; // Ausgabe des Fehlers mittels cerr
   }

   system("pause");
   return 0;
}
```

Erläuterung:
Die Division durch 0 im Beispiel würde normalerweise zu einem Programmabsturz führen. Man nennt einen solchen Fehler eine *Exception* oder auf Deutsch *Ausnahme*. Tritt eine solche Ausnahme in einem try-Block auf, dann wird die Verarbeitung in dem behandelnden catch-Block fortgesetzt.

Natürlich kann das Problem verhindert werden, indem vor jeder Division der Divisor geprüft wird. Aber mit Hilfe der try-Anweisung wird die Fehlerbehandlung bewusst aus dem normalen Programmablauf herausgenommen. Diese Trennung von Code und Fehlerbehandlung führt zu Programmen, die leichter lesbar und damit auch leichter zu warten sind. Ein weiterer Vorteil liegt darin, dass nicht jeder Teilschritt einzeln auf jede erdenkliche Fehlersituation geprüft werden muss (siehe Beispiele im Abschnitt 1.2.2).

In der Klammer der catch-Anweisung wird der Typ der Exception benannt, die von diesem Block abgefangen werden soll. Es ist möglich, mehrere catch-Blöcke für unterschiedliche Typen hintereinander zu setzen. Die drei Punkte hinter dem catch-Befehl sind ein Platzhalter für alle Typen und bezeichnen den generellen oder allgemeinen Ausnahmebehandlungsfall, so dass hier alle noch nicht behandelten Ausnahmen gefangen werden sollen. Dieser allgemeine catch muss immer als letzter Fehlerbehandlungsblock stehen.

Nicht alle Fehler führen immer gleich zu einer Ausnahmesituation. Aber nur Ausnahmen können durch catch abgefangen werden. Darum ist es besonders interessant, eigene Ausnahmesituationen definieren zu können. Der Befehl *throw* erzeugt eine solche Ausnahme. Er bricht die Verarbeitung sofort ab und springt direkt in die passende Ausnahmebehandlung.

Beispiel: Eigene Ausnahme mit Throw generieren

```
#include <cstdlib>
#include <iostream>

using namespace std;

void TuWas(int Problem)
{
   if(Problem>0)
   {
      throw 0;                  // Fehler mit “Fehlercode” 0 an catch-Block übergeben
   }
}

int main()
{
   try
   {
      TuWas(1);
   }
   catch (int a)
   {
      cerr <<  "Ausnahme:" << a << endl; // Ausgabe: Ausnahme: 0
   }

   system("pause");
   return 0;
}
```

Wenn der catch-Block *int* als Parameter hat, bearbeitet er nur Ausnahmen, die durch einen throw-Befehl mit einer Zahl als Argument ausgelöst wurden. Um andere Parameter zu bearbeiten, wird einfach ein weiterer catch-Block mit einem anderen Parametertyp angehängt. Um alle übrigen Ausnahmen zu behandeln, kann der allgemeine catch-Befehl mit den drei Punkten als Parameter ganz zum Schluss auch noch angehängt werden. Die passenden catch-Blöcke werden nach ihren Parametern ausgewählt. Allerdings hat ein catch immer nur einen Parameter. Eine automatische Typanpassung wie bei Funktionen wird hier nicht durchgeführt. Wenn Sie also beispielsweise throw mit dem Argument 1.2 verwenden, muss der passende catch als Parameter double und nicht float haben, weil Fließkommakonstanten vom Compiler standardmäßig als double behandelt werden. Das folgende Beispiel zeigt mehrere catch-Blöcke für verschiedene Typen.

Beispiel: Mehrere Catch-Blöcke

```
#include <iostream>
#include <cstdlib>

using namespace std;

// Tuwas wird unterschiedliche Typen werfen, je nach dem
// Wert des Parameters Problem.
void Tuwas(int Problem)
{
   switch (Problem)
   {
   case 0: throw 5; break;                    // wirft int
   case 1: throw (string)"test.dat"; break;   // wirft string
   case 2: throw 2.1; break;                  // wirft double
   case 3: throw 'c'; break;                  // wirft char
   }
}

// Testprogramm
int main()
{
   // Problem-Nummer eingeben
   int Auswahl;
   cout << "Zahl zwischen 0 und 3 eingeben:" << endl;
   cin >> Auswahl;
   // Der try-Block fängt die Exception in Tuwas ab
   try
   {
      Tuwas(Auswahl);
   }
   catch(int i) // Behandler für int
   {
      cerr << "Integer " << i << endl;
   }
   catch(string s) // Behandler für string
   {
      cerr << "Zeichenkette " << s << endl;
   }
   catch(double f) // Behandler für double
   {
      cerr << "Fließkomma " << f << endl;
   }
   catch(...) // fängt alle anderen Exception-Typen
   {
      cerr << "Allgemeinfall" << endl;
   }

   system("pause");
   return 0;
}
```

Wenn Sie das Programm starten, können Sie durch die Zahlen 0 bis 3 auswählen, welche Ausnahme ausgelöst wird. Bei 1 wird eine Zeichenkette geworfen, die dann im catch-Block als Parameter s auch weiterverarbeitet werden kann. Hierbei muss die Zeichenkette gecastet werden. Bei 3 wird ein char geworfen, für den kein catch-Block vorgesehen ist. Also wird diese Ausnahme vom allgemeinen catch-Block gefangen.

1.2.2 Fehlerbehandlung bei Dateizugriff

Bei der Bearbeitung einer Datei können diverse Probleme entstehen, auf die das Programm vorbereitet sein sollte. Die Datei auf die Sie zugreifen wollen existiert nicht oder beim Schreiben fehlt das Schreibrecht oder die Festplatte ist bereits voll. Aus diesem Grund ist die Klasse *ifstream* auf solche Ausnahmebehandlungen vorbereitet.

Sie können das Stream-Objekt jederzeit mit der Elementfunktion good() abfragen, ob dieser bei der letzten Aktion Fehler festgestellt hat. Die Elementfunktion hat keine Übergabeparameter. Sie liefert jedoch einen booleschen Wert zurück.

Beispiel:

```
#include <cstdlib>
#include <fstream>
#include <iostream>

using namespace std;

int main()
{
   ifstream aus_dat;
   ofstream kop_dat;
   aus_dat.open("quelle.txt", ios::in);
   kop_dat.open("sicherheitskopie.txt");

   if (aus_dat.good()){
      if (kop_dat.good()) {
      char ch;
      while(aus_dat.get(ch)){                                    // Zeichen für Zeichen Datei auslesen
         kop_dat.put(ch);                                        // Zeicheweise in die neue Datei kopieren
      }
      } else {
         cerr << "quelle.txt kann nicht geöffnet werden!" << endl;          // Datei existiert nicht
      }
   } else {
      cerr << "sicherheitskopie.txt kann nicht geöffnet werden!" << endl; // Datei ist schreibgeschützt
   }

   aus_dat.close();
   kop_dat.close();

   system("pause");
   return 0;
}
```

Diese Art von Fehlerbehandlung ist recht simpel und elegant, da wir hier direkt auf das Stream-Objekt zugreifen können und dessen Zustand abfragen. Bei einer größeren Anzahl von Stream-Objekten kann es jedoch schnell unübersichtlich werden, da es zu einer Überfrachtung von Fehlerbehandlungen kommt.

Eine einfachere Methode dieses Problem zu umgehen wäre das Erstellen eines einzigen Try-Catch Blocks und das Ausgeben der Fehlermeldung mit dem Dateinamen durch eine Throw-Anweisung, wobei der Dateiname als Parameter übergeben wird.

Beispiel:

```
#include <cstdlib>
#include <fstream>
#include <iostream>

using namespace std;

int main()
{
   ifstream aus_dat;
   ofstream kop_dat;

   try
   {
         aus_dat.open("quelle.txt",ios::in);
         kop_dat.open("sicherheitskopie.txt");
         if(!aus_dat.good())
            throw (string)"quelle.txt";                    // Wenn Datei nicht existiert
         else if(!kop_dat.good())
            throw (string)"sicherheitskopie.txt";          // Wenn Datei schreibgeschützt ist

         char ch;
         while(aus_dat.get(ch)){                           // Zeichen für Zeichen Datei auslesen
            kop_dat.put(ch);                               // Zeicheweise kopieren
         }
   }
   catch(string dateiname)
   {
      cerr << "Fehler: " << dateiname << " kann nicht geöffnet werden." << endl; // cerr wird wie cout verwendet
   }                                                                             // Ausgabe als Fehler

   aus_dat.close();
   kop_dat.close();

   system("pause");
   return 0;
}
```

2 Referenzen und Zeiger

Referenzen und Zeiger sind für Einsteiger in die C++ Programmierung nicht einfach zu verstehen und bilden einen der am schwierigsten zu erlernenden Bestandteile dieser Programmiersprache.

2.1 Definition von Zeiger

Ein Zeiger (englisch: pointer) ist eine Variable, die die Adresse einer anderen Variable enthält. Da der Zeiger keine direkte Verbindung zu den eigentlichen Daten herstellt, spricht man auch von Indirektion oder Referenzierung.

Syntax: Typ ***Name**;

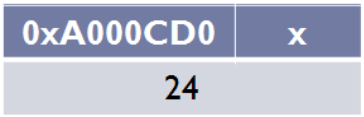
Beispiel: int *ptr; oder int* ptr;

Sie können Zeiger für alle beliebigen integralen Datentypen (int, char, long, short und so weiter) und auch für Objekte (Felder, Strukturen, Klassen oder Instanzen) deklarieren.

Um Zeiger bildlich darzustellen schauen wir uns zuerst an, woraus eine Variable besteht. Eine Variable ist durch vier Angaben eindeutig definiert:

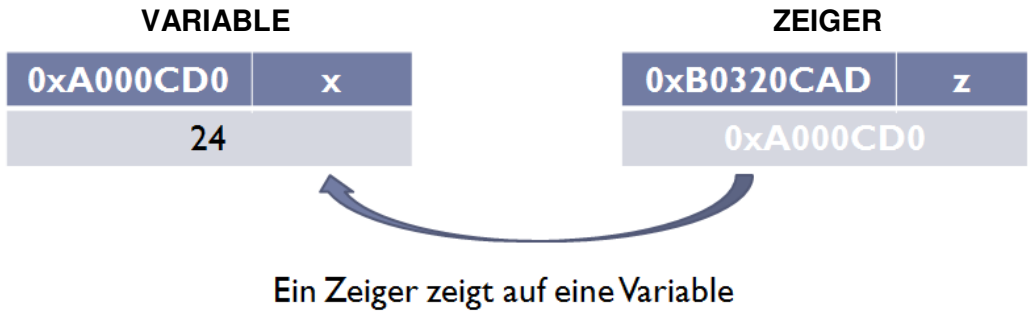
- Position (Adresse)
- Größe (benötigter Speicherplatz)
- Name
- Inhalt

Beispiel einer typischen Variablen.



In unserem Beispiel heißt die Variable x, beinhaltet den Wert 24 und liegt an der Speicheradresse 0xA000CD0. Die Größe der Variable wird durch den Datentyp festgelegt.

Wie bereits erwähnt wurde, ist ein Zeiger eine Variable, die die Adresse einer anderen Variablen enthält. Sehen wir uns ein Beispiel dafür.



Erläuterung: Dereferenzierungsoperator *****

Beispiel:

```
int x = 24, *z;
z = &x;
```

In diesem Beispiel ist z als Zeigervariable deklariert und zeigt auf die x Variable. Die Adresse von x wird der Variable z zugewiesen. D.h. z enthält die Adresse der Variable x.

Beachte:

- Zeiger speichern **Adressen**, wohingegen Variablen **Werte** speichern.
- Nicht initialisierte Zeiger enthalten, wie alle anderen nicht initialisierten Variablen, Zufallswerte. Das kann ein Programm zum Absturz bringen.
- Variable und Zeiger, die die Adresse der Variable enthält, müssen dieselben Datentypen haben.

2.2 Dereferenzierung von Zeigern

Einen Zeiger zu dereferenzieren bedeutet, den Inhalt der Speicherposition zurückzuliefern, auf die der Zeiger zeigt.

Beispiel: Dereferenzierung eines Zeigers

```
int x = 20;
int *ptrx = &x;      // dem Zeiger wird die Adresse von x zugewiesen
int z = *ptrx;       // z wird der Wert der Variablen auf die ptrx zeigt zugewiesen (also der Wert von x)
```

Erläuterung: die erste Zeile dieses Beispiels deklariert eine Integervariable x und weist ihr den Wert 20 zu. Die nächste Zeile deklariert einen Zeiger auf einen Integerwert und weist dem Zeiger die Adresse der Variablen x zu. Dazu wird der *Referenzierungsoperator* **&** benutzt. Im Beispiel teilt der Referenzierungsoperator dem Compiler mit: „Übergebe mir die Adresse der Variablen x aber nicht den Wert von x selbst „.
Nach der Zuweisung enthält ptrx die Speicheradresse von x. Eventuell werden Sie im Programm den Wert des Objekts benötigen, auf das ptrx zuweist. Dazu fällt Ihnen vielleicht folgender Code ein:

```
int z = ptrx; // FALSCH!!!
```

Dies ist leider falsch und funktioniert nicht. Grund: es wird versucht, einer regulären Variablen eine Speicheradresse zuzuweisen. An dieser Stelle bekommt man folgende Compiler Fehlermeldung *„Invalid conversion from int* to int“.* Dies ist verständlich, da Sie es hier mit zwei ganz verschiedenen Variablentypen zu tun haben. Deshalb muss der Zeigerverweis mit Hilfe des *Dereferenzierungsoperators* * aufgelöst werden:

```
int z = *ptrx;
```

Der Dereferenzierungsoperator stellt also praktisch das Gegenteil des Referenzierungsoperators dar. In diesem Fall wollen Sie nicht den eigentlichen Wert von ptrx, da dies eine Speicheradresse ist, sondern den Wert des Objekts auf das die Speicheradresse verweist. Der Dereferenzierungsoperator teilt dem Compiler mit: „Übergib mir den Wert des Objekts, auf das ptrx zeigt und nicht den eigentlichen Wert von ptrx“.

Wichtig:

Einen Zeiger zu dereferenzieren bedeutet den Inhalt des Speicherbereichs (des Objekts) zu ermitteln, auf den der Zeiger zeigt. Der *-Operator dient dazu, einen Zeiger zu deklarieren (int *x;) und zu dereferenzieren (int z = *x;).

Warnung:

Nicht initialisierte Zeiger enthalten, wie alle anderen nicht initialisierten Variablen, Zufallswerte. Der Versuch, einen nicht initialisierten Zeiger zu verwenden, kann ein Programm zum Absturz bringen. In den meisten Fällen werden Zeiger daher bei der Deklaration initialisiert.
Sehr oft werden Zeiger erst deklariert und erst später im Programm initialisiert. Wenn der Zeiger aber verwendet wird, bevor er initialisiert wurde, wird er auf eine zufällige Speicherposition zeigen. Eine Änderung dieses Speicherplatzes kann viele unvorhersehbare Folgen haben. Oft wirkt sich die Änderung von unbekanntem Speicher erst später aus, so dass der Fehler dann nicht mehr zuzuordnen ist. Um sicherzugehen, sollten Sie einen Zeiger daher bei seiner Deklaration mit 0 initialisieren.

```
Typ *zeiger = 0; // gleichbedeutend mit      Typ *zeiger = NULL;
```

Wenn Sie versuchen, einen NULL-Zeiger (jeder Zeiger, der auf NULL oder 0 gesetzt ist) zu verwenden, erhalten Sie vom Betriebssystem eine Fehlermeldung, dass eine Zugriffsverletzung vorliegt. Auch wenn dies nicht besonders gut klingt, ist es doch das geringere von zwei Übeln. Es ist wesentlich besser, direkt beim Auftreten eines Fehlers eine Meldung zu erhalten, als sich später mit einem nicht mehr zuzuordnenden Problem konfrontiert zu sehen.

2.3 Zugriffsmöglichkeiten bei Zeigern

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

int main()
{
    int x=18, *z;
    z=&x;
    cout << "eigene Adresse des Zeigers z = " << &z << endl;
    cout << "Wert der Variablen x: " << *z << endl;
    cout << "Die in z gespeicherte Adresse der Variablen x ist: " << z << endl;
    cout << "Adresse der Variablen x ist: " << &x << endl;
    system("pause");
}
```

Erläuterung:

- int x = 18, *z; // Deklaration einer Variable x und eines Zeigers z
  z = &x; // z erhält als Wert die Adresse von x zugewiesen

- Zugriff über den Namen z // Wert von z ist die Adresse von x
  Beispiel: cout << z; Ausgabe: 0x28ff44 // Adressen sind nur beispielhaft und werden vom Compiler vergeben

- Zugriff über den Dereferenzierungsoperator *z // Wert der Variablen x
  Beispiel: cout << *z; Ausgabe: 18

- Zugriff über den Adressoperator // eigene Adresse des Zeigers
  Beispiel: cout << &z; Ausgabe: 0x28ff66

2.4 Zeiger auf Felder

Zeiger und Felder (engl. Arrays) sind im Gebrauch sehr ähnlich. Nachfolgend werden die Unterschiede zusammengefasst:

- Ein Zeiger hat einen Speicherplatz, der einen Wert enthält, der als Adresse benutzt werden kann.
- Ein Array (d.h. Feldname) besitzt in diesem Sinne keinen Speicherplatz. Ein Array ist ein symbolischer Name für die Adresse (= den Anfang) eines Speicherbereichs. Wird der Name einer Feldvariable ohne den Indizierungsoperator ([ ]) verwendet, wird die Adresse des ersten Feld-Elements zurückgeliefert.

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main()
{
int feld[]={5,10,15,20,25};

cout << "Die Adresse des ersten Feldes ist: " << feld << endl;
// oder
cout << "Die Adresse des ersten Feldes ist: " << &feld[0] << endl;

cout << "Der Wert des ersten Feldelementes ist: " << *feld << endl;
// oder
cout << "Der Wert des ersten Feldelementes ist: " << feld[0] << endl;

system("pause");
}
```

Sie können auf die einzelnen Elemente eines Feldes mit dem Indizierungsoperator ([ ]) zugreifen.

```
int feld[]={5,10,15,20,25};
int eineVariable = feld[2]; // der Wert des 3. Elements (Index beginnt bei O!) ist 15
```

Das gleiche könnten wir mit einem Zeiger erreichen:

```
int feld[]={5,10,15,20,25};
int *ptr = feld;       // Deklaration eines Zeigers mit Namen ptr und Zuweisung der
                       // Adresse des ersten Feldelements
int eineVariable = ptr[2]; // Der Variable wird der Wert des 3. Feldeelements zugewiesen
```

In diesem Beispiel wird der Speicherplatz des ersten Feldelements einem Zeiger `ptr` zugewiesen. Beachten Sie, dass der Zeiger vom Datentyp `int` ist und ein Dereferenzierungsoparator (*-Symbol) für die Deklaration des Zeigers verwendet wird. Nach der Zuweisung enthält der Zeiger die Basisadresse des Feldes im Speicher und verweist somit auf dieses. Bei Zeigern auf Felder darf bei der Benutzung des Indexes **kein** Dereferenzierungsoperator verwendet werden.

2.5 Referenz

Eine Referenz ist eine besondere Art von Zeiger, die es ermöglicht, einen Zeiger wie ein reguläres Objekt zu behandeln. Referenzen werden mit dem Referenzierungsoperator (&) deklariert.

Syntax: Typ &name; oder Typ& name;

Folgendes Beispiel zeigt die Deklaration der Referenz.

```
int x;
int &rx = x;        // identisch mit   int& rx = x;
rx = 10;            // bewirkt das gleiche wie   x = 10;
```

Referenzen weisen einige Eigenheiten auf, welche sie für viele Fälle ungeeignet machen. So können beispielsweise Referenzen nicht deklariert werden und dann erst später einen Wert zugewiesen bekommen. Somit **müssen Referenzen gleich** bei der **Deklaration initialisiert** werden.

Hinweise: Der Gebrauch von Referenzen (außer bei Übergabe- und Rückgabeparametern) sollte aus Gründen der Verständlichkeit weitgehend unterbleiben.

2.6 Funktionsparameter als Zeiger oder als Referenz (call-by-reference)

Bei der Parameterübergabe mittels “call-by-reference” wird eine Referenz auf den aktuellen Parameter (also im Prinzip die Variable selbst und keine Kopie) verwendet.

2.6.1 Parameterübergabe als Referenz

Soll ein übergebenes Objekt modifiziert werden, kann die Übergabe durch eine **Referenz** des Objekts geschehen. Der Aufruf ist syntaktisch gleich wie bei der Parameterübergabe als Value (call-by-value). Dabei wird jedoch anstatt mit einer Kopie direkt mit dem Original gearbeitet. Die vorgenommenen Änderungen innerhalb der Funktion wirken sich direkt auf das Original aus. Es wird also keine Kopie angelegt. Wenn keine Änderung des Originals erwünscht ist und auch nicht mit call-by-value gearbeitet wird, so kommt die Übergabe eines Objekts als Referenz auf const in Frage. Die Parameterliste könnte z.B. *const TYP &objekt* lauten. Auf dieses Objekt kann innerhalb der Funktion nur lesend zugegriffen werden. Dieses wird auch vom Compiler geprüft.

2.6.2 Parameterübergabe als Zeiger

Die Übergabe als Zeiger ist im Prinzip die gleiche wie eine Übergabe als Referenz. Der Unterschied zwischen einem Zeiger und einer Referenz als Parameter ist zunächst sehr gering. Auf den ersten Blick sehen wir nur syntaktische Unterschiede. In der Praxis ist es oft möglich das eine durch das andere zu ersetzen. Der Unterschied liegt darin, dass einem Zeiger immer ein anderer Wert zugewiesen werden kann und damit auf eine andere Variable verwiesen werden kann. Eine Referenz ist dagegen der Stellvertreter für die übergebene Variable und kann nach der Parameterübergabe nicht mehr auf eine andere Zielvariable verwiesen werden.

2.6.3 Beispiel Call-By-Values vs. Call-By-Reference

Folgend finden Sie einen Vergleich zwischen call-by-value und call-by-reference als Referenz und als Zeiger.

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main()
{
   int a = 10, b = 13;
   int *c, *d;
   c = &a;
   d = &b;
   cout << “a: “ << a << “, b: “ << b << endl;
   system("pause");
   return 0;
}
```

	call by value	**call by reference (als Referenz)**	**call by reference (als Zeiger)**

Funktions-aufruf	tausche(a, b);	tausche(a,b);	tausche(&a, &b);
			oder
			tauschen(c,d);

Funktions-kopf und -rumpf	void tausche(int x, int y)
	{
	int tmp = x;
	x = y;
	y = tmp;
	}
		void tausche(int &x,int &y)
		{
		int tmp = x;
		x = y;
		y = tmp;
		}
			void tausche(int *x,int *y)
			{
			int tmp = *x;
			*x = *y;
			*y = tmp;
			}

Ausgabe	a: 10, b: 13	a: 13, b: 10	a: 13, b: 10

2.7 Zeiger auf Zeiger

Neben Zeigern auf Variablen und Funktionen, lernen Sie jetzt Zeiger auf Zeiger kennen.

Syntax: *TYP **zname*;

Erläuterung: Der Zeiger zname zeigt auf einen Zeiger, der wiederum auf einen Variablentyp TYP zeigt.

Beispiel: Zeiger auf Zeiger

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
   int x=10, y=10, *xptr, **ptrptr; // Deklaration der Variablen x,y des Zeigers xptr und des Zeigers auf einen Zeiger ptrptr

   cout << "x = " << x << endl; // Ausgabe von x als Wert

   xptr = &x; // Dem Zeiger xptr wird die Adresse von x als Wert zugewiesen
   *xptr = 20; // Der Wert der Variablen auf die xptr zeigt wird auf 20 gesetzt
   cout << "x = " << x << " und *xptr = " << *xptr <<endl;

   ptrptr = &xptr; // Dem Zeiger ptrptr wird die Adresse des Zeigers xptr zugewiesen
   **ptrptr = 30; // Die Variable x erhält über den Zeiger **ptrptr den Wert 30
   cout << "x = " << x << " *xptr = "<< *xptr << " und **ptrptr = " << **ptrptr << endl;

   *ptrptr = &y; // der Zeiger zeigt jetzt auf y
   **ptrptr = 40; // die Variable y erhält den Wert 40, da ptrptr auf ptr zeigt und ptr nun auf y
   cout << "x = " << x << ", y= " << y << ", *xptr = " << *xptr << " und **ptrptr = " << **ptrptr << endl;
   system("pause");
}
```

2.8 Elementoperatoren

Für den direkten Zugriff auf Komponenten (Attribute und Methoden) einer Klasse werden folgende Elementoperatoren verwendet: „.“ und „->“. Der Unterschied zwischen den beiden Operatoren besteht darin, dass „.“ auf eine Variable eines Klassentyps angewandt wird, während „->“ für Zeiger auf Klassentypen benutzt wird.

Beispiel:

```
#include <iostream>
#include <cstdlib>
using namespace std;
class Geburtstag
{
   private:
      int jahr, monat, tag;
   public:
      Geburtstag(int j, int m, int t): jahr(j), monat(m), tag(t) {}     // Konstruktor mit Elementinitialisierungsliste
      Geburtstag() {}                                                   // Standardkonstruktor

      void setJahr(int j) {  jahr = j; }
      void setMonat(int m) { monat = m; }
      void setTag(int t) { tag = t; }
      void datenausgeben() { cout << tag << "-" << monat << "-" << jahr << endl; }
};
int main()
{
   Geburtstag Heinz;
   Geburtstag *Peter;

   Heinz.setJahr(1964);
   Heinz.setMonat(1);
   Heinz.setTag(22);

   Peter = &Heinz;
   Peter->setMonat(12);
   Heinz.datenausgeben();
   Peter->datenausgeben();
   system("pause");
}
```

2.9 Beispiel

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

int main()
{
   int x = 3;
   int *zeiger = &x;
   cout << "Adresse worauf Zeiger zeigt: " << zeiger << endl;
   cout << "Adresse von x: " << &x << endl;
   cout << "Adresse von Zeiger: " << &zeiger << endl << endl;

   cout << "Wert von x: " << x << endl;
   cout << "Wert von Zeiger: " << *zeiger << endl << endl;
   cout << "Wert fuer x eingeben: ";

   cin >> x;
   cout << endl;
   cout << "Wert von x: " << x << endl;
   cout << "Wert von Zeiger: " << *zeiger << endl << endl;

   cout << "Wert fuer zeiger eingeben: ";
   cin >> *zeiger;
   cout << endl;
   cout << "Wert von x: " << x << endl;
   cout << "Wert von Zeiger: " << *zeiger << endl << endl;

   system("pause");

   int array[5] = {1,2,3,4,5};

   cout << "Adresse vom ersten Arrayfeld: " << array << endl;
   cout << "Adresse vom ersten Arrayfeld: " << &array << endl;
   cout << "Adresse vom ersten Arrayfeld: " << &array[0] << endl;

   cout << endl;
   cout << "Wert vom ersten Arrayfeld: " << array[0] << endl;
   cout << "Wert vom ersten Arrayfeld: " << *array << endl << endl;

   cout << endl;
   cout << "Wert vom dritten Zeigerfeld: " << array[2] << endl;
   //cout << "Wert vom dritten Zeigerfeld: " << *arrayzeiger[2]; // funktioniert nicht!
   cout << "Wert vom dritten Zeigerfeld: " << *(array+2) << endl;

   cout << "Adresse von Array anzeigen: "<<endl;
   for(int i = 0; i < 5; ++i)
   {
      cout << "&array[" << i << "] = " << &array[i] << endl;
   }

   cout << endl;
   cout << "Wert von Array anzeigen: "<<endl;
   for(int i = 0; i < 5; ++i)
   {
      cout << "array[" << i << "] = " << array[i] << endl;
   }
   system("pause");

   // ZUSAMMENFASSUNG:
   // &array[0] ist das gleiche wie array
   // *array ist das gleiche wie array[0]
   // &array[1] ist das gleiche wie array+1
   // *(arrayzeiger+1) ist das gleiche wie array[1]
}
```

3 Verwenden von Objekten

3.1 Der this-Zeiger

Eine Funktion bzw. eine Methode kann auf jedes Element eines Objekts zugreifen, ohne das dabei explizit ein Objekt anzugeben ist. Wie wir bereits wissen, arbeitet eine Funktion stets mit dem Objekt, für welches sie aufgerufen wurde. Aber woher weiß die Funktion, mit welchem Objekt sie momentan arbeitet? Sobald die Funktion aufgerufen wird, erhält diese als „verstecktes“ Argument die Adresse des aktuellen Objekts. Die Adresse des aktuellen Objekts steht in der Funktion als konstanter Zeiger *this* zur Verfügung. Der Name *this* ist in C++ ein Schlüsselwort. Innerhalb einer Methode bezeichnet this einen Zeiger auf das aktuelle Objekt und **this* das Objekt selbst, für das die Methode aufgerufen wird. **this* ist nichts anderes als nur ein anderer Name (sog. Alias) für das Objekt, welches innerhalb der Elementfunktion benutzt werden kann.

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

class Test {
     private:
        int t1, t2;
     public:
        void tueWas(int a, int b) {
            this->t1 = a + b; // this kann an dieser Stelle auch weggelassen werden
            this->t2 = a - b;  // this kann an dieser Stelle auch weggelassen werden
        }
};
int main()
{
   Test t;
   t.tueWas(7,8);
   system("pause");
   return 0;
}
```

Innerhalb einer Methode kann mit Hilfe des this-Zeigers ein Attribut bzw. Element eines Objektes wie folgt angesprochen werden:

```
this->t2;                  // Datenelement t2
this->datenausgeben();     // Funktionsaufruf
```

Der Compiler erzeugt implizit stets einen solchen Ausdruck, wenn nur ein Element des aktuellen Objekts angegeben ist.

```
t2 = 10;                   // entspricht this->t2 = 10;
```

Die explizite Angabe des this-Zeigers kann z.B. eingesetzt werden, um lokale Variablen einer Methode von Klassenelementen zu unterscheiden.

```
Person (string name, string vorname, int gebJahr) {
            this->name = name;
            this->vorname = vorname;
            this->gebJahr = gebJahr;
      }
```

Erläuterung:
Die Variablennamen in der Parameterliste heißen genauso, wie die Klassenelemente (name, vorname, gebJahr). Um besser zu unterscheiden was Klassenelement und was Übergabeparameter ist, wird der this-Zeiger explizit angegeben.

Die Verwendung des this-Zeigers ist dann notwendig, wenn das aktuelle Objekt als Ganzes angesprochen wird, nämlich **this*. Dieses passiert am häufigsten wenn das aktuelle Objekt als Kopie oder per Referenz zurückgegeben wird. Diese Anweisung lautet:

```
return *this;
```

3.2 Objekte als Argumente

```
#include <cstdlib>
#include <iostream>

using namespace std;

class Person {
    private:
        string name;
        string vorname;
        int gebJahr;
    public:
        Person() { }
        Person(string n, string v, int j): name(n), vorname(v), gebJahr(j) {}

        void datenausgeben()
        {
            cout << "Name:" << name << " Vorname: " << vorname << " Geburtsjahr: " << gebJahr << endl;
        }

        string werIstAelter(Person p, Person r) // Objekte vom Typ Person werden als Argument vergeben
        {
            string retString = "";
            if (p.gebJahr < r.gebJahr)
            {
                retString = p.name + " ist \x84lter als  " + r.name;
            }
            else if (r.gebJahr < p.gebJahr)
            {
                retString = r.name + " ist \x84lter als " + p.name;
            }
            else
            {
                retString = p.name + " und " + r.name + " sind gleich alt.";
            }
            return retString;
        }
};

int main()
{
    Person p("Meier", "Herbert", 1988);
    Person r("Muster", "Frau", 1965);
    cout << p.werIstAelter(p,r) << endl;

    system("pause");
    return 0;
}
```

3.3 Objekt als Return-Wert

Objekte können entweder als Kopie, Referenz oder Zeiger zurückgegeben werden. Wir werden uns alle drei Möglichkeiten näher anhand von Beispielen anschauen.

Rückgabe einer Kopie

Die Rückgabe eines Objekts in Form einer Kopie ist für „kleine“ Objekte sinnvoll, da diese sehr aufwendig ist.

Beispiel:

```
#include <cstdlib>
#include <iostream>

using namespace std;

class Person {
    private:
        string name;
        string vorname;
        int gebJahr;
    public:
        Person() { }
        Person(string n, string v, int j): name(n), vorname(v), gebJahr(j) {}

        void datenausgeben()
        {
            cout << "Name:" << name << " Vorname: " << vorname << " Geburtsjahr: " << gebJahr << endl;
        }
        Person update() {
            Person pers;
            cout << endl;
            cout << "Name: ";
            cin >> pers.name;
            cout << endl << "Vorname: ";
            cin >> pers.vorname;
            cout << endl << "Geburtsjahr: ";
            cin >> pers.gebJahr;
            return pers;                // Objekt (jedoch nur eine Kopie) vom Typ Person zurückgeben
        }
};

int main()
{
    Person p("Meier", "Herbert", 1988);

    p = p.update();
    p.datenausgeben();

    system("pause");
    return 0;
}
```

Wenn die Funktion verlassen wird, wird das lokale Objekt pers zerstört. Deshalb wird durch den Compiler eine temporäre Kopie des lokalen Objekts angelegt, welches der aufzurufenden Funktion zur Verfügung steht.

Rückgabe einer Referenz

Viel effizienter ist, wenn der Return-Wert eine Referenz auf das Objekt ist. Zu beachten ist hierbei jedoch, dass die Lebensdauer eines Objekts, auf welches eine Referenz zurückgegeben wird, nicht lokal sein darf. Ansonsten wird das Objekt beim Verlassen der Funktion zerstört, so dass die zurückgegebene Referenz ungültig ist. Wenn das Objekt innerhalb der Funktion definiert ist, so soll es als static deklariert werden.

Das folgende Beispiel zeigt wie man von dieser Möglichkeit Gebrauch macht. Die Funktion liefert eine Referenz mit dem aktuellen Alter einer Person, welches bei jedem Aufruf der Funktion neu berechnet wird. Das Objekt p ruft die Methode der Klasse Person updateAlter() auf, aktualisiert das Objekt und gibt das Alter auf dem Bildschirm aus.

Beispiel:

```
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

class Person {
    private:
        string name;
        string vorname;
        int gebJahr;
        int alter;
    public:
        Person() { }
        Person(string n, string v, int j): name(n), vorname(v), gebJahr(j) {}
        int getAlter()
        {
            return alter;
        }

        const Person& updateAlter()
        {
            static Person pers;
            //Aktuelles Jahr ermitteln
            time_t zeitstempel; // Aktuelle Datum und Uhrzeit in Variable speichern
            tm *datum; // Pointerobjekt, welches Datum und Uhrzeit enthält
            zeitstempel = time(0); // Zuweisung der Anzahl der Sekunden zu, die seit dem 01.01.1970 verstrichen sind
            datum = localtime(&zeitstempel); // localtime liefert Zeiger zurück, welches die aktuelle Zeitangabe enthält
            int aktJahr = datum->tm_year+1900; // Aktuelles Jahr wird ermittelt und gesetzt (Ausgegangen von 1900)
            pers.alter = aktJahr - gebJahr;
            return pers; // Objekt als Referenz zurückgeben
        }
};

int main()
{
    Person p("Meier", "Herbert", 1988);
    p = p.updateAlter();
    cout << "Alter: " << p.getAlter() << endl;

    system("pause");
    return 0;
}
```

Zeiger als Return-Wert

Statt einer Referenz kann auch ein Zeiger auf ein Objekt zurückgegeben werden. Auch in diesem Fall muss garantiert sein, dass nach verlassen der Funktion das Objekt noch existiert.

Beispiel:

```
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;

class Person {
    private:
        string name;
        string vorname;
        int gebJahr;
        int alter;
    public:
        Person() { }
        Person(string n, string v, int j): name(n), vorname(v), gebJahr(j) {}
        int getAlter()
        {
            return alter;
        }
        Person* updateAlter()
        {
            static Person pers;
            // Aktuelles Jahr ermitteln
            time_t zeitstempel; // Aktuelle Datum und Uhrzeit in Variable speichern
            tm *datum; // Pointerobjekt, welches Datum und Uhrzeit enthält
            zeitstempel = time(0); // Zuweisung der Anzahl der Sekunden zu, die seit dem 01.01.1970 verstrichen sind
            datum = localtime(&zeitstempel); // localtime liefert Zeiger zurück, welches die aktuelle Zeitangabe enthält
            int aktJahr = datum->tm_year+1900; // Aktuelles Jahr wird ermittelt und gesetzt (Ausgegangen von 1900)
            pers.alter = aktJahr - gebJahr;
            return &pers; // Als Zeiger auf das Objekt pers zurückgeben
        }
};

int main()
{
    Person *p = new Person("Meier", "Herbert", 1988);
    p = p->updateAlter();
    cout << "Alter: " << p->getAlter() << endl;
    system("pause");
    return 0;
}
```

4 Speicherreservierung zur Laufzeit

Sinnvoll werden Zeiger erst dann, wenn Objekte dynamisch im Speicher erzeugt werden. In solchen Fällen ist ein Zeiger notwendig, um auf das Objekt zuzugreifen. Bisher haben Sie gelernt wie Objekte lokal bzw. statisch angelegt werden, das bedeutet, der Speicher für die Variable oder das Objekt wurde auf dem Stack reserviert. In diesem Kapitel werden Sie eine neue Möglichkeit der Verwaltung der Objekte z.B. in verketteten Listen kennenlernen.

4.1 Lokale versus dynamische Speicherbelegung

Den gesamten Speicherplatz, den ein Programm für lokale Variablen, Funktionsaufrufe etc. benötigt, entnimmt es dem Stack. Dieser Speicher wird nach Bedarf reserviert und wieder freigegeben. In der Regel geschieht dies, wenn die Programmausführung in eine Funktion oder einen anderen Codeblock springt. Für die lokalen Variablen einer Funktion wird der Speicherplatz reserviert, wenn das Programm die Funktion aufruft. Wird die Funktion verlassen, wird der für die Funktion reservierte Speicher wieder freigegeben. All dies geschieht automatisch, so dass Sie sich keine Gedanken um die Speicherbelegung oder die Freigabe des Speichers machen müssen.

Ein Vorteil der lokalen Speicherbelegung kann z.B. sehr schnelle Reservierung der Speicher des Stacks genannt werden. Ein Nachteil ist, dass der Stack eine feste Größe hat, die während der Programmausführung nicht geändert werden kann. Sollte ihr Programm einmal über zu wenig Stack-Speicher verfügen, kann dies merkwürdige Folgen zeigen. Ihr Programm könnte abstürzen, unerklärliche Dinge tun oder aber scheinbar normal verlaufen und erst bei Programmende abstürzen.

Beispiel einer lokalen Speicherbelegung:

```
Person p[100]; // dabei werden 100 Objekte der Klasse Person angelegt und für alle 100 der Speicherplatz reserviert
```

Als Nachteil der statischen Programmierung kann die „Speicherplatzverschwendung“ genannt werden. Wie im oberen Beispiel zu sehen ist, haben wir Speicherplatz für 100 Personen reserviert. Was ist, wenn wir aber keine 100 Objekte benötigen, sondern lediglich 5 Objekte verwaltet wollen? Somit haben wir Speicherplatz für die weiteren 95 Objekte verschwendet.

Für Variablen der elementaren Datentypen und kleine Felder gibt es keinen Grund etwas anderes als eine lokale Speicherbelegung vorzunehmen. Wenn Sie aber große Felder, Strukturen oder Klassen einsetzen, werden Sie wahrscheinlich auf die dynamische Speicherbelegung von Speicher auf dem Heap zurückgreifen. Dieser umfasst den freien physikalischen RAM Ihres Computers, sowie sämtliche freien Festplattenspeicher. In anderen Worten, Sie können auf einem typischen Windows-System leicht auf frei verfügbaren Heap-Speicher von 100MByte kommen. Der Vorteil besteht darin, dass Sie damit praktisch unbegrenzt Speicher für Ihre Programme zur Verfügung haben. Dafür müssen Sie allerdings in Kauf nehmen, dass bei dynamisch reservierten Speicher zusätzlicher Overhead anfällt, wodurch die Ausführung insgesamt etwas langsamer abläuft als bei Speicher, der auf dem Stack reserviert wurde. In den meisten Programmen wird sich dieser zusätzliche Overhead jedoch kaum bemerkbar machen.
Ein weiterer Nachteil der dynamischen Speicherbelegung ist der erhöhte Arbeitsaufwand für den Programmierer.

4.2 Dynamische Speicherverwaltung (new/delete)

4.2.1 Der Operator new

Wie bereits angesprochen, kann Speicher lokal (auf dem Stack) oder dynamisch (auf dem Heap) reserviert werden. In C++ Programmen wird dynamischer Speicher mit Hilfe des Operators *new* reserviert und mit delete wieder freigegeben.

Im Folgenden finden Sie Beispiele für die Speicherbelegung von zwei Feldern. Ein Feld wird auf dem Stack angelegt (lokale Speicherbelegung) und das andere auf dem Heap (dynamische Speicherbelegung).

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

class Person {

    private:
        string name;
        string vorname;
        int gebJahr;
    public:
        Person() { cout << "Standardkonstruktor" << endl; }
        Person(string n, string v, int j): name(n), vorname(v), gebJahr(j)
        { cout << "Elementinitialisierungskonstruktur" << endl; }
        void ausgabe(){ cout << "Name: " << name << ", Vorname: " << vorname << " , Geb.Jahr: " << gebJahr << endl; }
};
int main() {
    int anzahl = 10;
    // Speicherplatz wird auf dem Stack reserviert
    Person personen[10]; // Standardkonstruktor wird 10x aufgerufen
    // Speicherplatz wir auf dem Heap reserviert
    // Zeiger auf Person
    Person *zpersonen;
    // 1. Möglichkeit
    zpersonen = new Person;
    // oder 2. Möglichkeit
    zpersonen = new Person("Meier", "Heinz", 1977);
    // oder 3. Möglichkeit
    zpersonen = new Person[anzahl];
    // oder 4. Möglichkeit
    Person *per[10]; // Standardkonstruktor wird 10x aufgerufen
    char ant='j';
    string name="", vorname="";
    int gebJahr=0;
    for (int i = 0; i < anzahl; i++) {
        cout << "Neue Person anlegen (j / n)?" << endl;
        cin >> ant;
        if (ant == 'j') {
            cout << "Name eingeben: ";
            cin >> name;
            cout << endl << "Vorname eingeben: ";
            cin >> vorname;
            cout << endl << "Geburtsjahr: ";
            cin >> gebJahr;
            per[i] = new Person(name, vorname, gebJahr);
        } else {
            break;
            }
        }
        for(int j = 0; j < i; j++)
        {
            per[j]->ausgabe();
        }
    system("pause");
    return 0;}
```

Wirkung des *new* Operators:

- 1. Möglichkeit – **new** reserviert Speicherplatz für ein Objekt vom Typ Person und initialisiert dieses Objekt durch Aufruf des Standardkonstruktors (Default-Konstruktor).
- 2. Möglichkeit – hier wird ebenfalls ein Objekt vom Typ Person angelegt. Aber es wird der Konstruktor mit passender Initialisierungsliste (Elementinitialisierungskonstruktor) aufgerufen.
- 3. Möglichkeit – new reserviert Speicherplatz für eine bestimmte Anzahl von Personen. Anzahl darf eine Variable eines Integer-Typs sein. Es kann hier nur der Default-Konstruktor zur Initialisierung der Personen angewandt werden. Er wird automatisch für jedes Element des Feldes aufgerufen.
- 4. Möglichkeit – Sie haben auch die Möglichkeit, ein Feld mit Objekten vom Typ Person zu deklarieren. Die einzelnen Objekte können dann einzeln dynamisch mit new angelegt werden. Diese Lösung ist z.B. sinnvoll, wenn Sie die einzelnen Objekte über Schleife anlegen wollen.

Der Datentyp des Rückgabewertes von *new* ist ein typisierter Zeiger auf den zu reservierenden Datentyp.

Fehlerprüfung:

```
// Fehlerprüfung
Person * pK = new Person;
if ( pK == 0 )
{
  // cerr wird verwendet wie cout und dient zur Ausgabe von Fehlermeldungen
   cerr << "out of memory error" << endl;
   exit (1);
}
```

Der Operator *new* ruft automatisch den passenden Konstruktor mit geeigneter Initialisierungsliste auf, um das neu angelegte Objekt zu initialisieren. Falls ein Feld von n Objekten angelegt wird, kann nur der Default-Konstruktor ohne Argumente verwendet werden.

4.2.2 Der Operator delete

Speicher, den Sie reserviert haben, müssen Sie auch wieder freigeben, wenn Sie ihn nicht mehr benötigen. Bei lokalen Objekten erfolgt dies automatisch. Der Speicher-Manager reserviert den von Ihren Objekten benötigten Speicher auf dem Stack und gibt ihn wieder frei, wenn das Objekt seinen Gültigkeitsbereich verliert (i.d.R. wenn die Funktion abgearbeitet ist oder wenn der Anweisungsblock, in dem das Objekt deklariert wurde, endet). Wenn Sie sich für die dynamische Speicherbelegung entscheiden, übernehmen Sie als Programmierer die Verantwortung dafür, dass der mit dem new-Operator reservierte Speicher auch wieder freigegeben wird.

Mit dem Operator delete, der ebenfalls ein **Schlüsselwort** in C++ ist, wird der vorher mit new reservierte **Speicherplatz wieder freigegeben** und damit die zugehörigen **Objekte zerstört**.

Beispiel: Verwendung des delete-Operators

```
Person *person = new Person;
// tue was
delete person; // das Objekt wird zertört, d.h. der Speicherplatz wird wieder freigegeben
```

Erläuterung: *delete* gibt den Speicherplatz für das ein mit *new* reservierte Objekt wieder frei. Vorab wird der Destruktor des Objekts aufgerufen.

Sie können auch mehrere Objekte über ein Feld anlegen:

```
Person *personen = new Person[10];
```

Dynamisch angelegte Felder, müssen Sie wieder korrekt löschen:

```
delete [] personen;
```

Erläuterung: hier wird das gesamte reservierte Feld wieder freigegeben. Für jedes einzelne Element des Feldes wird der Destruktor aufgerufen.

delete gibt den Speicher auf den der angegebene Zeiger verweist wieder frei. Es darf nur auf einen Zeiger angewandt werden, der vorher mit new erzeugt wurde. Eine Ausnahme bildet der NULL-Zeiger. Auf ihn kann *delete* ohne Konsequenzen angewandt werden.

4.2.3 Regeln für Zeiger und dynamische Speicherbelegung

- Stellen Sie sicher, dass ein Zeiger, der nicht direkt verwendet wird, mit 0 initialisiert wird.
- Achten Sie darauf, dass Sie keinen Zeiger zweimal löschen.
- Sie können bedenkenlos Zeiger löschen, die auf NULL oder 0 gesetzt sind.
- Setzen Sie Zeiger, nach dem Löschen des Objektes auf das sie zeigen, auf NULL oder 0.
- Um auf das Objekt, auf das ein Zeiger verweist, zuzugreifen, müssen Sie den Zeiger dereferenzieren.

5 Verkettete Listen

5.1 Einfach verkette Liste

Ein wesentliches Problem der Arrays in C++ besteht darin, dass ein einmal angelegtes Feld – und das unabhängig davon, ob Sie es statisch oder dynamisch reservieren – seine Größe nicht mehr verändern kann. Allerdings wären Programme höchst unflexibel, gäbe es nicht auch für dieses Problem eine sehr elegante Lösung. Die Idee ist, nicht eine bestimmte, fest vorgegebene Menge von Objekten anzulegen, sondern einzelne Objekte mit Hilfe einer oder mehrerer Zeiger zu verknüpfen. Dadurch werden größere, zusammenhängende Datenstrukturen beliebig verteilter Objekte aufgebaut. Dynamische Datenstrukturen werden erst zur Laufzeit eines Programms aufgebaut.

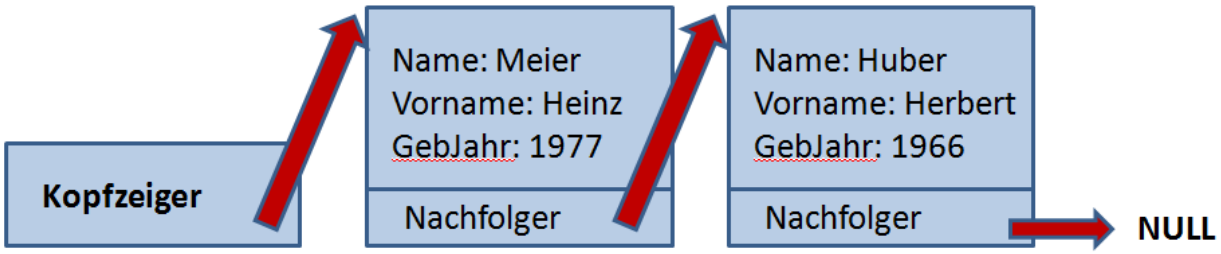
Einer der bekanntesten oder sogar die bekannteste dynamische Datenstruktur ist die einfach verkettete Liste. Sie besteht aus Elementen, die miteinander über Zeiger (i.d.R. Nachfolger genannt) verbunden sind.

Wir können eine Liste von beliebig vielen Personen aufbauen, indem wir einzelne Personen durch einen Verkettungszeiger miteinander verbinden. Genauer gesagt verweist eine Person mit Hilfe des Verkettungszeigers auf eine weitere.

Die Elemente einer Liste **müssen** aus **zwei Komponenten** aufgebaut werden:

- Dem eigentlichen Dateninhalt, also Attributen im Beispiel der Klasse Person
- Und einem Zeiger „Nachfolger“ auf ein weiteres Objekt vom Typ Person

Der Zeiger „Nachfolger“ deutet auf die nächste Zelle der gleichen Bauart – oder auf NULL, was das Ende der Liste kennzeichnet. Damit verweist jede Zelle dieser Liste auf ein nachfolgendes Element. Durch diese Technik der Verkettung sind die einzelnen eigenständigen Objekte zu einer einfach verketteten Liste zusammengefasst.



Wir benötigen lediglich noch einen Zeiger, der auf die erste Zelle verweist und damit den Listenanfang kennzeichnet. Die gesamte Liste wird durch diesen Zeiger auf das erste Element, den Kopfzeiger, repräsentiert. Prinzipiell können beliebig viele Elemente in einer Liste verkettet werden – im Extremfall kann die Liste den gesamten zur Verfügung stehenden Speicher ihres Computers belegen.

Wie müssen wir nun unsere Klasse Person erweitern, damit eine Liste von Personen aufgebaut werden kann. Wir benötigen ein neues Attribut vom Typ Zeiger auf die zu definierende Klasse Person. Das ist in C++ für Zeiger möglich, obwohl die Klasse Person gerade erst definiert wird.

5.1.1 Definition der Klasse Person als Listenelement

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

class Person
{
   friend class ListeVonPersonen; // friend Deklaration, damit die andere Klasse Zugriff auf die Private-Elemente hat
   // Attribute
   private:
      string name;
      string vorname;
      int gebJahr;
      Person* Nachfolger; // Verkettungszeiger
   // Methoden
   public:
      Person (string n, string v, int j): name(n), vorname(v), gebJahr(j) {}

      string getName () const { return name; }
      string getVorname () const { return vorname; }
      int getGebJahr() const { return gebJahr; }

      void berechneAlter () {
         //Aktuelles Jahr ermitteln
         time_t zeitstempel;
         tm *datum;
         zeitstempel = time(0);
         datum = localtime(&zeitstempel);
         int aktJahr = datum->tm_year+1900;

         cout << "Alter: " << aktJahr - gebJahr << " Jahre" << endl;
         }
      Person* next () { return Nachfolger; } // Gibt den Nachfolger meines aktuellen Objekts zurück

      void datenausgeben() {
          cout << "Name: " << name << ", Vorname: " << vorname;
          cout  << ", Geburtsjahr: " << gebJahr << endl;
      }
};
```

Damit erhält ein Objekt vom Typ Person die Möglichkeit auf ein anderes Objekt gleichen Typs zu verweisen. Diese Beziehung „zu sich selbst“ wird im Klassendiagramm durch eine Beziehung zu sich selbst dargestellt → reflexive Assoziation. Das soll versinnbildlichen, dass die Klasse Person die Eigenschaft hat, auf Objekte gleichen Typs zu verweisen.

Die Gesamtheit aller Personen bildet durch diese Verkettung eine zusammenhängende Struktur – einen Container für Personen. Wir können diesen Container auch als eigenständiges Objekt betrachten und eine Klasse – wir nennen sie – „ListeVonPersonen“ realisieren. Damit diese Klasse Zugriff auf den Verkettungszeiger von Person hat, haben wir sie in Person als „Freund“ deklariert.

Was sind die Attribute eines Objekts vom Typ „ListeVonPersonen“ und welche Methoden soll diese Liste bereitstellen? Wir wissen bereits, dass die gesamte Liste durch einen einzigen Zeiger – den Kopfzeiger – auf das erste Element repräsentiert werden kann.
Das wird das einzige Attribut unserer Listenklasse sein und die wichtigsten Methoden sind:

- Liste erzeugen – Konstruktor für leere Liste
- eine neue Person an den Anfang der Liste einfügen
- entfernen einer Person vom Anfang der Liste
- Überprüfung ob Liste leer

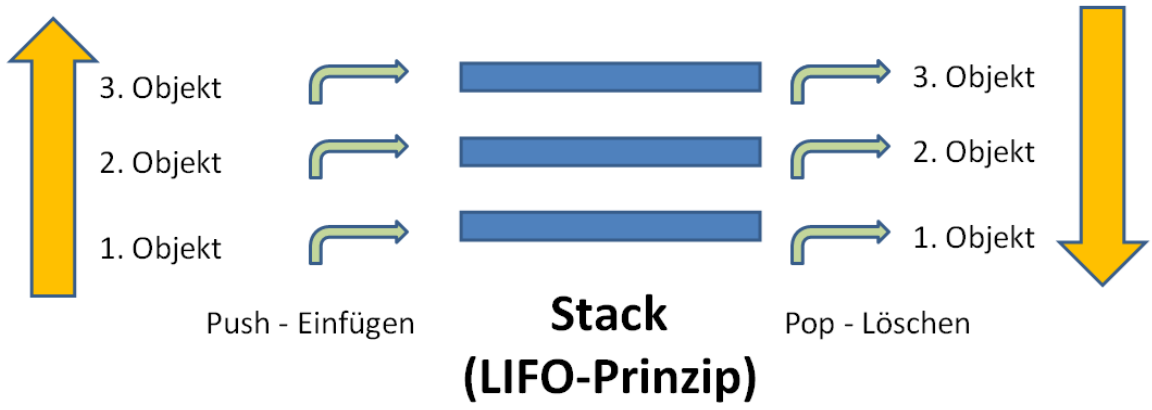
5.1.2 Definition der Klasse ListeVonPersonen

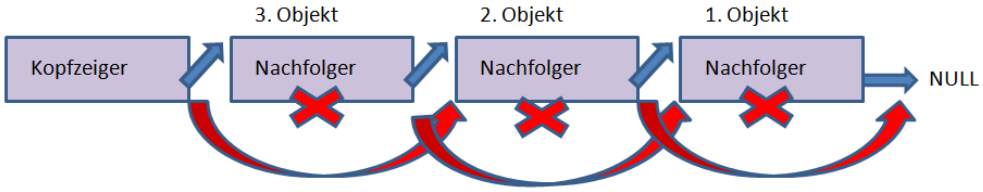
```
class ListeVonPersonen
{
   private:
          Person* Kopfzeiger;
   public:
          ListeVonPersonen (): Kopfzeiger(0) {};
          // an den Anfang der Liste einfügen
          void insert (Person * kp)
          {
                    kp->Nachfolger = Kopfzeiger; // Mein Nachfolger wird mein neuer Anfang
                    Kopfzeiger = kp; // und dessen Nachfolger ist das neue Objekt, welches ich erstellt habe
          }
          // Entfernen einer Person vom Anfang der Liste
          Person * remove ()
          {
                    if (Kopfzeiger != 0)
                              Kopfzeiger = tmp->Nachfolger;

                    return Kopfzeiger;
          }

          // Zugriff auf erste Person in der Liste
          Person * begin () { return Kopfzeiger; }

          // Überprüfen ob die Liste leer ist
          bool empty () const { return Kopfzeiger == 0; } // Liefert true falls Kopfzeiger 0 ist, ansonsten false
};
```

Wie Sie sehen, können die wichtigsten Methoden mit wenigen Programmcodezeilen realisiert werden. Der Standardkonstruktor erzeugt eine leere Liste, indem er den Kopfzeiger mit NULL initialisiert. Beim Einfügen eines neuen Elements an den Listenanfang müssen zwei Zeiger verändert werden:

- der Nachfolger des neu einzufügenden Elements und
- der Kopfzeiger der Liste muss nun auf das neue Element verweisen.

Mit einem derart einfachen Programm kann bereits eine beliebige Zahl an Personen verwaltet werden. Es fehlen lediglich noch weitere wichtige Methoden. Diese Liste ist nur in der Lage, bereits bestehende Personen zu verwalten. Wer übernimmt das Anlegen neuer Personen oder das Löschen nicht mehr benötigter Personen? Diese Liste kann nur bestehende Objekte verwalten, die vorher mit den Operatoren new erzeugt wurden und mit delete wieder freigegeben werden. Das Einfügen eines Objekts in die Liste erzeugt keine Kopie des Objekts.
Wir können in unserer Liste neue Elemente **nur** an den **Anfang** einfügen und auch wieder vom Anfang löschen. Eine sequentielle Datenstruktur, die den Zugriff nur an einem Ende ermöglicht, wird auch Stack genannt. Die Operation des Einfügens neuer Elemente bezeichnet der Stack als **push** und das Löschen als **pop**. Also auf einem Stack kann man nur oben was legen und von oben wieder was wegnehmen, d.h. der Stack arbeitet nach dem LIFO-Prinzip (last in, first out). Wir können die Elemente nur in Vorwärtsrichtung bearbeiten, weil die Information über das Vorgängerelement in den Elementen der Liste nicht vorhanden ist. Das ist der Nachteil der einfach verketteten Liste.





5.1.3 Hauptprogramm zur Personenverwaltung

```
int main ()
{
	ListeVonPersonen list; // Leere Liste erzeugen
	char ant = 'j';
	string name="", vorname="";
	int gebJahr=0;
	Person * kp;

	// Füllen der Liste
	while (ant != 'n')
	{
		cout << "Name: " << endl;
		cin >> name;
		cout << endl << "Vorname: " << endl;
		cin >> vorname;
		cout << endl << "Geburstsjahr: " << endl;
		cin >> gebJahr;
		kp = new Person(name, vorname, gebJahr);
		list.insert (kp);
		cout << "Weiter (j/n)" << endl;
		cin >> ant;
	}

	// Ausgabe der Liste
	for (kp=list.begin(); kp != 0; kp = kp->next()) // Beginne am Anfang der Liste; Gehe solange durch, bis NULL
		kp->datenausgeben();                    // erreicht wurde; Nach jedem Schleifendurchlauf ist mein neues
                                                // Objekt dessen Nachfolger
	// Leeren der Liste
	while (! list.empty ())                     // Solange nicht meine Liste leer ist
		list.remove ();                         // Lösche mir jedes einzelne Objekt (immer den Kopfzeiger)

	system("pause");
	return 0;
}
```

5.2 Sequentielle Container

Für einfach verkettete Listen gibt es bereits verschiedene Bibliotheken, welche genutzt werden können, um die Funktionalitäten einer Liste zu verwenden. Das einführende Beispiel verwendet das Schlüsselwort *list*. In diesem Programm werden positive Zahlen eingegeben. Diese werden in einer Liste abgespeichert und anschließend ausgegeben. Die Liste wird durch die bereits vorhandene Bibliothek <list> eingebunden und verwendet.

Beispiel: Liste

```
#include <iostream>
#include <list> // Bibliothek list

using namespace std;

int main()
{
	list<int> l;
	int x;
	cout << "Positive Zahl eingeben (Beenden mit 0):" << endl;
	do
	{
		cin >> x;
		if(x==0)
		   break;
		l.push_back(x); // Am Ende der Liste einfügen
	}
	while (true);

	list<int>::iterator i; // Iterator, welcher mir die einzelnen Zahlen der Liste ausliest

	for (i=l.begin(); i != l.end(); ++i)	// Beginne am Anfang des Vektors; Bis ich das Ende erreicht habe;
		cout << *i << " ";		// Erhöhe mir den Iterator um 1

	cout << endl;
	return 0;
}
```

Es besteht ebenfalls die Möglichkeit, die Liste zu sortieren. Dies kann man erreichen, indem man die in *list* definierte Funktion *sort* verwendet.
Die Sortierung wird durch den Befehl

```
sort();
```

ausgeführt.

Das erweiterte Programm sieht also folgendermaßen aus.

Beispiel: Liste sortieren

```
#include <iostream>
#include <list>                 // Bibliothek vector

using namespace std;

int main()
{
        list<int> l;
        int x;
        cout << "Positive Zahl eingeben (Beenden mit 0):" << endl;
        do
        {
                cin >> x;
                if(x==0)
                   break;
                l.push_back(x); // Am Ende der Liste einfügen
        }
        while (true);

        list<int>::iterator i; // Iterator, welches mir die einzelnen Zahlen im Vektor ausliest

        cout << "Vor dem Sortieren: " << endl;
        for (i=l.begin(); i != l.end(); ++i) // Liste durchgehen und ausgeben
                cout << *i << " ";

        cout << endl;

        l.sort(); // Sortiere mir die Liste

        cout << "Nach dem Sortieren: " << endl;
        for (i=l.begin(); i != l.end(); ++i) // Neue Liste durchgehen und ausgeben
                cout << *i << " ";

        cout << endl;
        return 0;
}
```

Aufgelistet stehen die wichtigsten Funktionen, die wir in einer Liste verwenden können:

Operation	**Funktion**
Anfangselement	begin
Endelement	end
Einfügen am Ende	push_back
Löschen am Ende	pop_back
Einfügen am Anfang	push_front
Löschen am Anfang	pop_front
Einfügen irgendwo	insert
Löschen irgendwo	erase
Liste leeren	clear
Sortieren	sort

Beispiel: Einfach verkette Liste durch einen Container

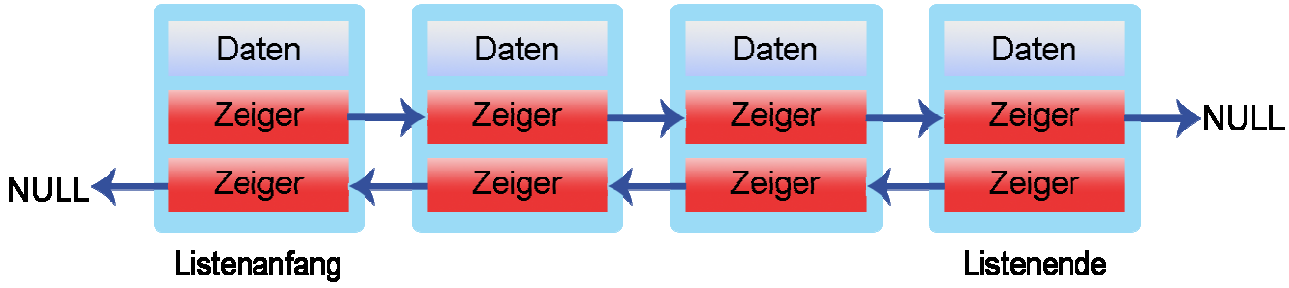
```
#include <iostream>
#include <list>
using namespace std;

void showlist(string, list<int>&);      // Deklaration der Methode showlist

int main()
{
   list<int> L; // Liste
   int x;
   cout << "Positive Zahl eingeben (Beenden mit 0):" << endl;
   do
   {
      cin >> x;
      if(x==0)
         break;
      L.push_back(x);                   // Am Ende der Liste einfügen
   }while (true);

   showlist("Anfangsliste:", L);        // Anfangsliste ausgeben

   L.push_front(123);                   // 123 am Anfang der Liste einfügen
   showlist("Nach dem Einfuegen von 123 am Anfang:", L);

   list<int>::iterator i = L.begin();   // Iterator, Zuweisung zum Anfang der Liste
   L.insert(++i, 456);                  // Iterator um 1 erhöhen, somit ist es das zweite Element
                                        // Einfügen an der zweiten Stelle

   showlist("Nach dem Einfuegen von 456 an zweiter Position:", L);

   i = L.end();                         // Iterator, Zuweisung zum Ende der Liste
   L.insert(--i, 999);                  // Iterator um 1 verringern, somit ist es das vorletzte Element
   showlist("Nach dem Einfuegen von 999 an vorletzter Position:", L);

   i = L.begin();                       // Iterator, Zuweisung zum Anfang der Liste
   x = *i;                              // x hat den Wert vom ersten Element
   L.pop_front();                       // Erstes Element der Liste löschen
   cout << "Am Anfang geloescht: " << x << endl;
   showlist("Nach dem Loeschen des ersten Elements:", L);

   i = L.end();                         // Iterator, Zuweisung zum Ende der Liste
   x = *--i;                            // x hat den Wert des letzten Elements (nur i wäre außerhalb der Liste)
   L.pop_back();                        // Löschen des letzten Elements
   cout << "Am Ende geloescht: " << x << endl;
   showlist("Nach dem Loeschen:", L);

   i = L.begin();                       // Iterator, Zuweisung zum Anfang der Liste
   x = *++i;                            // x hat den Wert des zweiten Elements
   cout << "Loesche: " << x << endl;
   L.erase(i);                          // Lösche an der Stelle vom Iterator
   showlist("Nach dem Loeschen des zweiten Elements:",L);

   L.clear();                           // Liste leeren
   showlist("Nach dem Loeschen der ganzen Liste:",L);

   return 0;
}

void showlist(string str, list<int> &L) // Definition der Methode showlist: Gibt mir den Inhalt der Liste aus
{
   list<int>::iterator i;
   cout << str << endl << " ";
   for (i=L.begin(); i != L.end(); ++i)
      cout << *i << " ";
   cout << endl;
}
```

5.3 Doppelt verkettete Liste, Bäume, Graphen

Komplexe Strukturen lassen sich mit einer einfachen Verkettung nicht aufbauen. Beispiel für komplexe Strukturen: Elemente eines Baumes, Äste, Zweige, Blätter, doppelt verkette Liste usw.

5.3.1 Doppelt verkette Liste

Die Elemente einer doppelt verketteten Liste enthalten nicht nur Verweise auf den Nachfolger, sondern auch auf das Vorgängerelement. Dadurch ist es möglich die Liste in beide Richtungen zu durchsuchen. Damit wird es auch möglich, vor oder hinter einem Element ein neues Element einzufügen. Der Nachteil dieser Liste besteht im zusätzlichen Speicherbedarf für einen Zeiger pro Element. Der Vorteil dagegen ist die große Flexibilität.



Beispiel:

```
#include <cstdlib>
#include <iostream>
#include <ctime>

using namespace std;

class Person
{
   friend class ListeVonPersonen; // friend Deklaration
   // Attribute
   private:
      string name;
      string vorname;
      int gebJahr;
      Person* Nachfolger; // Verkettungszeiger
      Person* Vorgaenger; // Zeiger auf den Vorgängen

   // Methoden
   public:
      Person (string n, string v, int j): name(n), vorname(v), gebJahr(j) {}

      string getName () const { return name; }
      string getVorname () const { return vorname; }
      int getGebJahr() const { return gebJahr; }
      Person* getNachfolger () { return Nachfolger; }

      void berechneAlter () {
         // Aktuelles Jahr ermitteln
         time_t zeitstempel;
         tm *datum;
         zeitstempel = time(0);
         datum = localtime(&zeitstempel);
```

```
        int aktJahr = datum->tm_year+1900;

        cout << "Alter: " << aktJahr - gebJahr << " Jahre" << endl;
        }
    Person* next () { return Nachfolger; }

    Person* vor() { return Vorgaenger; }

    void datenausgeben() {
        cout << "Name: " << name << ", Vorname: " << vorname << ", Geburtsjahr: " << gebJahr << endl;
    }
};

class ListeVonPersonen
{
  private:
    Person* Kopfzeiger;   // Zeichnet Listenanfang -- beinhaltet die Adresse des ersten Elements
    Person* Endezeiger;   // Zeichnet Listenende -- beinhaltet die Adresse des letzten Elements
  public:
  // Konstruktor für leere Liste
    ListeVonPersonen (): Kopfzeiger(0), Endezeiger(0) {}

    // Einfügen einer Person an den Anfang der Liste
    void insertAnfangListe (Person * kp)
    {
        // Kopfzeiger wird dem Nachfolger zugewiesen
        kp->Vorgaenger = 0;
        kp->Nachfolger = Kopfzeiger;

        if(Kopfzeiger != 0)
        {
            (kp->Nachfolger)->Vorgaenger = kp;
        }
        else {
            Endezeiger = kp;
        }
    Kopfzeiger = kp;
    }
    // Einfügen einer Person an das Ende der Liste
    void insertEndeListe (Person *kp)
    {
        // Wenn Kopfzeiger 0 ist, so wird die Adresse des übergebenen Objekts dem Kopfzeiger zugewiesen. Das
passiert nur einmal wenn die Liste leer ist
        if(Kopfzeiger == 0)
            Kopfzeiger = kp;

        // Wenn Endezeiger nicht 0 ist, so wird der Nachfolger des vorherigen Objekts aktualisiert. Dieses wird
ausgeführt, wenn die Liste nicht leer ist
        if(Endezeiger != 0)
            Endezeiger->Nachfolger = kp;

        // Ende der Liste festlegen, d.h. Nachfolger 0 zuweisen
        kp->Nachfolger = 0;

        // Vorgänger bekommt die Adresse des Endzeigers zugewiesen. Endezeiger beinhaltet die Adresse des vorher
angelegten Objekts.
        kp->Vorgaenger = Endezeiger;

        // Endezeiger bekommt die Adresse des neu eingefügten Objekts zugewiesen
        Endezeiger = kp;
    }
    // Entfernen einer Person vom Anfang der Liste
    Person * remove ()
    {
        if(Kopfzeiger != 0)
            Kopfzeiger = Kopfzeiger->Nachfolger;

        return Kopfzeiger;
```

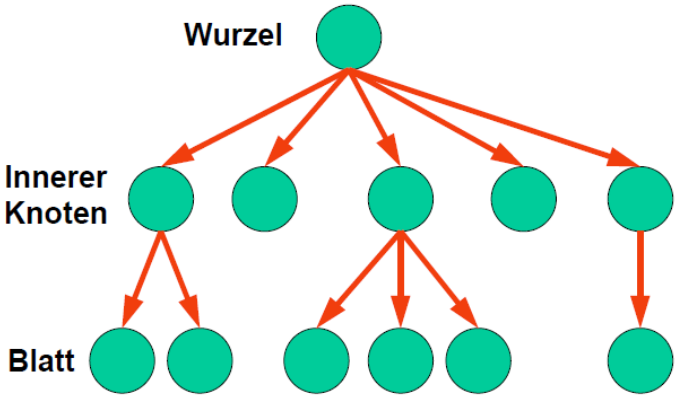
```
        }

    // Zugriff auf erste Person in der Liste
    Person * begin () { return Kopfzeiger; }

    // Zugriff auf letzte Person in der Liste
    Person * end () { return Endezeiger; }

    // Ist die Liste leer
    bool empty () const { return Kopfzeiger == 0; }
};

int main ()
{
    ListeVonPersonen list; // Leere Liste erzeugen
    char ant = 'j';
    string name="", vorname="";
    int gebJahr=0;

    Person *kp;

    // Füllen der Liste
    while (ant != 'n')
    {
        cout << "Name: ";
        cin >> name;
        cout << endl << "Vorname: ";
        cin >> vorname;
        cout << endl << "Geburstsjahr: ";
        cin >> gebJahr;
        kp = new Person(name, vorname, gebJahr);
        // Element werden an das Ende der Liste eingefügt
        list.insertEndeListe (kp);
        // oder Elemente werden an den Anfang der Liste eingefügt
        //list.insertAnfangListe(kp);
        cout << "Weiter (j/n)" << endl;
        cin >> ant;
    }

    // Ausgabe der Liste von Anfang
    for (kp=list.begin(); kp != 0; kp = kp->next())
        kp->datenausgeben();

    // Ausgabe der Liste von Ende
    for (kp = list.end(); kp!=0; kp = kp->vor())
        kp->datenausgeben();

    // Leeren der Liste
    while(! list.empty())
        list.remove();

    cout << "Nach Leeren der Liste" << endl;

    // Ausgabe der Liste
    for (kp=list.begin(); kp != 0; kp = kp->next())
        kp->datenausgeben();

    system("pause");
    return 0;
}
```

Das Beispiel zeigt eine von vielen Möglichkeiten, was mit doppelt verketten Listen möglich ist. Im Beispiel sehen Sie wie Elemente am Anfang oder am Ende der Liste eingefügt oder vom Anfang oder vom Ende der Liste ausgegeben werden können.

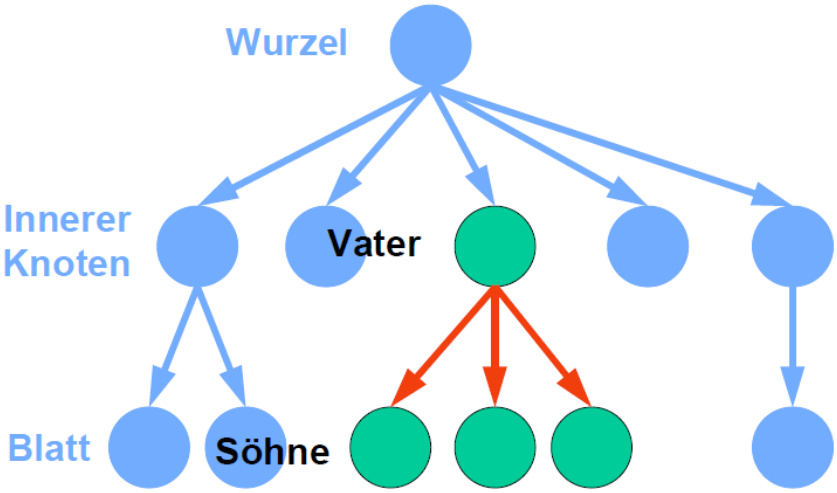
5.3.2 Bäume

Lassen wir eine beliebige Anzahl von Nachfolgern zu, ergibt sich ein sogenannter Baum:

- Die Elemente eines Baumes nennt man Knoten
- Die Nachfolger eines Knotens nennt man Söhne
- Ausgehend vom Sohn gilt stets:
  - Jeder Sohn-Knoten hat genau einen Vater
  - Außer einem Knoten, der keinen Vater hat und Wurzel genannt wird
- Knoten, die keine Söhne mehr haben, nennt man auch Blätter



Ein Baum
- ist leer oder
- besteht aus einem Knoten, der seinerseits wieder auf beliebig viele Bäume verweisen kann.



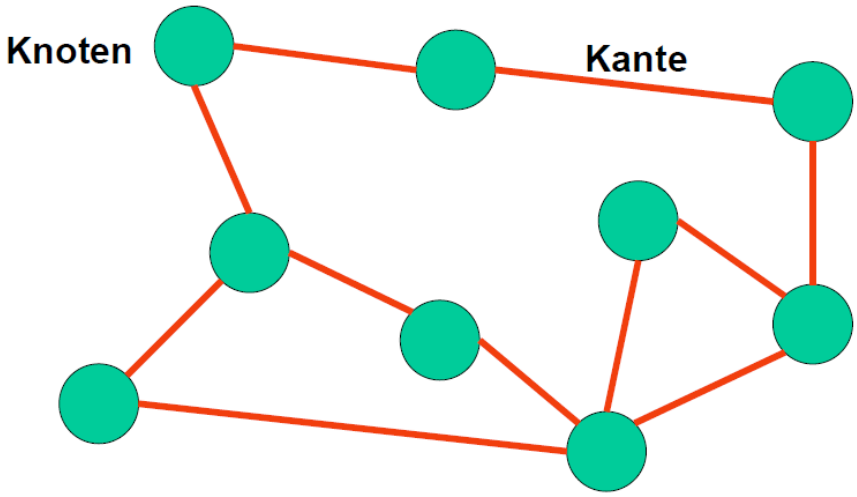
Derartige Definitionen nennt man rekursiv. Erinnern Sie sich an die Definition unserer Listenelemente. Auch das war eine rekursive Definition, da das einzelne Element einen Verweis auf gleichartige Objekte beinhaltet. In engen Zusammenhang damit stehen rekursive Algorithmen.

Für welche Daten kann man nun diese Baumstruktur einsetzen?
- Das Dateisystem ihres Computers bildet etwa einen solchen Baum
- Die Knoten sind die Daten oder Unterverzeichnisse
- Blätter werden stets von Dateien gebildet
- Die Wurzel ist das Laufwerk z.B. C

5.3.3 Graphen

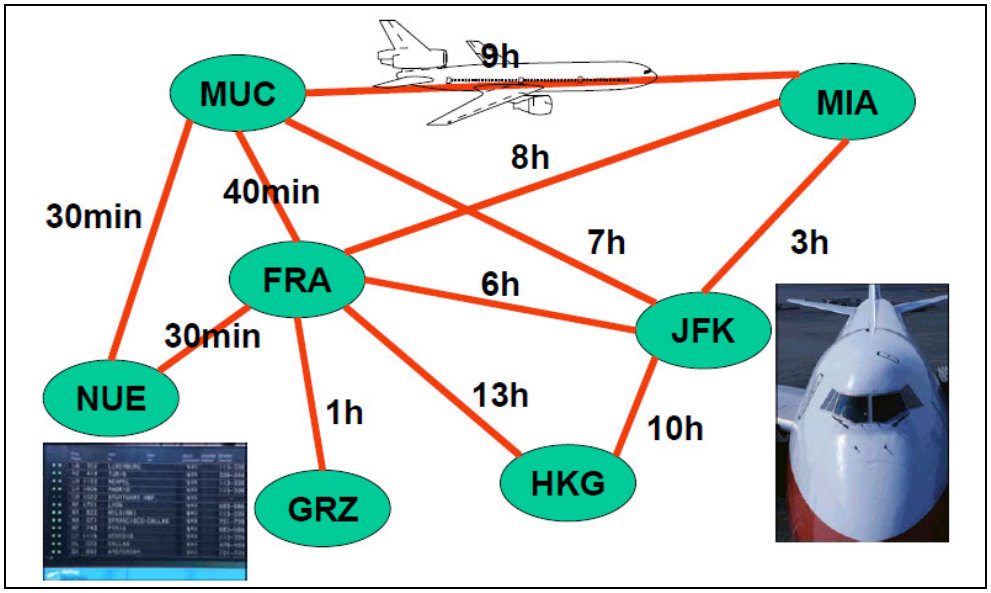
Gibt es noch weitere Datenstrukturen, die sich mit Hilfe von Zeigern bilden lassen?

Die wohl allgemeinste Form derartiger Datenstrukturen nennt man Graphen.
Ein Graph ist eine Kollektion von Knoten und Kanten. Knoten sind einfache Objekte. Sie haben einen Namen und können Träger von Werten oder Eigenschaften sein. Kanten sind Verbindungen zwischen diesen Knoten.



Ein Beispiel wären die Direktflugverbindungen in Deutschland. Dabei bilden die Flughäfen in Deutschland die Menge der Knoten. Eine Kante zwischen zwei Flughäfen existiert nur, wenn es eine direkte Flugverbindung gibt. Zusätzlich können wir hier bei den Kanten die Flugzeit angeben.

Mit diesem Graphen können wir z.B. folgende interessante Fragestellung lösen: Finde die günstigste Verbindung zwischen zwei Flughäfen, zwischen denen keine Direktverbindung besteht.



In ihren eigenen C++ Programmen können sie nun dynamische Datenstrukturen einsetzen, wenn Sie eine unbekannte Anzahl von Objekten verwalten müssen. Sie haben verschiedene derartige Strukturen kennengelernt:

- einfach verkettete Listen
- doppelt verkettete Listen
- Bäume und Graphen

6 Klassen

In diesem Kapitel werden wir uns mit dem Thema der Vererbung, Mehrfachvererbung, sowie der mit der Vererbung zusammenhängenden Begriffen wie Polymorphismus und abstrakte Klasse befassen.

6.1 Vererbung

Im Teil 1 des Kurses haben wir bereits das Thema Vererbung sehr ausführlich besprochen. Deswegen wird in diesem Kapitel nicht auf die Einzelheiten der Vererbung eingegangen.

Die Vererbung ist ein sehr wichtiges oder besser gesagt zentrales Konzept der objektorientierten Programmierung. Vererbung dient der Unterstützung der Wiederverwendbarkeit von Programmcode. Die vererbende Klasse heißt Basisklasse oder Oberklasse, die erbende Klasse heißt abgeleitete Klasse oder Unterklasse. Diese Begriffe werden jedoch in der Literatur nicht einheitlich gebraucht.

Eine Klasse kann Eigenschaften (Attribute, Elemente) und das Verhalten (Methoden) von einer oder mehreren anderen Klassen erben. Das bedeutet diese übernehmen oder auch ändern. Diese Klasse wird Unterklasse oder abgeleitete Klassen genannt. Die abgeleitete Klasse ist eine Spezialisierung der Basisklasse. Bei der Klassifikation ist es sehr ratsam nach den Ähnlichkeiten und Unterschieden zu suchen. Wenn eine Basisklasse bekannt bzw. deklariert ist, so brauchen Sie in einer zugehörigen abgeleiteten Klasse nur die Unterschiede bzw. Abweichungen beschreiben. Alles andere kann von der Basisklasse verwendet werden.

```
#include <cstdlib>
#include <iostream>

using namespace std;

class Person {
    protected:
        string name;
        string vorname;
        int gebJahr;
    public:         //.......
};

class Student : public Person { // Student erbt von Person
    private:
        string matrikelnr;
    public:         //......
};

class Angestellte : public Person { // Angestellte erbt von Person
    private:
        long gehalt;
    public:         //........
};

int main()
{
   Person p;
   Student s;
   Angestellte a;
   system("pause");
   return 0;
}
```

Die Vererbung beschreibt eine ist-ein-Beziehung. Auf das Beispiel bezogen: Student *ist eine* Person und Angestellte *ist eine* Person. Die Vererbung ist eine gerichtete Beziehung. Die Umkehrung gilt im Allgemeinen nicht, d.h. eine Person ist nicht unbedingt ein Student. Student und Angestellte hat alle Eigenschaften einer Person und kann noch um weitere Eigenschaften erweitert werden. An dieser Stelle muss noch erwähnt werden, dass die **Konstruktoren** und **Destruktoren nicht vererbt werden**.

6.2 Mehrfachvererbung

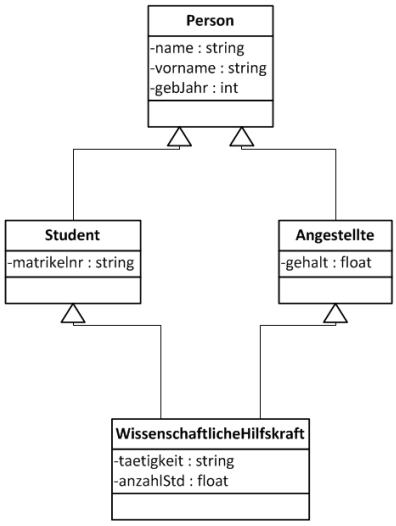
Die Mehrfachvererbung bietet dem Programmierer eine große Flexibilität und mehr Möglichkeiten die Objekte der realen Welt abzubilden als die Einfachvererbung. Die Mehrfachvererbung wird jedoch nicht so oft verwendet. Dieses hängt natürlich von der Art der Problemstellung ab.

Mehrfachvererbung bedeutet, dass eine Klasse von mehreren Basisklassen erben kann. Dieses wird im Programmcode wie folgt deklariert:

```
class WissenschaftlicheHilfskraft : public Student, public Angestellte
{
private:
        string taetigkeit;
         float anzahlStd;
public:
//.....
};
```

Die Klasse WissenschaftlicheHilfskraft erbt alle Eigenschaften und Methoden der Klassen Student und Angestellte, welche protected sind.

Anhand vom Klassendiagramm können Sie die Beziehungen zwischen den Klassen nachvollziehen.



Die Mehrfachvererbung wird in der Programmierung nicht so oft verwendet, weil die Programme dadurch unübersichtlich und fehleranfällig sein können. Das kann auch als Grund genannt werden, warum viele Programmiersprachen die Mehrfachvererbung gar nicht zulassen.

6.3 Polymorphismus (Vielgestaltigkeit)

Der weitere grundlegende Aspekt der OOP ist der Polymorphismus (Vielgestaltigkeit). Das Prinzip des Polymorphismus besagt nun, dass zur Zeit der Compilierung noch nicht feststehen muss, welche Methode/Funktion zur Laufzeit ausgeführt werden soll. Der Compiler muss Maschinencode erzeugen, bei dem erst zur Laufzeit die Zuordnung zu einer bestimmten Funktion (Methode) stattfindet. Dies bezeichnet man auch als *späte* oder ***dynamische Bindung*** - im Gegensatz zur herkömmlichen ***statischen Bindung***. Diese beiden Formen werden in diesem Kapitel präsentiert.

Wie wir bereits wissen erbt die abgeleitete Klasse die Methoden der Basisklasse. Wenn jedoch eine Methode der Basisklasse nicht die gewünschte Funktionalität für die abgeleitete Klasse liefert, so kann diese **überschrieben** werden.

Eine überschriebene Methode ist im Sinne der Polymorphismus:

- eine Methode, die in einer Basisklasse definiert wurde
- mit derselben Signatur und demselben Rückgabewert in einer abgeleiteten Klasse definiert wurde

Beispiel:

```
#include <cstdlib>
#include <iostream>

using namespace std;

class Person {
    protected:
        string name, vorname;
    public:
        Person(string n, string v): name(n), vorname(v) {}
        void datenausgeben() {
            cout << "Name: " << name << " Vorname: " << vorname << endl;
        }
};

class Student : public Person {
    private:
        string matrikelnr;
    public:
        Student(string n, string v, string nr): Person(n, v), matrikelnr(nr) {}
        void datenausgeben() {
            cout << "Student: " << name << " " << vorname << endl;
            cout << "Matrikelnummer: " << matrikelnr << endl;
        }
};

int main()
{
   Person p("Karl", "Heinz");
   Student s("Huber", "Hans", "225566");
   p.datenausgeben();
   s.datenausgeben();
   system("pause");
   return 0;
}
```

Was gibt dieses Programm aus?

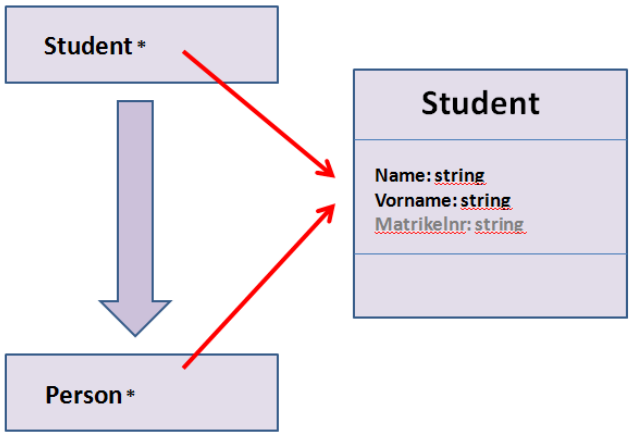
```
Name: Karl Vorname: Heinz
Student: Huber Hans
Matrikelnummer: 225566
```

Für ein Objekt vom Typ Person und vom Typ Student wird die eigene Methode datenausgeben() der jeweiligen Klasse aufgerufen. Der Typ eines Objekts entscheidet also, welche Methode aufzurufen ist. Das gilt auch für Zeigervariablen, denn auch sie haben einen Typ, nämlich den Typ des Objekts, auf das sie verweisen. Wir bezeichnen diesen Vorgang auch als **statisches Binden**. Wenn es erst während der Programmausführung entschieden wird, welche Methode aufgerufen werden soll, so nennt man diesen Vorgang **dynamisches** oder **spätes Binden**. Eine dann zur Laufzeit ausgewählte Methode heißt *virtuelle Methode* bzw. *Funktion*. Auf diese wird jetzt im folgenden eingegangen.

Virtuelle Funktionen

Was würde passieren, wenn wir den Typ eines Zeigers in einen anderen Typ umwandeln wollen? Normalerweise kann der Typ eines Objekts nicht einfach geändert werden. Wenn wir mit abgeleiteten Klassen arbeiten, ist unter bestimmten Umständen eine Typkonvertierung möglich. Und zwar kann eine spezielle Klasse (abgeleitete Klasse) auf die generelle Klasse (Basisklasse) konvertiert werden.

Implizite Typkonvertierung
Beachte: Ein Objekt, ein Zeiger oder eine Referenz einer abgeleiteten Klasse kann einem Objekt, einem Zeiger oder einer Referenz seiner Basisklasse zugewiesen werden. Dabei findet eine implizite Typkonvertierung statt (**gilt nicht umgekehrt**). Bei Objekten gehen die zusätzlichen Daten der abgeleiteten Klasse verloren, der Basisklassenanteil wird kopiert.



In folgendem Beispiel legen wir ein Objekt vom Typ Student an und weisen den Zeiger auf dieses Objekt einer Variablen vom Typ Zeiger auf Person zu. Also eigentlich zwei unterschiedliche Klassen, aber diese Umwandlung ist möglich, weil Student „ist-eine“ Person. Deshalb darf auch ein Person-Zeiger auf einen Studenten verweisen. Wir haben also gar keine Konvertierung des Objekts selbst vorgenommen, sondern lediglich einen andersartigen Verweis auf das Objekt Student.

```
int main()
{
   // Instanz von Student anlegen
   Student *s = new Student("Huber", "Hans", "225566");
   // implizite Typkonvertierung: Student zu Person
   Person *p = s;
   p->datenausgeben();
   system("pause");
   return 0;
}
```

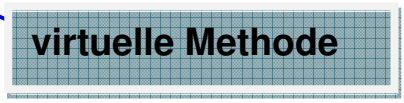
Was gibt dieses Programm aus?

Name: Huber Vorname: Hans

Das ist eigentlich nicht überraschend. Die Regel, dass der Datentyp des Objekts die zugehörige Methode bestimmt gilt auch hier. Also ein Zeiger auf Person → Methode datenausgeben() von Person. Dieses Verhalten ist aber nicht immer erwünscht. Wir hatten eigentlich in diesem Beispiel ein Objekt vom Typ Student angelegt, es wird aber die Methode der Klasse Person aufgerufen. Dieses Verhalten können wir grundlegend ändern, wenn wir unsere datenausgeben() Methode **virtuell** deklarieren.

Ziel von virtuellen Methoden, ist die Verwendung „allgemeiner" Zeiger zur Verwaltung unterschiedlicher Objekte einer Klassenhierarchie. Wobei die (in den abgeleiteten Klassen gleichnamigen) Methoden des Objekts angesprochen werden, auf das gerade gezeigt wird (nicht der Zeigertyp bestimmt die Methode, sondern der Objekttyp).
Um dies zu gewährleisten müssen die relevanten (nicht statischen) Methoden zumindest in der Basisklasse mit dem Schlüsselwort virtual als **dynamisch bindbar** gekennzeichnet werden.

```
class Person {
    protected:
        string name, vorname;
    public:
        Person(string n, string v): name(n), vorname(v) {}
        virtual void datenausgeben() {
            cout << "Name: " << name << " Vorname: " << vorname << endl;
        }
};

class Student : public Person {
    private:
        string matrikelnr;
    public:
        Student(string n, string v, string nr): Person(n, v), matrikelnr(nr) {}
        void datenausgeben() {
            cout << "Student: " << name << " " << vorname << endl;
            cout << "Matrikelnummer: " << matrikelnr << endl;
        }
};
```




```
int main()
{
   // Instanz von Student anlegen
   Student *s = new Student("Huber", "Hans", "225566");
   // implizite Typkonvertierung: Student zu Person
   Person *p = s;
   p->datenausgeben();
   system("pause");
   return 0;
}
```

Die einzige Veränderung an unserer Klasse Person ist das Schlüsselwort virtual vor der Deklaration der Methode datenausgeben().

Was gibt dieses Programm aus?

Student: Huber Hans
Matrikelnummer: 225566

Die Ausgabe ist jetzt etwas vollkommen Neues. Der Typ der Zeigervariablen bestimmt nun nicht mehr die aufzurufende Methode, sondern der tatsächlich vorliegende Objekttyp entscheidet, welche Methode aufgerufen wird. Denn obwohl p ein Zeiger auf Person ist, wird die Methode datenausgeben() der Klasse Student aufgerufen. Die richtige Methode wird während der Laufzeit des Programms gewählt. Diesen Vorgang nennt man **dynamisches** oder **spätes Binden**.

Bei den virtuellen Methoden können wir also nicht vorhersagen, welche Methode tatsächlich ausgeführt wird. Eine Klasse mit virtuellen Methoden wird polymorphe Klasse genannt. Um dynamisches bzw. polymorphes Verhalten in C++ zu erzielen, müssen die aufgerufenen Methoden virtuell deklariert sein.

Hier noch einige Hinweise zur Verwendung von virtual:

- *virtual* muss in der Basisklasse angegeben werden und sollte in den davon abgeleiteten Klassen wiederholt werden (z.B. für Mehrfachvererbung, virtual vererbt sich)
- eine Klasse, in der eine virtuelle Methode vorhanden ist (durch Deklaration oder Vererbung), heißt auch polymorphe Klasse
- *virtual* wirkt nur, wenn die so gekennzeichnete Methode in der abgeleiteten Klasse exakt die gleiche Schnittstelle (Methodenname + Rückgabetyp + Parameterliste) besitzt. Ausnahme: Ist der Rückgabetyp eine Referenz oder ein Zeiger auf die Basisklasse, dann darf der Rückgabetyp in der abgeleiteten Klasse eine Referenz oder ein Zeiger auf die abgeleitete Klasse sein.

Beachten Sie, dass eine als virtual deklarierte Methode in einer Basisklasse explizit eine Schnittstelle für alle davon abgeleiteten Klassen definiert. Nicht virtuelle Methoden sollten in abgeleiteten Klassen nicht überschrieben werden.

6.4 Abstrakte Klasse

In vielen Fällen sollte die Basisklasse sehr allgemein sein und nur Code enthalten, der nicht geändert werden soll oder muss. Es ist auch nicht notwendig oder sogar gewünscht, dass Instanzen dieser Klassen angelegt werden. Diese Klassen nennt man abstrakte Klassen. Diese sollen ausschließlich als Ober- oder Basisklasse dienen. Objekte bzw. Instanzen werden nur von den abgeleiteten Klassen erzeugt. Von der abstrakten Klasse **können keine** Objekte angelegt werden. Um eine Klasse abstrakt zu machen, müssen die Methoden rein virtuell deklariert werden. Abstrakte Klassen haben **mindestens eine** rein virtuelle Methode, die typischerweise keinen Definitionsteil hat. Die rein virtuellen Methoden gewährleisten, dass immer die zu dem Objekttyp passende Methode aufgerufen wird. Kurz zusammengefasst, definiert die abstrakte Klasse ein gemeinsames Protokoll oder Muster oder eine Art Designvorschrift für alle abgeleiteten Klassen. Eine rein virtuelle Methode wird durch die Ergänzung von „=0“ deklariert. Deklarationsteil in der Basisklasse für die Methode datenausgeben() fällt weg. Rest des Beispielprogramms bleibt jedoch unverändert.

Beispiel:

```
#include <cstdlib>
#include <iostream>

using namespace std;

class Person {
    protected:
        string name, vorname;
    public:
        Person(string n, string v): name(n), vorname(v) {}
        virtual void datenausgeben() = 0;
};

class Student : public Person {
    private:
        string matrikelnr;
    public:
        Student(string n, string v, string nr): Person(n, v), matrikelnr(nr) {}
        void datenausgeben() {
            cout << "Student: " << name << " " << vorname << endl;
            cout << "Matrikelnummer: " << matrikelnr << endl;
        }
};
int main()
{
   // Instanz von Student anlegen
   Student *s = new Student("Huber", "Hans", "225566");
   // implizite Typkonvertierung: Student zu Person
   Person *p = s;
   p->datenausgeben();
   system("pause");
   return 0;
}
```

Hinweise zu abstrakten Klassen:

- Von der Klasse Person kann **keine Instanz** gebildet werden. Sie wird deshalb als abstrakte Klasse bezeichnet.
- Zeiger und Referenzen auf eine abstrakte Klasse **können** jedoch definiert werden. So haben wir das auch in unserem Beispiel gemacht.
- Rein virtuelle Methoden werden vererbt. D.h. eine von einer abstrakten Klasse abgeleitete Klasse ist nur dann nicht mehr abstrakt, wenn sie für alle rein virtuelle Methoden eine Implementierung angibt.
- Eine abgeleitete Klasse kann erst durch eine eigene rein virtuelle Methode zu einer abstrakten Klasse werden.

7 Überladen von Operatoren

Die Verwendung der arithmetischen Operatoren wie +, -, * oder / ist uns allen geläufig. Darüber hinaus kennt C++ noch eine ganze Menge weiterer Operatoren. Die Überladung von Operatoren ermöglicht uns vorhandene Operatoren auch auf Klassenobjekte anzuwenden. Wir können z.B. festlegen, welche Bedeutung der Operator „–“ für Objekte einer bestimmten Klasse haben soll.
In diesem Kapitel werden die verschiedenen Möglichkeiten der Operator-Überladung beschrieben. Wir werden neben den arithmetischen Operatoren und den Vergleichen auch die Shift-Operatoren für die Ein-/Ausgabe überladen. Da für die Operator-Überladung auch das Konzept der friend-Funktionen von Bedeutung ist, wird dieser in diesem Zusammenhang vorgestellt.

7.1 Übersicht aller Operatoren

Ein Operator ist überladen, wenn es unterschiedliche Datentypen gibt, für die der Operator definiert ist. Einen Operator zu überladen bedeutet nichts anderes als dem Operator für einen neuen Datentyp eine eigene Bedeutung zu geben. Für die elementaren Datentypen, wie z.B. int, float sind die meisten Operatoren bereits überladen.

Beispiel:

```
int a = 10, b = 7;
cout << a / b;
```

Im Beispiel entscheidet der Typ der Operanden, welchen Maschinencode der Compiler für die Division erzeugt. Wenn beide Operanden einen ganzzahligen Typ haben, wie im Beispiel, so wird die Ganzzahl-Division ausgeführt, ansonsten die Gleitpunktdivision. Also werden je nach Typ der Operanden verschiedene Aktionen ausgeführt.

C++ kennt eine große Menge an Operatoren. Wir können mit Hilfe der Operatoren beliebig komplexe Ausdrücke in unserem Programm aufbauen. Wir können sogar viele Operationen in einer Programmzeile schreiben. Dieses erhöht aber nicht gerade die Lesbarkeit eines Programms. Zum Verständnis eines komplexen Ausdrucks sind folgenden Eigenschaften eines Operators hilfreich:

- Die Stelle eines Operators gibt die Anzahl, der an der Operation beteiligten Operanden an. Es gibt:
  - unäre Operatoren: - (als Vorzeichen), ++ (Inkrement) , -- (Dekrement), ….
  - binäre Operatoren: +, -, *, /, &&, ||, ….
  - einen tertiären Operator: ?:
- Die Priorität eines Operators legt in Ausdrücken mit verschiedenen Operatoren die Reihenfolge fest, in der die einzelnen Operationen ausgeführt werden.
- Bei mehreren Operatoren gleicher Priorität muss man wissen, ob der Ausdruck von links nach rechts oder rechts nach links ausgewertet wird (Assoziativität).

7.2 Überladbare und nicht überladbare Operatoren

Operatoren	Bedeutung
Überladbare Operatoren	
+, -, *, /, %, ++, --	arithmetische Operatoren
==, !=, <, <=, >, >=	Vergleichsoperatoren
&&, ||, !	Logische Operatoren (and, or, not)
=	Zuweisungsoperator
&, |, ^, ~, <<, >>	Bitoperatoren
(), []	Funktionsaufruf, Indexoperator
&, *, ->	sonstige Operatoren
Nicht überladbare Operatoren	
. :: .*	Zugriffsoperatoren
?:	Auswahloperator

Einschränkungen:

- Die Priorität und die Ausführungsreihenfolge der Operatoren kann nicht geändert werden.
- Binäre Operatoren bleiben binär und unäre Operatoren bleiben unär.
- Neue Operatorsymbole können nicht definiert werden.
- Ein Operand muss immer ein Klassentyp sein, d.h. man kann die Operatoren für die Standardtypen nicht „umdefinieren“.
- Operatorfunktionen dürfen keine default-Parameter haben.

7.3 Motivation zur Operatorüberladung

Die Idee der Operatorüberladung ist die Verwendung der einheitlichen Notation für das Verknüpfen bzw. Manipulieren nicht nur für Standarddatentypen, sondern auch für Klassenobjekte.

```
int a = 10, b = 23, c = 0;
c = a + b;

Zahlen a(1, 4), b(2, 9), c;
c = a + b;
```

Vorteil der Operatorüberladung: Die Programme werden übersichtlicher und verständlicher.

7.4 Syntax der Operatorüberladung

Operatorfunktion wird definiert, um einen entsprechenden Operator zu überladen. Diese Funktion legt die Aktionen fest, die der Operator ausführen soll. Der Operatorfunktionsname beginnt immer dem Schlüsselwort *operator* und Operatorsymbol (op steht als Platzhalten für den zu überladenden Operator z.b. +):

Rückgabetyp **operator op**(Parameterliste);

Eine Operatorfunktion kann entweder als externe (globale) Funktion oder als Memberfunktion d.h. Methode einer Klasse definiert werden. Grundsätzlich hat der Programmierer die Wahl zwischen diesen beiden Möglichkeiten.

7.4.1 Operatorfunktion als externe Funktion

Ist eine Funktion keine Methode der Klasse, so werden alle Operanden als Parameter übergeben.

Notation im Programm	alternative Notation im Programm	Syntax bei der Deklaration als **externe Funktion**
a **op** *b*	**operator op** (*a, b*)	RTyp **operator op** (Typ *par1*, Typ *par2*);
op *a*	**operator op** (*a*)	RTyp **operator op** (Typ *par1*);
a **op**	**operator op** (*a, 0*)	RTyp **operator op** (Typ *par1*, int *par2*);

Zu beachten ist, dass die externe (globale) Funktion keinen Zugriff auf die privaten Elemente der Klasse hat. Um einer externen Funktion doch den Zugriff auf die privaten Elemente zu geben, kann diese als „Freund“ der Klasse definiert werden oder die Klasse um die sogenannten „getter“ Methoden zu erweitern.

Beispiel:

```
#include <cstdlib>
#include <iostream>

using namespace std;

class Zahlen {
     private:
          float zahl1, zahl2;
     public:
          Zahlen(): zahl1(0), zahl2(0) {}
          Zahlen(float z1, float z2): zahl1(z1), zahl2(z2) {}
          //Die Methode operator== ist "Freund" der Klasse
          friend bool operator==(Zahlen, Zahlen);
};

   // Operator == wird überladen
   bool operator== (Zahlen a, Zahlen b) {
   // Attribute beider Objekte vergleichen – Rückgabe entweder true oder false
    return (a.zahl1 == b.zahl1 && a.zahl2 == b.zahl2);
}

int main()
{
   Zahlen x(3,5), y(4,6);
   // Operator == soll als externe Funktion überladen werden, dabei werden die Zahlen zahl1 und zahl2 miteinander
verglichen
   if (x == y) // gleichbedeutend mit      operator==(x, y)
     cout << "Gleich!" << endl;
   else
      cout << "Unterschiedlich" << endl;

   system("pause");
   return 0;
}
```

7.4.2 Operatorfunktion als Funktion einer Klasse

Ist die Funktion eine Methode der Klasse, so ist der linke Operand immer das aktuelle Objekt (*this*-Zeiger).

Notation im Programm	alternative Notation im Programm	Syntax bei der Deklaration als **Memberfunktion**
a **op** *b*	*a*.**operator op** (*b*)	RTyp **operator op** (Typ *par*);
op *a*	**operator op** ()	RTyp **operator op** ();
a **op**	**operator op** (*0*)	RTyp **operator op** (int *par*);

Ist der linke Operand, d.h. der 1. Parameter kein Klassenobjekt so kann die Überladung nicht als Memberfunktion deklariert werden. Sie muss, falls trotzdem auf private-Komponenten zugegriffen werden soll, als friend-Funktion deklariert werden.

Beispiel: operator== als Memberfunktion

```
#include <cstdlib>
#include <iostream>

using namespace std;

class Zahlen {
     private:
           float zahl1, zahl2;
     public:
           Zahlen(): zahl1(0), zahl2(0) {}
           Zahlen(float z1, float z2): zahl1(z1), zahl2(z2) {}

           bool operator== (Zahlen a) {
               return (this->zahl1 == a.zahl1 && this->zahl2 == a.zahl2);
           }
};

int main()
{
   Zahlen x(3,5), y(4,6);
   // Operator == soll als Memberfunktion überladen werden, dabei werden die Zahlen zahl1 und zahl2 miteinander
verglichen
   if (x == y) // gleichbedeutend mit     x.operator==(y)
      cout << "Gleich!" << endl;
   else
       cout << "Unterschiedlich" << endl;

   system("pause");
   return 0;
}
```

Folgende Operatoren müssen als Methoden der Klasse deklariert sein:

Operator	Bedeutung
=	Zuweisung
[ ]	Indexoperator
( )	Funktionsaufruf
->	Elementzugriff für Zeiger

7.4.3 Shift-Operatoren für die Ein- und Ausgabe

Soll das Objekt der Klasse Zahlen auf dem Bildschirm ausgeben, so führt die Ausgabeanweisung:

```
Zahlen a(8,7);
cout << a;
```

zu einer Fehlermeldung des Compilers. Der stream cout schickt nur dann ein Objekt zur Standardausgabe, wenn für diesen Typ auch eine Ausgabefunktion existiert. Für unsere Klasse Zahlen ist das jedoch nicht der Fall. Der Compiler kann jedoch diese Anweisung übersetzen, wenn er eine passende Operatorfunktion operator<< () findet.

7.4.3.1 Operator<< überladen

In obiger Anweisung ist der linke Operand von << das Objekt cout, das der Klasse ostream angehört. Die Standardklasse ostream kann nicht verändert werden. Deshalb ist es notwendig die Operatorfunktion global mit zwei Parametern zu definieren. Der rechte Operand ist ein Objekt der Klasse Zahlen. Somit ergibt sich folgende Operatorfunktion:

```
ostream& operator<< (ostream& os, const Zahlen& a);
```

Der Returnwert der Funktion ist eine Referenz auf ostream. Dadurch ist es möglich, den Operator wie gewohnt zu verketten.

```
cout << a << endl;
```

7.4.3.2 Operator>> überladen

Wie wir bereits wissen, dient der Operator >> der Eingabe. Dieser soll aber für unsere Zwecke überladen werden, sodass folgende Anweisung möglich wird:

```
cout << „Geben Sie die Zahlen ein:“;
cin >> a;
```

Diese Anweisung führt zum Aufruf von:

```
operator>> (cin, a);
```

Der erste Parameter der Funktion wird als Referenz auf istream deklariert, da cin ein Objekt der Standardklasse istream ist. Der zweite Parameter ist eine Referenz auf Zahlen.

Sollen die externen Funktionen Zugriff auf private Elemente der Klasse benötigen, müssen diese Funktionen als friend der Klasse deklariert werden.

Beispiel:

```
#include <cstdlib>
#include <iostream>

using namespace std;

class Zahlen {
    private:
        float zahl1, zahl2;
    public:
        Zahlen(): zahl1(0), zahl2(0) {}
        Zahlen(float z1, float z2): zahl1(z1), zahl2(z2) {}

        // Methoden für die Ein-/Ausgabe als Freund der Klasse deklarieren
        friend istream& operator>> (istream& is, Zahlen& a);
        friend ostream& operator<< (ostream& os, const Zahlen& b);
};

// Eingabe
istream& operator>> (istream& is, Zahlen& a) {
      cout << endl << "Zahl1: ";
      is >> a.zahl1;
      cout << "Zahl2: ";
      is >> a.zahl2;
      return is;
}

// Ausgabe
ostream& operator<< (ostream& os, const Zahlen& a) {
      os << "Zahl1; " << a.zahl1 << " Zahl2: " << a.zahl2 << endl;
      return os;
}

int main()
{
   Zahlen x;
   cin >> x;
   cout << x;

   system("pause");
   return 0;
}
```

7.5 Beispiele

7.5.1 Arithmetische Operatoren

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

class Zahlen {
     private:
          float zahl1, zahl2;
     public:
          Zahlen(): zahl1(0), zahl2(0) {}
          Zahlen(float z1, float z2): zahl1(z1), zahl2(z2) {}

          //Memebermethoden
          const Zahlen operator += (const Zahlen& a);
          const void operator -= (const Zahlen& a);

          friend ostream& operator<< (ostream& os, const Zahlen& b);
          friend Zahlen operator+ (const Zahlen& links, const Zahlen& rechts);
          friend void operator- (Zahlen& links, const Zahlen& rechts);
};

const Zahlen Zahlen::operator += (const Zahlen& a) {
     zahl1 += a.zahl1;
     zahl2 += a.zahl2;
     return *this;
}

const void Zahlen::operator -= (const Zahlen& a) {
     zahl1 -= a.zahl1;
     zahl2 -= a.zahl2;
}

// Ausgabe
ostream& operator<< (ostream& os, const Zahlen& a) {
       os << "Zahl1; " << a.zahl1 << " Zahl2: " << a.zahl2 << endl;
       return os;
}

// externe Funktion
Zahlen operator+ (const Zahlen& links, const Zahlen& rechts) {
      Zahlen z;
      z.zahl1 = links.zahl1 + rechts.zahl1;
      z.zahl2 = links.zahl2 + rechts.zahl2;
      return z;
}

// Beachte links darf nicht als const deklariert sein!
void operator- (Zahlen& links, const Zahlen& rechts) {
     links.zahl1 -= rechts.zahl1;
     links.zahl2 -= rechts.zahl2;
}

int main()
{
   Zahlen x(5,6), y(9,18), z;
   x += y;
   x -= y;

   z = x + y;
   cout << z;
   x - y;
   cout << x;
   system("pause");
   return 0;
}
```

7.5.2 Vergleichsoperatoren

Die Vergleichsoperatoren liefern als Ergebnis einen boolschen Wert. Sie können Vergleichsoperatoren aber auch ohne Rückgabewert programmieren. Die Vergleichsoperatoren sollten als externe Funktionen definiert werden. Dieses ist aber nur eine Empfehlung.

```
#include <cstdlib>
#include <iostream>

using namespace std;

class Zahlen {
     private:
          float zahl1, zahl2;
     public:
          Zahlen(): zahl1(0), zahl2(0) {}
          Zahlen(float z1, float z2): zahl1(z1), zahl2(z2) {}

          float getZahl1() { return zahl1; }
          float getZahl2() { return zahl2; }
};

bool operator == (Zahlen& a, Zahlen& b) {
    return (a.getZahl1() == b.getZahl1() && a.getZahl2() == b.getZahl2());
}


bool operator != (Zahlen& a, Zahlen& b) {
    return !(a == b);
}


bool operator < (Zahlen& a, Zahlen& b) {
    return (a.getZahl1() < b.getZahl1() && a.getZahl2() < b.getZahl2());
}


bool operator <= (Zahlen& a, Zahlen& b) {
    return (a.getZahl1() <= b.getZahl1() && a.getZahl2() <= b.getZahl2());
}

//.....
```

7.5.3 Inkrement- und Dekrement-Operator

C++ kennt spezielle Operatoren um ganze Zahlen um eins zu erhöhen (increment) oder eins zu erniedrigen (decrement). Der Incrementoperator ‚++' ist nichts anderes als eine verkürzte Schreibweise von:

```
int a = 10;
a++; // entspricht  a = a + 1;
```

Den In- / Decrementoperator gibt es in einer Pre- und Postfix Notation. Sie unterscheiden sich lediglich durch den Rückgabewert. Folgender Programmausschnitt:

```
int a = 5;
cout << a++ << endl;
cout << a << endl;
cout << ++a << endl;
cout << a << endl;
```

liefert folgende Ausgabe:

```
5
6
7
7
```

Vorüberlegung zur Überladung:

- Eine Unterscheidung zwischen Prefix und Postfix wird durch einen nicht verwendeten (dummy) int-Parameter ermöglicht.
- Die unäre Operation wird als Memeberfunktion deklariert:
  RückgabeTyp operator++(); // Prefix: ++a
  RückgabeTyp operator++(int); // Postfix: a++
- Prefix: Rückgabewert ist die um 1 erhöhte Zahl
- Postfix: Rückgabewert ist die noch unveränderte Zahl

```
#include <cstdlib>
#include <iostream>

using namespace std;

class Zahlen {
    private:
        float zahl1, zahl2;
    public:
        Zahlen(): zahl1(0), zahl2(0) {}
        Zahlen(float z1, float z2): zahl1(z1), zahl2(z2) {}

        // Prefix Increment
        Zahlen& operator++() {
            zahl1++;
            zahl2++;
            return *this;
        }

        // Postfix Increment
        const Zahlen operator++ (int) {
            Zahlen t = *this;
            ++(*this);
            return t;
        }

        friend ostream& operator<< (ostream& os, const Zahlen& b);
};

// Ausgabe
ostream& operator<< (ostream& os, const Zahlen& a) {
      os << "Zahl1; " << a.zahl1 << " Zahl2: " << a.zahl2 << endl;
      return os;
}

int main()
{
   Zahlen x(15,6), y(10,18), z;
   // Postfix
   cout << x++;
   // Prefix
   cout << ++x;

   system("pause");
   return 0;
}
```

7.5.4 Indexoperator [ ]

Der Indexoperator [ ] wird meist überladen, um auf einzelne Vektorelementen zuzugreifen. Er ist ein unärer Operator, hat also zwei Operanden. Ist z.B. v ein Vektor, so ist im Ausdruck v[i] der Vektorname v der linke Operand und der Index i der rechte Operand.

Es ist zu beachten, dass die Operatorfunktion stets eine Methode der Klasse ist, die nur einen Übergabeparameter für den rechten Operanden besitzt. Daher gilt folgendes:

- der linke Operand muss Klassenobjekt sein
- der rechte Operand darf einen beliebigen Datentyp haben
- der Ergebnis-Typ ist nicht festgelegt.

Da der Index einen beliebigen Datentyp haben darf, haben Sie viele Anwendungsmöglichkeiten. Z.B. können sehr leicht assoziative Arrays definiert werden, d.h. Arrays deren Elemente über Strings angesprochen werden.

```
#include <cstdlib>
#include <iostream>
#define MAX 10 // globale Konstante

using namespace std;

class Zahlen {
    private:
        float zahl1[MAX];
    public:
        float& operator[] (int i) {
            if (i < 0 || i >= MAX) {
              cerr << "Zahlen: Out of Range!" << endl;
              exit(1);
            }
            return zahl1[i]   // Referenz auf das i-te Element
        }
        static int getMaxIndex() { return MAX - 1; }
};

int main()
{
   Zahlen zufall; // Vektor anlegen
   int i = 0; // Index

   for (i = 0; i <= Zahlen::getMaxIndex(); i++) {
      zufall[i] = rand();        // Zufallszahl wird generiert und zugewiesen. rand() befindet sich in der Bibliothek cstdlib
   }

   for (int j = 0; j <= (MAX - 1); j++) {
      cout << zufall[j] << endl;
   }

   system("pause");
   return 0;
}
```

8 Templates

Das Ziel, das man mit dem Einsatz von standardisierten Vorlagen verfolgt, ist den Aufwand bei der Programmentwicklung zu reduzieren, da nicht alle Programmteile neu erfunden und selbst entwickelt werden müssen. Auch beim Bau eines neuen Autos wird das Rad nicht jedes Mal neu erfunden.

Man wird feststellen, dass sich auch in einem höchst individuellen Programm viele allgemein gültige generelle Konzepte wiederfinden. Diese generellen Konzepte kann man durch passende „Schablonen“ einmalig vorfertigen und dann entsprechend modifiziert in den eigenen Programmen immer wieder verwenden. Derartige Programmschablonen werden in C++ **Template** genannt.

C++ Templates sind Programmtextvorlagen, die einen oder mehrere Platzhalter besitzen, die in der Vorlage noch nicht näher bestimmt werden. Erst wenn aus der Vorlage Programmcode generiert wird, werden die Platzhalter mit konkretem Inhalt versehen. Dadurch kann eine generelle Vorlage an die jeweiligen Bedürfnisse der Programmumgebung angepasst werden.

Das Besondere an C++ Templates ist, dass Platzhalter auch für Datentypen eingesetzt werden können. Damit kann eine Klasse oder eine Funktion unabhängig von speziellen Datentypen erstellt werden. Man bezeichnet diese Art der Programmierung, die nicht an bestimmte Datentypen gebunden ist, als **generische Programmierung**. C++ kennt zwei Arten von Templates:

- **Funktions-Templates** für generische oder parametrisierte Funktionen
- **Klasse-Templates** für generische oder parametrisierte Klassen

8.1 Funktions-Templates

Als Beispiel für Funktions-Templates dient eine Sortierfunktion, die ein Feld a0, a1, …, aN aus N Elementen in aufsteigender Reihenfolge anordnen soll. Da eine derartige Sortierfunktion sehr häufig benötigt wird, müssten wir für alle möglichen Elementtypen eine eigene Sortierfunktion erstellen:

```
void sort (int a[], int n);       // für ganze Zahlen
void sort (float a[], int n);     // für Fließkommazahlen
void sort (char a[], int n);      // für Zeichen
void sort (string a[], int n);    // für Zeichenkette
……
```

Nachteile:

- für jeden Datentyp muss eine neue Funktion erstellt werden
- Fehler werden mit „copy and paste“ übernommen
- Änderungen beeinflussen unter Umständen alle Funktionen

8.1.1 Deklaration und Definition von Funktions-Templates

```
#include <cstdlib>
#include <iostream>

using namespace std;

// Prototyp-Deklaration:
template<class T> void sort(T[], int);

// Funktions-Definition
template<class T> void sort(T arr[], int n)
{
   for (int i = 0; i < n-1; ++i)
   {
      // suche min in unsortierten Array
      int iMin = i;
      for (int j = i+1; j < n; ++j)
      if(arr[j] < arr[iMin])
      iMin = j;
      // tauschen
      T tmp = arr[iMin];
      arr[iMin] = arr[i];
      arr[i] = tmp;
   }
}
```

Erläuterung:

- *template* leitet die Deklaration oder Definition ein.
- In spitzen Klammern < > folgen die formalen Template-Parameter (mindestens ein Parameter muss vorhanden sein).
- Das Schlüsselwort class kennzeichnet einen Typ-Parameter. Hier können später alle Standardtypen und auch benutzerdefinierte Typen (z.B. Klassen) eingesetzt werden.
- Default-Parameter z.B. der Form class T = double sind nur bei Klassen-Templates erlaubt.
- Auch der Rückgabetyp der Funktion kann ein formaler Typ-Parameter sein.
- In der Template-Parameterliste können neben den Typ-Parametern konkrete Wertparameterdeklaration (Typ1, Typ2, ….) vorkommen. Diese dienen natürlich nicht als Platzhalter, sondern sie müssen in der Parameterliste der Funktion auftauchen:

  ```
  template < class T1, int>
  T1 funktion(T1 a, int);
  ```

- In der Parameterliste der Funktion können außer den Typ-Parametern weitere konkrete Parameter vorkommen.
- Spezifizierer wie inline, static, usw. folgen nach template <…>.

  ```
  template<class T>
  inline void fkt (T,T);
  ```

Beachte: Die Schablone für sort() könnte auch für Klassentypen verwendet werden. Dies setzt allerdings voraus, dass der Kopierkonstruktor, die Zuweisung und der Operator < korrekt arbeiten oder ggf. für diesen Zweck überladen wurden.

8.1.2 Instanzierung

Eine Template-Definition führt alleine noch nicht zu lauffähigem Programmcode. Der Compiler generiert erst für jeden konkreten Parametersatz eine spezifische Instanz der Template-Funktion.

Diese Instanzierung kann auf drei Arten erfolgen:

1. Beim erstmaligen Aufruf der Funktion leitet der Compiler aus den Typen der Funktionsparameter die nötigen Template-Parameter ab, setzt diese in die Platzhalter des Templates ein und generiert dann den vollständigen Code.
2. Ebenfalls beim erstmaligen Aufruf, aber hier werden die Template-Parameter vom Programmierer angegeben.
3. Durch explizite Deklaration mit entsprechenden Parametern (= explizite Instanzierung) wird die nötige Code-Generierung auch ohne Aufruf der Funktion angestoßen.

```
int main()
{
   int feld[] = {5,9,1,7,8};
   sort<int> (feld, 5);          // Datentyp kann auch explizit angegeben werden
   for (int j = 0; j < 5; j++) { cout << feld[j] << endl; }

   string str[] = {"Meier", "Auge", "Ohr", "Heute", "Morgen", "Bildschirm"};
   sort(str, 6); // gleichbedeutend mit sort<string>(str, 6);
   for (int i = 0; i < 6; i++) { cout << str[i] << endl; }

   system("pause");
   return 0;
}
```

Die explizite Angabe der Parameter ist eindeutiger und trägt zum besseren Verständnis von Programmen bei. Wenn man also *sort<int>* lediglich in *sort* abändern würde, dann ist nicht gleich ersichtlich, welcher Datentyp meinem Funktions-Template übergeben wird.

8.1.3 Überladung

Ein Funktions-Template kann durch eine normale, d.h. nicht parametrisierte Funktion oder eine andere Template-Funktion desselben Namens überladen werden (d.h. in den einzelnen Definitionen unterscheiden sich Typen und/oder Anzahl der Argumente).

```
// Überladung einer Template Funktion
void sort(string arr[], int n) {

   for (int i = 0; i < n-1; ++i)
   {
      // suche min in unsortierten Array
      int iMin = i;
      for (int j = i+1; j < n; ++j)
      if(arr[j] < arr[iMin])
      iMin = j;
      // tauschen
      string tmp = arr[iMin];
      arr[iMin] = arr[i];
      arr[i] = tmp;
   }
}
```

Ist der Funktionsaufruf nicht mit <….> qualifiziert und liegen Template-Funktionen, Spezialisierungen und überladene Funktionen vor, geht der Compiler bei der Auswahl der „passenden“ Funktion wie folgt vor:

- Existiert eine normale Funktion oder eine Spezialisierung mit exakt passenden Parametern?
  - nimm diese!
- Sonst: kann eine passende Template-Funktion erzeugt werden?
  - nimm diese!
- Sonst: kommt eine normale Funktion mit impliziten Typkonvertierungen in Frage? (Standardregel für überladene Funktionen)
  - nimm diese!
- Sonst: Liegen Mehrdeutigkeiten vor!
  - Fehlermeldung!

8.2 Klassen-Templates

Auch ein Klassen-Template kann Platzhalter für Datentypen verwenden. Es beschreibt daher eine ganze Menge von Klassen, aus der erst durch die Angabe eines konkreten Datentyps eine individuelle Klasse generiert wird. Klassen-Templates sind ein weiterer wichtiger Mechanismus für das generische Programmieren.

Das Hauptanwendungsgebiet von Klassen-Templates findet sich in der Standard Template Library (STL). Diese Bibliothek besteht aus einer ganzen Reihe von verschiedenen Vorlagen für Klassen und Funktionen, ist international standardisiert und ist damit ein fester Bestandteil der C++ Programmierumgebung.

Ein Großteil der Klassen-Templates der STL sind sogenannte Container-Klassen. Ein Container dient der Verwaltung von vielen Objekten eines bestimmten Typs. Der Datentyp der zu verwaltenden Objekte ist Parameter des jeweiligen Templates. Es stehen in der STL eine Reihe verschiedenartiger Container bereits „fertig programmiert“ zur Verfügung. Sie müssen nur noch für den gewünschten Datentyp instanziiert werden.

Deklaration und Definition von Klassen-Templates

Die Deklaration eines Klassen-Templates ist einem Funktions-Template sehr ähnlich.

```
// Definition einer Klassen-Template:
template <class T1, class T2, …, Typ1 bez1, Typ2 bez2, …>
class KlassenName {
        //…….
};
```

- Standardargumente sind erlaubt:
  Typ-Parameter: …., class T2 = double, …..
  Wert-Parameter: …... int bez1 = 10, ……
- Ein Typ-Parameter kann nur dann vorbesetzt werden, wenn alle nachfolgenden Typ-Parameter in der Liste ebenfalls vorbesetzt sind.
- Die Typen von Wert-Parametern sind auf ganzzahlige Typen beschränkt. Auch sie können vorbesetzt werden.
- Wert-Parameter dienen dazu, um beim Instanzieren einer Klasse einen konstanten Wert in die Klassendefinition einzubringen. Sie können nicht verändert oder referenziert werden.
- Elementfunktionen, die außerhalb der Klassendefinition definiert werden, sind automatisch Template-Funktionen mit den gleichen Parametern und müssen entsprechend formuliert werden.

Beispiel: Klassen-Templates und Spezialisierung

```
#include <cstdlib>
#include <iostream>

using namespace std;

template <class T>
class two {
   private:
      T a, b;
   public:
      two (T first, T second) {
         a=first;
         b=second;
      }
      T getmax ();
};

template <class T>
T two<T>::getmax ()
{
   T retval;
   retval = a>b ? a : b;
   return retval;
}

// Explizite Deklaration für int
template <>
int two<int>::getmax() {
   int retval;
   retval = a>b? a : b;
   return retval;
}

int main () {
   // implizite Instanzierung
   two<double> myobjectd(2.5,4.7);
   // explizite Instanzierung
   two<int> myobjectv(8,4);
   cout << myobjectd.getmax() << endl;
   cout << myobjectv.getmax();

   system("pause");
   return 0;
}
```

Instanzierung

Die Instanzierung kann implizit oder explizit geschehen. Angegeben werden muss in beiden Fällen (falls keine Default-Argumente eingesetzt werden):

- für Typ-Parameter: der konkrete Typ
- für Wert-Parameter: ein konstanter, ganzzahliger Ausdruck