Einführung in die Programmierung zur Numerik mit C++

WS 2010/2011

Skript: Bärbel Schlake und Patrick Henning

Stand: 20. Januar 2011

Inhaltsverzeichnis

Kapitel 1

Einführung

Die Programmiersprache C++ unterstützt die objektorientierte Programmierung (OOP). Im Folgenden werden einige wichtige Begriffe kurz erläutert.

Objekt: Ein Objekt ist eine gekapselte Datenstruktur. Objekte enthalten sowohl Datenkomponenten als auch Funktionen, die diese bearbeiten. Diese Funktionen werden auch Methoden genannt.

Klasse: Eine Klasse ist der Typ eines Objekts. Sie beschreibt Typ und Aufbau der Datenkomponenten und Methoden. Man sagt auch ein Objekt ist eine Instanz einer Klasse.

Kapselung: Die Kompetenz und die Verantwortlichkeit zur Verarbeitung von Objektdaten liegt im Objekt selbst. Unautorisierte Zugriffe von außen sind nicht mehr möglich.

Vererbung: Man kann in C++ neue Klassen aus bereits definierten bilden. Diese Wiederverwendbarkeit wird Vererbung genannt. Die neue Klasse übernimmt Datenkomponenten und Methoden der Basisklasse und fügt eigene hinzu oder modifiziert die geerbten.

Polymorphie: Polymorphie ist die Eigenschaft einer Klasse oder Funktion, in verschiedenen Zusammenhängen verschieden zu wirken. Polymorphie bezeichnet denselben Ausdruck in verschiedenen Formen. Beispielsweise kann eine Funktion mehrfach mit identischem Namen implementiert werden, solange sich die Argumente in dn einzelnen Implementierungen eindeutig unerscheiden.

1.1 Kleine Einführung in Unix

Das Compilieren und Linken von C++ Programmen führen wir über die Konsole/das Terminal aus. Das Terminal wird auf CentOS Rechnern folgendermaßen gestartet: *Launch->Anwendungen->Zubehör->Terminal.*

Die wichtigsten Befehle in der Konsole sind:

- `cd <Verzeichnisname>`
  (**cd** → **c**hange **d**irectory): Wechselt in das Verzeichnis `<Verzeichnisname>`
- `cd ..`
  Wechselt in das nächste Oberverzeichnis. Nach Aufrufen von `cd ..` im Verzeichnis `Beispiel/Datei` befindet man sich im Verzeichnis `Beispiel`.
- `ls`
  (**ls** → **l**i**s**t): Listet den Inhalt des aktuellen Verzeichnisses auf.
- `ls -l`
  (**ls -l** → **l**i**s**t **l**ong): **-l** ist hier die Option für mehr Details. Der Befehl listet den Inhalt des aktuellen Verzeichnisses mit Zusatzinformationen auf.
- `rm <Dateiname>`
  (**rm** → **r**e**m**ove): Die Datei `<Dateiname>` wird (unwiderruflich) gelöscht.
- `rm -r <Verzeichnisname>`
  -r steht hier für die Option eines **r**ekursiven Löschens, d.h. mit dem Befehl kann statt einer einzelnen Datei auch ein Verzeichnis gelöscht werden.
- `rmdir <Verzeichnisname>`
  (**rm** → **r**e**m**ove **dir**ectory): eine Alternative zum vorherigen Befehl. Hier wird ebenfalls das gesamte Verzeichnis `<Verzeichnisname>` gelöscht.
- `cp <Quelldatei> <Zielpfad/Dateiname>`
  (**cp** → **c**o**p**y): Die Datei `<Quelldatei>` wird nach `<Zielpfad>` kopiert und erhält den Namen `<Dateiname>`.
- `cp -r <QuellVZ> <Zielpfad/Verzeichnisname>`
  (**cp** → **c**o**p**y): **-r** steht wieder für die Option eines **r**ekursiven Ausführens, d.h. diesmal wird ein ganzes Verzeichnis `<QuellVZ>` nach `<Zielpfad>` kopiert und erhält den Namen `<Verzeichnisname>`.
- `mv <Quelldatei> <Zielpfad/QuelldateiNeu>`
  (**mv** → **m**o**v**e): Die Datei `<Quelldatei>` wird nach `<Zielpfad>` verschoben und in `<QuelldateiNeu>` umbenannt.
- `touch <Dateiname>`
  Erstellet eine leere Datei mit dem Namen `<Dateiname>`.
- `pwd`
  (**pwd** → **p**rint **w**orking **d**irectory): Dieser Befehl gibt den Namen des aktuellen Verzeichnisses aus.
- `mkdir <Verzeichnisname>`
  (**mkdir** → **m**a**k**e **dir**ectory). Es wird ein Verzeichnis mit dem Namen `<Verzeichnisname>` angelegt.

- `man <Befehlsname>`
  (**man** → **man**ual): Hilfe für die Verwendung eines Befehls. Hier wird die Manual-Page ('Handbuch') des Kommandos `<Befehlsname>` aufgerufen.

Ein '**-**' (z.B. in **-l** oder **-r**) signalisiert eine *Option* für einen Befehl.

Wenn man die ↑-Taste drückt, steht in der Konsole wieder der Befehl, den man zuletzt eingegeben hatte.

Unix sowie C++ unterscheiden die Groß- und Kleinschreibung!

1.1.1 `gnuplot`

Falls man die Daten, die man in eine Datei geschrieben hat, plotten möchte, kann man dies mit Hilfe des Funktionsplotters `gnuplot` realisieren. Er wird über die Konsole mit `gnuplot` aufgerufen. Der Befehl zum Plotten der Werte, die paarweise pro Zeile in der Datei `data` stehen, lautet

```
plot "data"
```

Zum Plotten mehrerer Datensätze in ein Diagramm schreibt man

```
plot "data1", "data2"
```

1.2 Erste Schritte

Die Erstellung eines C++-Programms erfolgt in mehreren Arbeitsschritten:

1. Der Quelltext wird mit einem Editor, z. B. `gedit` oder `kate` erstellt und abgespeichert, etwa unter dem Namen `hello.cc` (die Endung `.cc` gibt an, dass es sich um eine C++-Quelldatei handelt). Andere Compiler erfordern evtl. eine andere Endung der Datei, z.B. `.cpp`.

2. Man übersetzt das Programm mit `g++ -o hello hello.cc` in ein ausführbares Programm. `g++` ist der Befehl zum compilieren und linken, der Zusatz `-o hello` bewirkt, dass das ausführbare Programm den Namen `hello` hat.

3. Mit der Eingabe von `./hello` führt man das Programm dann aus.

1.3 Das erste Programm

Das Programm `hello.cc`, sieht folgendermaßen aus (die Zeilennummern dienen nur zur Orientierung, sie gehören nicht in den Quellcode):

```
1: #include<iostream>
2: using namespace std;
3:
4: //Hier beginnt das Programm
5: int main()
6: {
7:   cout << "Hello World!" << endl;
8:   return 0;
9: }
```

Die Ausgabe auf der Konsole sieht wie folgt aus:

```
Hello World!
```

Nun die zeilenweise Analyse des Programms:

Zeile eins bindet die Datei `iostream.h` der Standardbibliothek in die Datei `hello.cc` ein (siehe auch Kapitel 10). Dies wirkt auf den Compiler, als wenn der gesamte Inhalt der Datei `iostream.h` unmittelbar am Anfang der Datei `hello.cc` stände. Das Symbol `#` ist eine Anweisung an den Präprozessor. Der Präprozessor durchsucht den Quellcode nach Zeilen, die mit `#` beginnen. Sobald er eine solche findet, führt er entsprechende Aktionen aus, die den Code ändern und übergibt ihn in geänderter Form dem Compiler.

Die zweite Zeile weist den Compiler an, den Namespace `std.` (standard) zu verwenden. Dies ist für die in Zeile sieben verwendeten Befehle `cout` und `endl` wichtig, die sonst nicht bekannt wären. Es ist auch möglich, Zeile zwei durch folgende Zeilen zu ersetzen:

```
using std::cout;
using std::endl;
```

Man kann Zeile zwei auch weglassen und Zeile sieben durch Folgende ersetzen:

```
7: std::cout << "Hello World!" << std::endl;
```

So sind nur die beiden Befehle `cout` und `endl` bekannt, und es kann nicht zu etwaigen Namenskollisionen mit selbst benannten Objekten kommen.

In Zeile vier wird ein Kommentar durch // eingeleitet. Die gesamte Zeile wird vom Compiler ignoriert. Es gibt auch noch eine andere Kommentarart, die aus C übernommen wurde und mit der man ganze Blöcke ausklammern kann: Diese Kommentare beginnen mit /* und enden mit */. Alles zwischen /* und */wird vom Compiler ignoriert.

In Zeile fünf beginnt das Hauptprogramm mit der Funktion `main`. Jedes C++-Programm verfügt über eine `main`-Funktion. Dies ist ein Codeblock, der eine oder

mehrere Aktionen ausführt. Beim Starten des Programms erfolgt der Aufruf von `main` automatisch. Der Rückgabewert der Funktion `main` ist immer `int`, also eine ganze Zahl. Alle Funktionen beginnen mit einer geöffneten Klammer { (Zeile sechs) und enden mit einer geschlossenen Klammer }(Zeile neun). Alles zwischen diesen beiden Klammern ist Teil der Funktion. Der Befehl `cout` (c out) bewirkt die Ausgabe auf die Konsole. Alles, was nach dem Operator `«` steht, wird auf den Bildschirm ausgegeben. Um eine Zeichenkette wie oben auszugeben, muss man sie in Anführungszeichen setzen. Der Befehl `endl` (end of line) sorgt für einen Zeilenumbruch in der Konsole. Die Anweisung `"\n"` sorgt - ähnlich wie `endl`- für einen Zeilenumbruch.

Zeile acht gibt den Wert 0 an das Betriebssystem zurück. In manchen Systemen gibt dieser Wert an, ob die Funktion fehlerfrei ausgeführt wurde. Per Konvention kennzeichnet 0 eine fehlerfreie Operation.

Funktionen werden während des Programms von `main` oder weiteren Funktion aufgerufen. Ein Programm wird zeilenweise entsprechend der Reihenfolge im Quellcode ausgeführt. Wenn eine Funktion aufgerufen wird, verzweigt dass Programm und führt die aufgerufene Funktion aus. Nach Abarbeitung der Funktion führt das Programm in der nächsten Zeile der aufrufenden Funktion fort.

Leerzeichen und Tabulatoren gehören zu den **Whitespaces** in C++. Der Compiler ignoriert diese Zeichen. Whitespaces werden eingefügt, damit sich der Quellcode leichter lesen lässt.

1.4 `make`

Es ist umständlich, bei jeder Übersetzung eines Programms den Befehl `g++ -o hello hello.cc` in die Konsole zu tippen. Dies kann vereinfacht werden durch den Befehl `make`. Dazu muss man in dem Ordner, in dem man arbeiten möchte, eine Datei mit dem Namen `makefile` oder `Makefile` anlegen. Inhalt der Datei sollte sein:

```
CXXFLAGS = -g
```

Der Befehl `-g` ist nötig zum Gebrauch des Debuggers, also zur Fehlersuche (auf die Details gehen wir später ein). Der Compiler weiß automatisch, wie er eine C++-Datei zu kompilieren hat. Nun reicht es, in der Konsole auf SUN Solaris Rechnern statt `g++ -g -o hello hello.cc` einfach `gmake hello` zu schreiben. Weitere Optionen sind zum Beispiel `-O1, -O2, -O3` für verschiedene Optimierungslevel. Auf Details zum Befehl `make` gehen wir im Laufe des Kurses noch ein. Auf Linux-Rechnern lautet der Befehl `make hello`.

1.5 Der Debugger

Ein Debugger hilft, Fehler in Programmen zu finden. Mithilfe des Debuggers ist es möglich, den Programmverlauf zu verfolgen und Werte zu überprüfen. Die Program-

me springen bei ihrer Ausführung unter Umständen in Funktionen oder andere Implementationen. Der Debugger macht es einfacher, diese Sprünge nachzuvollziehen. Wir benutzen den GNU Debugger GDB. Die graphische Oberfläche dieses Debuggers wird mit Hilfe des Befehls `ddd` gestartet. Es ist allerdings nur möglich, den Debugger zu nutzen, wenn man beim Kompilieren den Optionsparameter `-g` benutzt hat. Falls der Debugger auf den Rechnern in der Universität nicht richtig funktioniert, ist ein ändern der Sprachoption nötig. Dies geschieht in der Konsole durch Aufruf von

```
unset LANG
```

Der Debugger würde also nach dem Kompilieren von obiger Datei `hello.cc` folgendermaßen aufgerufen werden:

```
ddd hello
```

Nun wird die graphische Oberfläche des Debuggers automatisch gestartet. Mit Hilfe von Breakpoints ist es möglich, das Programm an einer bestimmten Stelle anzuhalten, beispielsweise am Aufruf einer Funktion oder einer Schleife. Ein Breakpoint wird bei Benutzung der Oberfläche mit einem Rechtsklick und *->Set Breakpoint* in das Programm eingefügt. Mit einem erneuten Rechtsklick auf den Breakpoint kann man ihn entweder deaktivieren (*->Disable Breakpoint*) oder ihn ganz löschen (*->Delete Breakpoint*).

Die wichtigsten Befehle des Debuggers sind:

- `run`: Startet das Programm; das Programm läuft bis zum ersten Breakpoint.
- `step`: Das Programm läuft zeilenweise weiter, der Debugger springt in Funktionsaufrufe.
- `next`: Das Programm läuft zeilenweise weiter, der Debugger springt allerdings nicht in Funktionsaufrufe.
- `display var`: Zeigt den aktuellen Wert einer Variable `var` im oberen Fenster an. (Rechtsklick auf die Variable `var` und dann `Display var`.)

Der Debugger wird durch Eingabe des Befehls `quit` im unteren Fenster des Debuggers wieder verlassen.

1.6 Variablen und Konstanten

Variablen sind 'Behälter' in denen das Programm Zahlen und Zeichenfolgen ablegt. Genau genommen ist eine Variable eine Stelle im Hauptspeicher des Computers, in der man einen Wert ablegen und später wieder abrufen kann. Über den Variablennamen kann man auf den Inhalt des betreffenden Speicherbereichs zugreifen, ihn verändern oder lesen. Es gibt verschiedene Typen von Variablen. Diese verschiedenen Typen belegen unterschiedlich viel Speicherplatz. Die wichtigsten Typen sind:

- `char`: speichert ein Zeichen, üblicherweise aus dem ASCII
- `int`: ganzzahliger Wert, Größe: 2 Byte
- `float`: Gleitkommazahl, Größe: 4 Byte
- `double`: Gleitkommazahl, Größe: 8 Byte
- `bool`: Wahrheitswert, speichert true oder false
- `char*`: speichert eine Zeichenkette
- `long`: speichert eine lange Integerzahl, Variante des `int`, Größe: 4 Byte

Integerzahlen können zusätzlich noch in den Formaten `unsigned` und `signed` variiert werden. Bevor eine Variable im Programm benutzt werden kann, muss sie deklariert werden. Dies geschieht durch Angabe des Typs und Namens der Variablen. Außerdem kann man die Variable direkt initialisieren, d.h. einen bestimmten Wert zuweisen.

```
int i;                          //Deklaration
int j = 0;                      //Deklaration und Initialisierung
int iteration1, iteration2      //Zwei Deklarationen
char a = '2';                   //Deklaration eines Zeichens
char* array = "Hello";          //Deklaration einer Zeichenkette
```

In der ersten Zeile wird die ganzzahlige Variable `i` deklariert, die zweite Zeile deklariert die ganzzahlige Variable `j` und initialisiert sie mit 0. Es können auch mehrere Variablen desselben Typs in einer Zeile deklariert werden. C++ beachtet Groß- und Kleinschreibung, also sind `i` und `I` zwei verschiedene Variablen.

Konstanten sind, wie Variablen, benannte Speicherstellen. Allerdings behalten Konstanten immer ihren Wert. Eine Konstante muss bei ihrer Deklaration initialisiert werden und behält ihren Wert während der gesamten Programmausführung. Konstanten werden mit dem Schlüsselwort `const` definiert:

```
const float pi = 3.14;
```

1.6.1 `typedef`

Vorhandenen Typen (z.B. `unsigned int`) kann mittles `typedef` ein Name gegeben werden:

```
typedef unsigned int UINT
```

Statt

```
unsigned int x
```

kann nun einfach

```
UINT x
```

geschrieben werden.

1.6.2 Aufzählungskonstanten

Aufzählungskonstanten sind ein Satz von Konstanten mit einem zugeordneten Bereich von Werten. Nach der Definition wird eine Aufzählungskonstante fast genau wie ein `int` benutzt. Die Aufzählung sieht so aus:

```
enum {Schwarz, Weiss};
```

`Schwarz` hat nun den Wert 0, `Weiss` den Wert 1 zugewiesen bekommen. `Schwarz` und `Weiss` sind nun ganzzahlige Konstanten, auch Enumeratoren genannt. Eine Aufzählung kann auch benannt sein:

```
enum Farbe {Schwarz, Weiss};
```

Es ist möglich, einzelne Konstanten mit einem bestimmten Wert zu initialisieren:

```
enum {Schwarz=50, Weiss, Gelb =500};
```

Nun hat `Schwarz` den Wert 50, `Weiss` den Wert 51, und `Gelb` den Wert 500.

1.7 Operatoren

In C++ gibt es für die Grundrechenarten die Operatoren `+, -, *, /` wie in den meisten anderen Programmiersprachen auch. Wendet man diese Verknüpfungen auf verschiedene Typen an, kommt es zu impliziten Typumwandlungen. Vorsicht ist auch geboten, wenn man den Operator `/` auf `int`- Variablen anwendet, denn der Nachkommateil wird hier abgeschnitten.

Der Modulo-Operator `%` gibt den Rest der Ganzzahldivision zurück: `23 % 5` liefert 3, da `23/5` gleich 4 mit Rest 3 ist.

Der Inkrementoperator `++` und der Dekrementoperator `--` sind zwei weitere Operatoren. `++` addiert 1 zu seinem Operanden. `x++` bewirkt also dasselbe wie `x = x+1`, `x--` bewirkt dasselbe wie `x = x-1`. Diese beiden Operatoren kann man auch vor die Variablen schreiben (`++x`), dann wird der Wert der Variablen zuerst inkrementiert und danach zugewiesen. Dies macht nur in komplexeren Anweisungen einen Unterschied. Diese Variationen werden mit Postfix bzw. Präfix bezeichnet.

Der Operator `x+=2` bewirkt dasselbe wie `x = x+2`. Analog funktionieren auch die Operatoren `-=`, `*=`, `/=` und `%=`.

Mit Vergleichsoperatoren ermittelt man das Verhältnis zweier Zahlen. Der Ausdruck `i==j` ermittelt, ob `i` denselben Wert hat wie `j`. Ein Vergleichsausdruck liefert entweder 1 (true) oder 0 (false). Zu den Vergleichsoperatoren gehören: Gleich (`==`), Ungleich (`!=`), Kleiner als (`<`), Größer als (`>`), Kleiner gleich (`<=`) und Größer gleich (`>=`).

Schließlich gibt es auch noch logische Operatoren, die ebenfalls einen Wahrheitswert zurückgeben. Die logische AND Anweisung wertet zwei Ausdrücke aus und gibt true zurück, falls beide Ausdrücke wahr sind (Syntax: Ausdruck1 `&&` Ausdruck2). Falls

Ausdruck1 schon falsch ist, wird der zweite Ausdruck gar nicht mehr überprüft. Die logische OR Anweisung gibt schon true zurück, wenn minsestens einer der Ausdrücke wahr ist (Ausdruck1 `||` Ausdruck2). Die logische NOT Anweisung invertiert den Wahrheitswert, d.h liefert true, wenn der zu testende Ausdruck false ist (`!`Ausdruck).

1.8 Ein- und Ausgabe

Die Standardausgabe `cout` haben wir schon kennengelernt. Der Befehl zur Standardeingabe lautet `cin`. Die Variable, den man einliest, muss vorher deklariert sein.

```
int x;
cin >> x;
```

Die obigen Zeilen lesen eine Integerzahl ein. Diese wird der Variablen `x` zugeordnet.

1.9 Kommandozeilenargumente

Über den `cin` Befehl ist es möglich, während des Programmablaufs Werte einzulesen (beispielsweise Werte für Variablen oder die Anzahl der Iterationen). Man kann diese Argumente aber auch direkt während des Funktionsaufrufs eingeben. Dies demonstriert folgendes Programm `multipl.cc`:

```
1:  #include <iostream>
2:
3:  using namespace std;
4:
5:  int main(int argc, char* argv[])
6:  {
7:     if (argc==3)
8:     {
9:       int x = atoi(argv[1]);
10:      double y = atof(argv[2]);
11:      cout << "x * y = " << x*y << endl;
12:    }
13:    else
14:   {
15:      cerr << "Error: two arguments expected." << endl;
16:      cerr << "Usage: "<<  argv[0] << " x y" << endl;
17:      cerr << "for x*y" << endl;
18:    }
19:   return 0;
20: }
```

In Zeile fünf erhält `main()` zwei Argumente: die Anzahl der Kommandozeilenargumente, üblicherweise `argc` genannt, und ein Feld mit den Kommandozeilenargumenten, üblicherweise `argv[]` genannt. Auf Felder gehen wir später im Kurs noch genauer ein. Die Kommandozeilenargumente sind Zeichenketten, also ist der Typ von `argv` ein `char*[argc+1]`, wobei der Name des Programms an Position `argv[0]` übergeben wird. Die Zeilen neun und zehn wandeln die Kommandozeilenargumente in Integer (`atoi`) bzw. Double (`atof`) um und weisen sie dann an `x` und `y` zu, falls es drei Kommandozeilenargumente (Name des Programms plus zwei Werte) gibt. Zeile elf gibt den berechneten Wert aus. Falls man nicht drei Kommandozeilenargumente übergeben hat, erscheint eine Fehlermeldung. `cerr` (c error) bezeichnet die Standardfehlerausgabe (analog zu `cout`). Der Benutzer wird darauf hingewiesen, dass er den Namen des Programms plus zwei Werte zu übergeben hat, damit das Programm funktioniert. Um obiges Programm für x=3 und y=4 auszuführen, muss man folgendes in die Konsole eintippen:

```
./multipl 3 4
```

1.10 Dateibearbeitung

Daten, die man mit C++ berechnet, sollen oft in Dateien geschrieben werden, damit man die Werte später noch verwenden kann (z. B. zum Plotten). Um Dateien bearbeiten zu können, muss man zunächst die Header-Datei `fstream` einbinden. Die Ausgabe auf eine Datei wird durch ein Objekt der Klasse `ofstream` eingeleitet. Das Einlesen der Daten hat eine ähnliche Syntax wie die Standardeingabe. Folgendes Programm schreibt einige Sinuswerte in die Datei `ausgabe` schreibt. Die Datei `cmath` wird für die Verwendung der Sinusfunktion mit eingebunden, `string` für den Dateinamenstring. Die Datei `string.cc` sieht folgendermaßen aus:

```
1:   #include<iostream>
2:   #include<fstream>
3:   #include<string>
4:   #include<cmath>
5:   using namespace std;
6:
7:   int main ()
8:   {
9:     string aus("ausgabe");
10:    ofstream fin(aus.c_str());
11:    for (int i = 1; i < 100; i++)
12:       fin << i*0.1 << " " << sin(i*0.1) << endl;
13:     fin.close();
14:   }
```

Die Zeichenkette `ausgabe` kann im Programm über den String `aus` angesprochen werden. In Zeile 10 wirde die Datei zum Einlesen der Werte geöffnet. Die Endung `.c_str()` bewirkt, dass `\0` hinter den String angehängt wirkt (das Programm weiß so wo der String endet). Zeile 12 bewirkt das Schreiben der Werte in die Datei.

Das Einlesen von Dateien funktioniert ähnlich. Die Datei wird mit Hilfe von `ifstream` eingelesen. Der Befehl zum Einlesen lautet `getline`.

1.11 Der Präprozessor

Bei Aufrufung des Compilers startet zunächst der Präprozessor. Dieser sucht nach Anweisungen, die mit `#` beginnen. mit der Anweisung `#define` kann man deklarieren, dass eine bestimmte Zeichenfolge definiert ist.

```
#define SMALL
```

Später prüft man die Definition von `SMALL` und leitet entsprechende Aktivitäten ein. Die Anweisungen `#ifdef` und `#ifndef` testen, ob ein String definiert oder nicht definiert ist. Der Befehl `#ifdef` liefert true, wenn die getestete Zeichenfolge definiert ist. Wenn der Präprozessor auf die `#ifdef`-Anweisung trifft, durchsucht er eine von ihm angelegte Tabelle, ob die angegebene Zeichenfolge definiert ist. Ist dies der Fall, so liefert `#ifdef` das Ergebnis true und der Code bis zum nächsten `#else`- oder `#endif`-Befehl wird ausgeführt. Falls die Auswertung von `#ifdef` false ergibt, so wird der Code zwischen `#ifdef` und `#endif` nicht ausgewertet.

Das logische Gegenstück zu `#ifdef` ist `#ifndef`. Diese Anweisung liefert true, wenn die Zeichenfolge bis zu diesem Punkt nicht in der Datei definiert wird. Zwischen `#ifdef` bzw. `#ifndef` und `#endif` lässt sich noch der Befehl `#else` einfügen.

Kapitel 2

Funktionen

Mit Funktionen kann man in C++-Programmen bestimmte Aufgaben erledigen. Mit der Funktionsdefinition gibt man an, was die Funktion tun soll. Eine Funktion kann nicht aufgerufen werden, solange sie nicht deklariert wurde. Funktionen geben entweder einen Wert oder `void` (das heißt kein Rückgabetyp) zurück. Funktionen gliedern sich in Kopf und Rumpf. Der Kopf besteht aus dem Rückgabetyp, dem Namen der Funktion und den Parametern. Ein typischer Funktionskopf sieht folgendermaßen aus:

```
int sum(int a, int b)
```

Der Name der Funktion samt Parameter wird auch als **Signatur** bezeichnet. Der Rumpf einer Funktion besteht aus einer öffnenden geschweiften Klammer, einer beliebigen Zahl von Anweisungen, und einer schließenden geschweiften Klammer. Funktionen können einen Wert mit Hilfe der `return` Anweisung zurückgeben. Falls man keine `return` Anweisung vorsieht, liefert die Funktion automatisch `void` zurück. Folgende Funktion `add.cc` liest zwei Werte ein und gibt die Summe der beiden Zahlen:

```
1: #include<iostream>
2: using namespace std;
3:
4: int add(int x, int y);
5:
6: int main()
7: {
8:    int a, b, c;
9:    cout << "Bitte geben Sie zwei Zahlen ein: " << endl;
10:   cin >> a;
11:   cin >> b,
12:   c = add(a,b);
13:   cout << "Die Summe von " << a << " und " << b << " ist " << c << endl;
14:   return 0;
```

```
15: }
16:
17: int add(int x, int y)
18: {
19:    return (x+y);
20: }
```

Die Funktionsdeklaration steht in Zeile vier. Die Funktionsdefinition steht in den Zeilen 18 bis 20. Die Funktionsdefiniton und -deklaration müssen bis auf das Semikolon identisch sein, sonst erhält man einen Compilerfehler. Die Anweisungen innerhalb des Rumpfes der Funktion müssen mit einem Semikolon abgeschlossen sein.

Variablen, die in Funktionen deklariert sind, heißen **lokale Variablen**. Sie existieren nur innerhalb der Funktion. Variablen, die außerhalb aller Funktionen definiert sind, heißen **globale Variablen**. Sie sind in jeder Funktion sowie in `main` verfügbar. Lokale Variablen verdecken globale: Bei Variablen mit gleichem Namen wird bei Aufruf dieser Variablen zuerst in der lokalen Ebene gesucht. Im Allgemeinen vermeidet man globale Variablen.

Funktionsargumente müssen nicht alle vom selben Typ sein.

Es ist auch möglich, Funktionen als Parameter zu übergeben.

Die an eine Funktion übergebenen Werte sind lokal. Das demonstriert folgendes Programm `swap_wert.cc`.

```
#include <iostream>

void swap(int x, int y);

int main()
{
    int x = 5, y = 10;
    std::cout << "Vor der Vertauschung x: " << x << " y: " << y << std::endl;
    swap(x,y);
    std::cout << "Nach der Vertauschung x: " << x << " y: " << y << std::endl;
    return 0;
}

void swap (int x, int y)
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

Die Ausgabe lautet:

```
Vor der Vertauschung x: 5 y: 10
Nach der Vertauschung x: 5 y: 10
```

Die Funktion `swap` legt lokale Kopien der Werte an und vertauscht nur diese Kopien. Nach Rückkehr aus der Funktion `void` in die Funktion `main` werden diese lokalen Kopien gelöscht. In `main` ist kein Unterschied festzustellen.

2.1 Rückgabewerte

Funktionen geben entweder einen Wert oder `void` zurück. Das Schlüsselwort hierfür ist `return`, gefolgt von dem zurückzugebenden Wert. Dieser Wert kann auch ein Ausdruck sein, der einen Wert liefert.

```
return 1;
return (x>1);
```

Der Wert in der zweiten Anweisung ist 1, falls `x` größer als 1 ist, sonst 0. Alle Anweisungen, die nach `return` stehen, werden nicht mehr ausgeführt.

2.2 Standardparameter

Falls man eine Funktion

```
int function (int x)
```

deklariert hat, so muss diese Funktion tatsächlich eine Integer-Variable übernehmen, damit man keinen Compilerfehler erhält. Man kann allerdings auch mit Standard- oder Defaultparametern arbeiten. Dies bedeutet, dass die Funktion einen Standardparameter verwendet, falls man keinen anderen Wert bereitstellt.

```
int function (int x=50)
```

Falls man diese Funktion ohne Argument aufruft, benutzt sie den Standardwert 50. Die Funktionsdefinition ändert sich hierdurch nicht. Man kann allen Parametern einer Funktion oder einem Teil davon Standardwerte zuweisen. Die Regeln hierfür sind wie folgt: Der Funktion `int function (int x, int y, int z)` kann für `y` nur ein Standardwert zugeordnet werden, wenn man für `z` einen Standardwert festgelegt hat. An `x` lässt sich nur dann ein Standardwert zuweisen, wenn sowohl `y` als auch `z` Standardwerte haben.

2.3 Funktionen überladen

Es ist in C++ möglich, mehrere Funktionen mit demselben Namen zu erzeugen, die sich nur in der Parameterliste (durch unterschiedliche Typen oder eine andere Anzahl an Parametern) unterscheiden.

```
int function (int x, int y);
int function (double x, double y);
int function (int x);
```

Ein Überladen bezüglich des Rückgabetyps ist nicht möglich. Das Überladen von Funktionen bezeichnet man auch als **Funktionspolymorphie**.

2.4 Inline Funktionen

Normalerweise springt der Compiler beim Aufruf einer Funktion zu den Anweisungen der Funktion und geht bei Rückkehr aus dieser Funktion in die nächste Zeile der aufrufenden Funktion. Das bedeutet für den Prozessor zusäzlich Arbeit. Falls man oft Funktionen benutzt, die nur aus wenigen Zeilen bestehen, empfielt sich daher eine andere Technik. Man definiert diese Funktion als `inline`. Dann erzeugt der Compiler keine Funktion im engeren Sinn, sondern kopiert den Code für die Inline-Funktion direkt in die aufrufende Funktion. Die Sprünge des Compilers fallen weg und es ist so, als hätte man die Anweisungen der Funktion direkt in die aufrufende Funktion geschrieben. Dadurch wird die Datei größer. Generell kommen nur Funktionen als `inline` in Frage, die sehr klein sind (wenige Anweisungen). Die Deklaration einer Inline-Funktion schreibt sich:

```
inline int function (int);
```

Die Deklaration einer Funktion als `inline` ist nur eine Empfehlung an den Compiler, die dieser auch ignorieren kann.

2.5 `if`-Anweisung

Wie in anderen Programmiersprachen kann man in C++ auch die `if`-Anweisung verwenden. Die Syntax ist

```
if (Ausdruck1)
  Anweisung1;
```

Solange `Ausdruck1` wahr ist, wird die Anweisung ausgeführt. Auch die `else`-Anweisung kann verwendet werden:

```
if (Ausdruck1)
  Anweisung1;
else
  Anweisung2;
```

Auch kompliziertere `if-else`-Anweisungen sind möglich, allerdings müssen Aneisungen über mehrere Zeilen in geschweiften Klammern stehen.

```
if (Ausdruck1)
  {
    if (Ausdruck2)
       Anweisung1;
    else
    {
      if (Ausdruck3)
         Anweisung2;
      else
         Anweisung3;
    }
}
else
  Anweisung4;
```

2.6 `switch`-Anweisung

Mit der `switch`-Anweisung lassen sich Verzweigungen in Abhängigkeit von mehreren unterschiedlichen Werten aufbauen. Die Synatx einer allgemeinen `switch`-Anweisung lautet:

```
switch (Ausdruck)
{
case Wert1: Anweisung;
   break;
case Wert2: Anweisung;
   break;
...
case WertN: Anweisung;
   break;
default:    Anweisung;
}
```

`Ausdruck` ist jeder gültige C++-Ausdruck, `Anweisung` jede beliebige C++-Anweisung. `switch` wertet `Ausdruck` aus und vergleicht das Ergebnis mit den Werten hinter

`case`. Stimmt kein Wert überein, wird die Anweisung hinter `default` ausgeführt. Die Auswertung erfolgt nur auf Gleichheit. Falls die `break`-Anweisung am Ende eines `case`-Zweigs fehlt, führt das Programm den nächsten `case`-Zweig aus.

2.7 Schleifen

2.7.1 `while`

Eine `while`-Schleife wiederholt eine Folge von Anweisungen, solange die Startbedingung wahr ist.

```
while(Ausdruck)
{
  Anweisung1;
  Anweisung2;
}
```

Falls das Programm zum Anfang der `while`-Schleife zurückkehren soll, noch bevor alle Anweisungen abgearbeitet sind, benutzt man die `continue`-Anweisung. Die `break`-Anweisung führt zum Austritt aus der `while`-Schleife und die Programmausführung setzt sich nach der schließenden Klammer fort.

2.7.2 `for`

Die `for`-Schleife hat drei Anweisungen. In der ersten Anweisung wird üblicherweise die Zählervariable initialisiert. Die zweite Anweisung testet eine Bedingung und die dritte Anweisung inkrementiert oder dekrementiert normalerweise die Zählervariable. Eine typische `for`-Schleife sieht wie folgt aus:

```
for (int i=0; i<10; i++)
{
   cout << "i= " << i << endl;
}
```

Man kann auch mehrere Variablen in einer Schleife initialisieren und inkrementieren:

```
for (int i=0; j=0; i<10; i++, j++)
{
   cout << "i= " << i << " j: " << j << endl;
}
```

2.8 Statische Variablen

Statische Variablen sind mit globalen Variablen vergleichbar. Sie werden nur beim ersten Aufruf der Funktion durch ihre Definition initialisiert. Das Beispiel `static.cc` verdeutlicht dies:

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

void f(int a)
{
  while(a>0)
  {
   static int n = 0;
   int x = 0;
   cout << "n == " << n++ << "; x == " << x++ << endl;
   a--;
  }
}

int main()
{
   f(3);
}
```

mit der Ausgabe

```
n == 0; x == 0
n == 1; x == 0
n == 2; x == 0
```

Die Variable `n` wird nur einmal initialisiert, die Variable `x` hingegen `a`-mal.

2.9 Typcast

Zum Typcast dienen die Operatoren `static_cast`, `dynamic_cast` und `const_cast`. Die Syntax (für alle Operatoren gleich) ist

`static_cast<Typ>(Ausdruck)`

Beim Aufruf der Operatoren wird `Ausdruck` geprüft und in `Typ` umgewandelt. `static_cast` castet zwischen elementaren Datentypen und Klassen. `dynamic_cast` castet Zeiger oder Referenzen auf miteinander verwandte ploymorphe Klassen. Dieser Cast wird im Gegensatz zum statischen Cast zur Laufzeit ausgeführt. `const_cast` dient zur Entfernung der Konstantheit eines Typs.

Kapitel 3

Zeiger

Ein **Zeiger** ist eine Variable, die eine Speicheradresse enthält. Jede Variable eines Programms befindet sich an einer eindeutig addressierten Speicherstelle. Mit Hilfe des Adressoperators `&` kann man diese Adresse ermitteln. Die Adresse wird im Hexadezimalsystem ausgegeben. Beisiel `adresse.cc`

```
1:  #include<iostream>
2:   using namespace std;
3:
4:  int main()
5:  {
6:    int x=5;
7:    cout << "Die Adresse von x ist " << &x << endl;
8:  }
```

hat die Ausgabe (die Ausgabe der Adresse kann ein etwas anderes Aussehen haben):

```
Die Adresse von x ist 0xbfc3ea00
```

Für einen Typ `T` ist `T*` der Typ „Zeiger auf T“. Also kann eine Variable vom Typ `T*` die Adresse eines Objekts vom Typ `T` speichern.

Einen Zeiger kann man so erzeugen:

```
int x=0;
int* px = &x;
```

Es ist nicht wichtig, wo genau der `*` Operator steht. Folgende Anweisungen sind identisch:

```
int* px = &x;
int * px = &x;
int *px = &x;
```

px wird als Zeiger auf x deklariert. Es ist wichtig, alle Zeiger bei Erzeugung auch zu initialisieren. Wenn man nicht weiß, was man dem Zeiger zuweisen soll, wählt man den Wert 0. So kann man testen ob der Zeiger auf etwas (0) zeigt. Nicht initialisierte Zeiger sind gefährlich, falls man diese im Programm verwendet.

Den Wert der im Zeiger gespeicherten Adresse ruf man mit Hilfe des Dereferenzierugsoperators * ab. Der Operator * kommt bei Zeigern in zwei Versionen vor: Deklaration und Dereferenz. Bei der Deklaration gibt * an, das es sich um einen Zeiger und nicht um eine normale Variable handelt.

```
int* px=0;
```

Bei der Dereferentierung gibt das Sternchen an, dass man auf den Wert der im Zeiger gespeicherten Adresse zugreift und nicht auf die Adresse selbst.

```
*px=1;
```

Um mit Zeigern umgehen zu können muss man also drei Dinge auseinander halten können:

- Zeiger
- die vom Zeiger gespeicherte Adresse
- Wert an der im Zeiger enthaltenen Adresse

Es ist auch möglich, Daten mithilfe von Zeigern zu manipulieren, wie folgendes Programm pointer_manipulate.cc zeigt:

```
1:  #include<iostream>
2:   using namespace std;
3:
4:  int main()
5:  {
6:    int x=5;
7:    int* px=&x;
8:    cout << " x = " << x << endl;
9:    cout << " *px= " << *px << endl;
10:   *px=10;
11:   cout << " x = " << x << endl;
12:   cout << " *px = " << *px << endl;
13:  }
```

Die Ausgabe dieses Programms sieht so aus:

```
x = 5
*px = 5
x = 10
*px = 10
```

Es ist folglich möglich, Werte von Variablen mit Hilfe von Zeigern zu verändern.

3.1 `new` und `delete`

Mit dem Schlüsselwort `new` erzeugt man neue Objekte auf dem Freispeicher. Der Rückgabewert von `new` ist eine Speicheradresse, welcher man einem Zeiger zuweisen muss.

```
int* px = new int;
```

Nun zeigt `px` auf einen `int`. Man sagt auch, dass die Variable **dynamisch allokiert** wurde.

Mit dem Befehl `delete` gibt man den Speicherbereich wieder frei. Ein mit `new` reservierter Speicher muss mit `delete` wieder frei gegeben werden.

```
delete px;
```

Für jeden Aufruf von `new` sollte ein korrespondierender Aufruf von `delete` vorhanden sein.

3.2 Konstante Zeiger

Bei Zeigern kann man das Schlüsselwort `const` wie folgt verwenden:

```
const int* p1;
int* const p2;
const int* const p3;
```

`p1` ist ein Zeiger auf eine konstante ganze Zahl. Man kann also nicht schreiben `*p1=1`. `p2` ist ein konstanter Zeiger auf eine ganze Zahl. Man kann die ganze Zahl ändern, aber man kann nicht schreiben `p2=&y`. `p3` ist ein konstanter Zeiger auf eine konstante ganze Zahl. Man kann weder die ganze Zahl ändern, noch den Zeiger auf eine andere Adresse zeigen lassen.

3.3 Referenz

Eine **Referenz** ist ein zusätzlicher Name für eine Variable (Alias). Die Referenz selbst ist keine eigenständige Variable. Die Deklaration erfolgt mit Hilfe des & Operators.

```
int x=3;
int& y=x;
int& z=y;
```

Jetzt bezeichnen `x`, `y`, `z` dieselbe Variable. Nach der Zuweisung `y=5` ist sowohl der Wert von `x` als auch der Wert von `z` gleich fünf. Eine Referenz muss bei der Deklaration initialisiert werden. Die Adresse der Referenz ist identisch mit der Adresse des Ziels der Referenz. Eine Referenz muss auf ein bestehendes Objekt zeigen.

3.4 Funktionsargumente als Referenz

Wir sind nun in der Lage, das Problem, das bei der Ausführung von `swap_wert.cc` in Kapitel 2 auftauchte, zu beheben. Zur Erinnerung: Wir haben der Funktion `swap` Parameter als Werte übergeben. Die Funktion hat lokale Kopien dieser Werte vertauscht. In der Hauptfunktion `main` waren die Werte unverändert. In der folgenden Funktion `swap_pointer.cc` übergeben wir nicht die Werte, sondern die Adressen der Werte.

```
1:  #include <iostream>
2:
3:   void swap(int *px, int *py);
4:
5:   int main()
6:   {
7:       int x = 5, y = 10;
8:       std::cout << "Vor der Vertauschung x: " << x << " y: " << y << std::endl;
9:       swap(&x,&y);
10:      std::cout << "Nach der Vertauschung x: " << x << " y: " << y << std::endl;
11:      return 0;
12:  }
13:
14:  void swap (int *px, int *py)
15:  {
16:      int temp;
17:      temp = *px;
18:      *px = *py;
19:      *py = temp;
20:  }
```

Die Ausgabe lautet:

```
Vor der Vertauschung x: 5 y: 10
Nach der Vertauschung x: 10 y: 5
```

Das Übergeben der Adressen statt Werte an `swap` hat also funktioniert. Die Werte, die als Adressen an `swap` übergeben wurden, sind nun tatsächlich vertauscht. Das Programm funktioniert zwar, allerdings ist das Programm durch die ständige Dereferenzierung schwer zu lesen. Daher nun das Programm `swap_referenz.cc`, welches mit Referenzen arbeitet:

```
1:  #include <iostream>
2:
3:   void swap(int &x, int &y);
```

```
4:
5:   int main()
6:   {
7:       int x = 5, y = 10;
8:       std::cout << "Vor der Vertauschung x: " << x << " y: " << y << std::endl;
9:       swap(x,y);
10:      std::cout << "Nach der Vertauschung x: " << x << " y: " << y << std::endl;
11:      return 0;
12:  }
13:
14:  void swap (int& rx, int& ry)
15:  {
16:      int temp;
17:      temp = rx;
18:      rx = ry;
19:      ry = temp;
20:  }
```

Die Ausgabe lautet:

```
Vor der Vertauschung x: 5 y: 10
Nach der Vertauschung x: 10 y: 5
```

In Zeile neun werden die Variablen `x` und `y` an die Funktion `swap` übergeben. Beim Aufruf von `swap` springt das Programm in Zeile 14. Die Variablen sind hier als Referenzen gekennzeichnet. Da die Parameter an `swap` als Referenz deklariert sind, werden die Werte aus `main` als Referenz übergeben und sind demnach in `main` vertauscht.

3.5 Rückgabe mehrerer Werte

Mit Hilfe von Referenzen ist es möglich, mehrere Werte aus einer Funktion zurückzugeben (`many_arguments.cc`).

```
1:  #include <iostream>
2:
3:    using namespace std;
4:
5:  int xmal(int x, int& res3, int& res6);
6:
7:  int main()
8:  {
9:    int y=5, loes3, loes6;
```

```
10:   xmal(y, loes3, loes6);
11:   cout << loes3 << endl;
12:   cout << loes6 << endl;
13: }
14:
15: int xmal(int x, int& res3, int& res6)
16: {
17:   res3 = 3*x;
18:   res6 = 6*x;
19: }
```

Die Variablen `loes3` und `loes6`, die in der Funktion `xmal` berechnet wurden, sind also in der Funktion `main` bekannt, da sie als Referenz übergeben wurden.

3.6 Felder und Arrays

Ein Array oder Datenfeld ist eine Zusammenfassung von Speicherstellen für Daten desselben Datentyps.

`int myArray[10];`

deklariert ein Array namens `myArray` mit 10 Zahlen des Typs `int`. Auf das erste Array-Element greift man mit `myArray[0]` zu, auf das zehnte mit `myArray[9]`. Ein Array `Array` mit `n` Elementen ist folglich von `Array[0]` bis `Array[n-1]` numeriert. Falls man in obigem Beispiel auf das Element `myArray[12]` schreibt, ignoriert der Compiler, dass es dieses Element eigentlich nicht gibt und schreibt auf den Wert an der Speicherstelle, an der das Element eigentlich liegen würde. Dies kann zu unvorhergesehenen Ergebnissen führen, denn der Wert, der sich dort befindet, wird einfach übergeschrieben. Ein Array für drei Integerzahlen wird folgendermaßen initialisiert:

```
int IntArray[3]={1, 2, 3};
```

Das Element `int IntArray[0]` hat den Wert 1 zugewiesen bekommen, `int IntArray[1]` den Wert 2 usw. Wenn man die Größe des Arrays nicht angibt, wird sie entsprechend der Initialisierungswerte erzeugt.

```
int IntArray[] = {1, 2, 3};
```

erzeugt dasselbe Array wie oben. Die Anweisung

```
int IntArray[3] = {1, 2, 3, 4};
```

lässt sich nicht realisieren, sie führt zu einem Compilerfehler, da man ein dreielementiges Array mit vier Werten initialisieren will. Die Anweisung

```
int IntArray[3] = {1, 2};
```

funktioniert, hier werden nicht initialisierte Werte (in unserem Fall `IntArray[2]`) automatisch auf Null gesetzt.

3.6.1 Mehrdimensionale Arrays

Es ist auch einfach möglich, mehrdimensionale Arrays zu initialisieren. Ein zweidimensionales Array hat zwei Indizes, ein dreidimensionales drei u.s.w. Die Initialiseirung funktioniert mittels

```
int Array[4][2] = {1,2,3,4,5,6,7,8};
```

oder übersichtlicher

```
int Array[4][2] = { {1,2},
                    {3,4},
                    {5,6},
                    {7,8} };
```

3.6.2 Arrays und Zeiger

Folgende Deklarationen muss man unterscheiden:

```
int array1[5];
int* array2[5];
int* array3 = new int[5];
```

`array1` ist ein Array von fünf `int` Objekten, `array2` ist ein Array von fünf Zeigern auf `int` Objekte und `array3` ist ein Zeiger auf ein Array mit fünf `int` Objekten. Ein Zeiger auf ein Array zeigt konstant auf die Adresse des ersten Elements in diesem Array.

Ein weiteres Anwendungsgebiet von Pointern sind Listen. In Listen lassen sich Daten einer festen Größe speichern. Diese Größe muss vor Erzeugung der Liste fest gelegt sein. Die einzelnen Elemente der Liste heißen Knoten, der erste Knoten wird als Kopf bezeichnet, der letzte als Endknoten. In einer einfach verketteten Liste speichert jeder Knoten einen Wert und einen Zeiger, der auf das nächste Element zeigt. Um einen bestimmten Knoten zu finden startet man am Anfang der Liste und tastet sich von Knoten zu Knoten. In einer doppelt verketteten Liste speichert jeder Knoten der Liste zwei Zeiger: Einen, der auf das vorherige Element zeigt und einen, der auf das nächste Element zeigt. Es ist möglich, aus einer Liste beliebige Elemente zu entfernen oder einzufügen, indem man die Zeiger, die auf diese Elemente zeigen, entspreichen manipuliert.

3.6.3 `char*`

Man kann eine Zeichenkette mit Hilfe von

```
char array[] = "Hello";
```

einlesen. Die Größe des Feldes `array` beträgt nun sechs Zeichen, das letzte Zeichen wird durch `'\0'` belegt, welches den Abschluss eines Stringzeichens angibt. Die Einlesung vom Zeichenketten ist auch fogendermaßen möglich (noch ein Erbe aus C):

```
char* array_stern = "Hello";
```

Es ist allerdings bei Benutzung von `char` oder `char*` nicht möglich, diese Zeichenkette über Zugriff auf einzelne ELemente zu verändern:

```
array_stern[2] = "u";
```

3.7 Dynamische Vergabe von Speicherplatz

Mit Hilfe von Zeigern kann man Speicherplatz dynamisch vergeben. Damit ist es möglich, die Größe eines Feldes erst zur Laufzeit eines Programms festzulegen. Um eine Matrix, also ein zwei-dimensionales Feld dynamisch zu allokieren, musste man folgendes schreiben:

```
matrix = new double*[size_x];
for (int i = 0; i < size_x; ++i)
  matrix[i] = new double[size_y];
```

In der ersten Zeile weist man Matrix einen Zeiger auf das erste Element eines Vektors zu, dessen Elemente wiederum Vektoren sind (Zeile drei). Nun ist es möglich, auf Elemente von `matrix` mittels `matrix[i][j]` zuzugreifen.

Kapitel 4

Klassen

Wir kommen nun zu einem sehr wichtigen Aspekt der OOT: zu den **Klassen**. Eine Klasse ist eine vom Benutzer definierte Datenstruktur, eine Sammlung von Variablen (unterschiedlichen Typs) und eine Gruppe verwandter Funktionen. Ein **Objekt** ist eine Instanz einer Klasse. Die Daten eines Objekts werden auch **Komponenten** oder **Member** genannt. Eine **Methode** ist eine Funktion dieser Klasse, auch **Memberfunktion** genannt. Die Memberfunktionen sind an die Klasse gebunden und für alle Objekte der Klasse identisch. Sie können nur über ein konkretes Objekt aufgerufen werden. Der Begriff **Kapselung** bezeichnet die Zusammenfassung aller Informationen, Fähigkeiten und Zuständigkeiten einer Einheit zu einem einzelnen Objekt.

Zum Schutz vor unberechtigten Zugriffen unterteilt man die Klassenkomponenten in private und öffentliche. Die privaten Komponenten sind nur von Memberfunktionen der eigenen Klasse ansprechbar. Die öffentlichen unterliegen keinen Einschränkungen. Die öffentlichen Memberfunktionen bilden daher die **funktionale Schnittstelle** zur Außenwelt.

4.1 Klassendeklaration

Klassendeklarationen beginnen mit dem Schlüsselwort `class` und bestehen aus einem öffentlichen und privaten Teil:

```
class Klasse
{
  public:
   //öffentliche Komponenten...
  private:
    //private Komponenten...
};
```

Man kann die Deklaration auch mit `struct` einleiten (ein Erbe aus C). Die Voreinstellung ist hier `public`.

Ein Objekt des neuen Typs definiert man fast so wie eine neue Integer-Variable. Angenommen, wir haben eine Klasse `Dog`, dann kann man ein Objekt dieser Klasse namens `Nero` definieren.

`Dog Nero;`

Mit dem Punktoperator (`.`) greift man auf Variablen oder Methoden eines Objekts zu. Wir weisen 20 an die Elementvariable `age` von `Nero` zu und rufen die Methode `Bark` auf:

```
Nero.age = 20;
Nero.Bark();
```

Man weist also an Objekte, nicht an Klassen zu!

Eine allgemeine Regel der OOP it es, dass Datenelemente möglichst privat gehalten werde sollten. Daher muss man Elementfunktionen erzeugen, mit denen man die privaten Datenelemente lesen oder setzen kann.

4.2 Klassenmethoden implementieren

Jede Klassenmethode, die man deklariert, muss man auch definieren. Die Definition einer Elementfunktion beginnt mit dem Namen der Klasse gefolgt von zwei Doppelpunkten, dem Namen der Funktion und ihren Parametern, wie folgendes Beispiel `simple_class.cc` demonstriert:

```
1:  #include <iostream>
2:
3:    using namespace std;
4:
5:   class Dog
6:   {
7:     public:
8:       int GetAge();
9:       void SetAge(int age);
10:    private:
11:      int itsAge;
12:  };
13:
14:  int Dog::GetAge()
15:  {
16:      return itsAge;
```

```
17:  }
18:
19:  void Dog::SetAge(int age)
20:  {
21:      itsAge = age;
22:  }
23:
24:  int main()
25:  {
26:      Dog Nero;
27:      Nero.SetAge(3);
28:      cout << "Nero ist " << Nero.GetAge() << " Jahre alt." << endl;
29:      return 0;
30:  }
```

Die Definition der Klasse `Dog` steht in Zeile fünf bis zwölf. Das Schlüsselwort `public` in Zeile sieben teilt dem Compiler mit, dass die folgenden Funktionen `GetAge` und `SetAge` öffentlich sind. Die in Zeile elf definierte Variable `itsAge` hingegen ist privat. Die Klassendeklaration endet in Zeile zwölf mit einer geschweiften Klammer und einem Semikolon.

Die Funktion `GetAge` wird in den Zeilen 14 bis 17 definiert, sie ist eine Zugriffsfunktion auf die Zahl `itsAge` und liefert diesen Wert zurück. Der Kopf der Funktion enthält den Klassennamen, gefolgt von zwei Doppelpunkten, und den Namen der Funktion. Die restliche Definition unterscheidet sich nicht von der Definition für jede andere Funktion. Die Funktion `main` kann nicht auf `itsAge` zugreifen, da diese Variable privat ist. `main` hat aber Zugriff auf `GetAge`, und daher kann `GetAge` den Wert von `itsAge` an `main` zurückliefern.

Die Funktion `SetAge` wird in den Zeilen 19 bis 22 definiert. Diese Funktion übernimmt einen ganzzahligen Parameter und setzt den Wert von `itsAge` auf diesen Parameter (Zeile 21). Als Elementfunktion hat `SetAge` direkten Zugriff auf `itsAge`.

In Zeile 24 beginnt das Hauptprogramm mit `main`. In Zeile 26 wird ein `Dog` Objekt namens `Nero` erzeugt. Zeile 27 setzt das Alter von `Nero` auf 3. Der Aufruf der Funktion erfolgt mit dem Objektnamen `Nero`, gefolgt von einem Punkt (`.`) und dem Namen der Methode (`SetAge`). Zeile 28 gibt das Alter auf den Bildschirm aus.

4.3 Konstruktoren und Destruktoren

Man kann die Datenelemente einer Klasse mit Hilfe von **Konstruktoren** initialisieren. Der Konstruktor kann beliebig viele Parameter übernehmen, aber er liefert keinen Rückgabewert. Der Konstruktor ist eine Klassenmethode mit demselben Namen wie die Klasse selbst. Ein Konstrukter ohne Parameter heißt `Standardkonstrukter`. Wenn man keine Konstruktoren definiert, erzeugt der Compiler automatisch einen

Standardkonstruktor. Wenn man jedoch irgendeinen Konstruktor deklariert, stellt der Compilerr *keinen* Standardkonstruktor bereit. Beispiel:

```
class Dog
{
//...
Dog(int initialAge);
//...
};
Dog::Dog(int initialAge)
{
  itsAge=initialAge;
}
```

Konstruktoren lassen sich wie normale Funktionen überladen. Für einen Konstruktor mit einem Parameter sind folgende Deklarationsarten äquivalent:

```
Klasse variable(parameter);
Klasse variable = parameter;
Klasse variable = Klasse(parameter);
```

Ein Konstruktor wird automatisch aufgerufen, wenn ein Objekt erzeugt wird. Es ist auch möglich, den Konstruktor mit Standardwerten zu versehen:

```
Dog(int initialAge=2);
```

Zur Deklaration eines Konstruktors gehört in der Regel auch ein **Destruktor**. Die Destruktoren geben nach Abbau des Objekts den reservierten Speicher frei. Ein Destruktor hat immer den Namen der Klasse, wobei eine Tilde (`~`) vorangestellt ist. Destruktoren haben keinen Funktionstyp, übernehmen keine Argumente, haben keinen Rückgabewert und können auch nicht überladen werden. Der Destruktor der Klasse `Dog` ist also:

```
 ~Dog();
```

Der Compiler stellen einen Standarddestruktor bereit, falls man keinen definiert.

4.3.1 Der Kopierkonstruktor

Der Compiler stellt neben Standardkonstruktor und -destruktor auch einen Standardkopierkonstruktor bereit. Der Aufruf des Kopierkonstruktors erfolgt jedes Mal, wenn eine Kopie des Objekts angelegt wird. Alle Kopierkonstruktoren übernehmen einen Parameter: eine Referenz auf ein Objekt derselben Klasse. Diese Referenz sollte als konstant deklariert werden, da das Objekt nicht verändert werden muss.

Der Standardkopierkonstruktor kopiert jede Elementvariable von dem als Parameter übergebenem Objekt in die Elementvariable des neuen Objekts. Man spricht hier von einer elementweisen oder flachen Kopie. Bei Elementvariablen, die Zeiger auf Objekte sind, klappt dies schon nicht mehr. Die Zeiger beider Objekte verweisen auf denselben Speicher. Falls ein Objekt nun gelöscht wird, gibt der Destruktor den Speicher wieder frei. Das andere Objekt verweist aber immer noch auf den Speicher. Dies führt zu Fehlern in der Ausführung des Programms.

Der Ausweg ist, sich einen eigenen Kopierkonstruktor zu definieren, der eine tiefe Kopie anlegt. Eine tiefe Kopie legt neuen Speicher an. Die Zeiger verweisen nun nicht mehr auf denselben Speicher. Das Löschen eines Objekts ist unproblematisch. Der Kopierkonstruktor der Klasse `Dog` aus 4.2 würde folgendermaßen implementiert werden

```
Dog::Dog(const Dog& rhs)
{
itsAge = new int;
*itsAge = rhs.GetAge();
}
```

Ein Objekt wird dann wie folgt implementiert:

```
Dog Nero         //Erstellen eines Objekts
...
Dog Carus(Nero)  //Carus ist nun einen Kopie von Nero
```

Der Parameter ist wie bei einem Kopierkonstruktor üblich mit `rhs` (right-hand-side) benannt. Die erste Anweisung reserviert den benötigten Speicher. In der zweiten Anweisung werden die Werte aus dem existierende Objekt auf die neue Speicherstelle übergeben. Nun haben beide Objekte verschiedene Speicherstellen.

4.4 Konstante Memberfunktionen

Auch für Memberfunktionen kann der Zusatz `const` angegeben werden, und zwar unmittelbar hinter dem Funktionskopf. Beispiel:

```
int f() const;
```

Das Schlüsselwort `const` muss sowohl in der Funktionsdeklaration als auch in der -definition vorkommen.

Konstanten Memberfunktionen ist es nicht möglich, den Wert irgendeines Elements der Klasse zu ändern. In obigem Beispiel kann nur die Funktion `GetAge` als konstant deklariert werden, da diese Funktion nichts in der Klasse ändert. `SetAge` kann nicht als `const` deklariert werden, da diese den Wert von `itsAge` ändert. Wann immer möglich, sollte man Funktionen als `const` deklarieren. Falls man ungewollterweise mit

konstanten Funktionen versucht, Werte zu ändern, bekommt man einen Compiler-Fehler. Diese Fehler sind weitaus angenehmer als Fehler, die erst während der Laufzeit eines Programmes gefunden werden.

4.5 Header-Dateien

Es gehört zum guten Programmierstil, die Deklaration einer Klasse und das Hauptprogramm zu trennen. Gewöhnlich wird die Deklaration in einer sogenannten Header-Datei untergebracht, die den gleichen Namen hat wie die Klasse, aber auf `.h` endet. Auch die Implementation kann noch in eine eigene Datei geschrieben werden. Diese hat wieder die Endung `.cc`. Das Hauptprogramm und die Implementierungsdatei müssen die Deklaration der Klasse mittels `#include"Name.h"` einbinden.

4.6 Inline-Implementierung

In C++ ist es möglich, Memberfunktionen als `inline` zu implementieren. Die Inline-Implementierung der Funktion `GetAge` sieht folgendermaßen aus:

```
inline int Dog::GetAge()
{
    return itsAge;
}
```

Man kann die Definition einer Funktion auch in die Deklaration der Klasse schreiben, was die Funktion automatisch zu einer Inline-Funktion macht:

```
class Dog
{
public:
    int GetAge() {return itsAge};
    void SetAge(int age) {itsAge = age};
};
```

4.7 Klassen in Klassen

Eine Klassenkomponente kann in C++ jeden beliebigen Typ haben, also auch den einer anderen Klasse. So kann die Klasse `Car` beispielsweise aus den Klassen `Wheel`, `Motor` usw. bestehen. Dies entspricht einer „hat ein“ Beziehung.

4.8 Zeiger und Klassen

Zeiger können auf beliebige Objekte zeigen. Wenn man ein Objekt vom Typ `Dog` erzeugt hat, ist es möglich, einen Zeiger auf diese Klasse deklarieren. Die Syntax entspricht der in Kap. 3.1 eingeführten:

```
Dog* pDog = new Dog;
```

Das Objekt wird mit `delete` gelöscht:

```
delete pDog;
```

Auf Datenelemente greift man mit mit dem Elementverweisoperator (->) zu. C++ behandelt diese Zeichenfolge als einzelnes Symbol. In obiger `Dog`-Klasse könnte man also folgendermaßen auf die Funktion `SetAge` zugreifen:

```
Dog* pDog = new Dog;
pDog->SetAge(2);
```

Jede Elementfunktion einer Klasse verfügt über die spezielle Größe `this`, einen Zeiger auf das aktuelle Objekt. `this` weist auf das jeweilige Objekt, dessen Methode aufgerufen wurde. In der oben eingeführten Funktion `SetAge` ist `itsAge` also nur eine Abkürzung für:

```
void SetAge(int age) {this->itsAge = age};
```

Notwendig ist die Verwendung von `this` vor allem in zwei Fällen: Wenn in der Memberfunktion eine Funktion aufgerufen wird, der das aktuelle Objekt (genauer: ein Zeiger darauf) als Parameter übergeben werden soll oder wenn die Funktion einen Zeiger oder eine Referenz auf das aktuelle Objekt als Funktionswert zurückgibt (`return this` oder `return *this`).

4.9 Überladen von Operatoren

Operatoren lassen sich in C++ wie Funktionen überladen. Die zu überladenden Operatoren werden auf Funktionen abgebildet. Der Name dieser Funktion beginnt mit `operator`, gefolgt von dem Operatorsymbol, also z.B. `operator+`. Operatorfunktionen auf Klassenobjekte lassen sich wie normale Funktionen überladen, sofern die Signatur verschieden ist. Es ist allerdings nicht möglich, Operatorfunktionen Standardparameter zu übergeben. Eine Operatorfunktion hat so viele Parameter, wie der Operator Operanden hat. Mindestens einer der Operatoren muss ein Klassenobjekt sein. man unterscheidet wieder zwei Fälle:

1. Die Funktion ist Memberfunktion. Der erste Operand ist das aktuelle Objekt. Es gibt einen Parameter weniger als Operanden.

2. Die Funktion ist normale Funktion. Der erste Parameter entspricht dem linken, der zweite Operand dem rechten Parameter.

Folgendes Beispiel überlädt den `operator+`:

```
#include<iostream>

class Counter
{
public:
  Counter();
  Counter(int initialValue);
 ~Counter(){}
  int GetItsVal() const {return itsVal;}
  Counter operator+ (const Counter &);
private:
  int itsVal;
};

Counter::Counter(int initialValue):
itsVal(initialValue)
{}

Counter::Counter():
itsVal(0)
{}

Counter Counter::operator+ (const Counter & rhs)
{
  return Counter (itsVal + rhs.GetItsVal());
}
int main()
{
  Counter One(3), Two(4), Three;
  Three = One + Two;
  std::cout << "One: " << One.GetItsVal() << std::endl;
  std::cout << "Two: " << Two.GetItsVal() << std::endl;
  std::cout << "Three: " << Three.GetItsVal() << std::endl;
  return 0;
}
```

4.10 Statische Objektkomponenten und Memberfunktionen

In C++ braucht man gelegentlich Komponenten, die einer Klasse und nicht einem individuellen Objekt gehören (wenn man beispielsweise mitzählen möchte, wie oft man ein bestimmtes Objekt gebildet hat). Hierfür verwendet man in der objektorientierten Programmierung normalerweise keine globalen Variablen. In jedes Objekt eine entsprechende Komponente einzufügen wäre Platzverschwendung. Für dieses Problem gibt es in C++ statische Member. Diese Objekte sind durch den Zusatz `static` gekennzeichnet und nicht an ein einzelnes Objekt, sondern an eine Klasse gebunden und gehören allen Objekten gemeinsam. Von jeder statischen Komponente gibt es genau ein Exemplar, welches unmittelbar nach Definition der Klasse verfügbar ist, auch wenn noch kein Objekt dieser Klasse existiert. Statische Member sind mit globalen Variablen vergleichbar. `static`-Komponenten werden wie andere Komponenten angesprochen und unterliegen denselben Schutzmechanismen (`private/public`). Falls ein Objekt eine statische Komponente verändert, so ist der Wert auch für alle anderen Objekte geändert. Statische Member müssen definiert werden. Es muss genau eine Definition geben, welche in irgendeinem beteiligten Programm-Modul stehen kann. Das Schlüsselwort `static` darf bei der Definition nicht wiederholt werden. Die Verlagerung aus der Klassendeklaration unterstreicht, dass das Member nicht einem individuellen Objekt gehört. Die Initialisierung muss vor Erzeugung des ersten Objekts stattfinden.

Man kann nicht nur Datenkomponenten, sondern auch Memberfunktionen als `static` deklarieren. Dies hat prinzipiell die gleiche Wirkung wie bei Datenkomponenten. Die Funktion sollte über die Klasse aufgerufen werden. In statischen Funktionen gibt es keinen `this` Zeiger. Man kann also nicht auf normale Objektkomponenten zugreifen. In statischen Funktionen können ausschließlich statische Klassenkomponenten verwandt und statische Memberfunktionen aufgerufen werden. Beispiel:

```
#include <iostream>

class Dog
{
 public:
  Dog() {HowManyDogs++;}
  ~Dog() {HowManyDogs--;}
  int GetAge() {return itsAge;}
  void SetAge (int age){itsAge = age;}
  static int GetHowMany() {return HowManyDogs;}
 private:
  int itsAge;
  static int HowManyDogs;
};
```

```
int Dog::HowManyDogs = 0;

int main()
{
  Dog *Dogsarray[3];
  for (int i = 0; i < 3; i++)
  {
    Dogsarray[i] = new Dog;
    std::cout << "Es existieren " << Dog::GetHowMany() << " Hunde." << std::endl;
  }
  for (int i = 0; i < 3; i++)
  {
    delete Dogsarray[i];
    std::cout << "Es existieren " << Dog::GetHowMany() << " Hunde." << std::endl;
  }
  return 0;
}
```

Mit der Ausgabe:

```
Es existieren 1 Hunde.
Es existieren 2 Hunde.
Es existieren 3 Hunde.
Es existieren 2 Hunde.
Es existieren 1 Hunde.
Es existieren 0 Hunde.
```

Kapitel 5

makefile

Mit dem Befehl `make` übersetzt man ein Programm, welches man geschrieben hat, nicht direkt, sondern man liest aus der Datei `makefile` die Informationen, welche Datei mit welchen Optionen übersetzt und verbunden wird. Das `makefile` wird erst bei größeren Programmen wirklich wichtig. Wir geben hier nur eine kleine Einführung in den Gebrauch von `make`.

Der Befehl `make` erwartet eine Datei `makefile` oder `Makefile`, die im aktuellen Arbeitsverzeichnis steht. In dieser Datei steht, was `make` machen soll. Nach dem Aufruf von `make` liest das Programm das `makefile` ein und interpretiert es. Falls das Programm eine Datei mit anderem Namen, z.B. `anderesMakefile` benutzen soll, muss man `make` mit dem Parameter `-f anderesMakefile` aufrufen.

Die Regeln innerhalb des `makefile` genügen folgender Syntax:

```
Ziel... : Abhängigkeiten ..
<Tab>Befehl
<Tab>...
```

Das Ziel hängt von seinen Abhängigkeiten ab. Die Befehle darunter geben an, wie das Ziel gebildet wird. **Wichtig:** Die jeweils zweiten Zeilen für eine Regel müssen mit einem Tabulator-Befehl beginnen! Kommentare beginnen im `makefile` mit `#`.

Compileroptionen sind:

- `-Wall`, alle Warnungen des Compiler werden ausgegeben
- `-g`, nötig zum Gebrauch des Debuggers
- `-O1`, `-O2`, `-O3`, verschiedene Optimierungslevel
- `-I`, Einbinden von Headerverzeichnissen
- `-c`, nur kompilieren, nicht linken

Hierzu ein Beispiel. Das Programm `programm` soll aus den Quelldateien `programm.cc` und `source.cc` gebildet werden. `source.cc` bindet die Headerdatei `source.h` ein. Das `makefile` sieht wie folgt aus:

```
programm : programm.o source.o
          g++ programm.o source.o -o myprg

porgramm.o : programm.cc source.h
          g++ -c programm.cc

source.o : source.cc source.h
          g++ -c source.cc

clean :
         rm -f programm programm.o source.o

rebuild : clean programm
```

Beim Aufruf von `make` geschieht Folgendes: `make` liest das makefile ein und interpretiert es. Zu Beginn sind alle Quelldateien (`*.cc`) und die Headerdatei (`*.h`) vorhanden. Die Objektdateien (`*.o`) und die Programmdatei `programm` ist noch nicht vorhanden.

Die erste Regel in unserem `makefile` gibt an, dass für das Linken von `programm` die Objektdateien `programm.o` und `source.o` vorhanden sein müssen. Das ausführbare Programm heißt `myprg` nach Aufruf von `make`. Also sucht `make` nach Regeln zum Erzeugen der Objektdateien `programm.o` und `source.o`.

Die zweite Regel besagt, dass `programm` auf die Quelldatei `programm.cc` und die Headerdatei `source.h` angewiesen ist. Diese Dateien sind vorhanden, deshalb führt `make` den Befehl `g++ -c programm.cc -o programm.o` aus, erzeugt also `programm.o`. Danach wird auf ähnliche Weise `source.o` erzeugt. Nun sind die erforderlichen Programme zum Linken des Hauptprogramms `programm` vorhanden und `programm` wird erzeugt.

Die vorletzte Regel (`clean`) dient zum Aufräumen. Sie wird mittels `make clean` aufgerufen und sorg für das Löschen der ausführbaren Programmdatei sowie der Objektdateien.

`rebuild` vereinfacht das Neuerzeugen. Ruft man `make rebuild` auf, löscht man zuerst mittels `clean` und führt danach `make` aus.

Der Vorteil des `makefile` ist es, dass `make` bemerkt, ob `source.cc` seit der letzten Ausführung geändert wurde (und ergo das Datum von `source.cc` aktueller ist als das Datum von `source.o`). Die zugehörige Befehlszeile wird ausgeführt, also `source.cc` neu kompiliert. Am Schluss werden die Objektdateien zu einer neuen Version von

`programm` zusammengelinkt. Dieses Feature wird wichtig, wenn ein Programm aus vielen Objektdateien besteht. Hier werden nur die Quelldateien neu gelinkt, die tatsächlich geändert wurden. Dies erspart viel Arbeit.

Wir verwenden nun Variablen und schreiben obiges `makefile` neu. Variablen werden folgendermaßen definiert:

```
VAR = WERT
```

Man verwendet die Variablen `VAR` mittels:

```
$(VAR)
```

Wir verwenden nun in Variablen Compiler (`CXX`), Compilerparameter (`CXXFLAGS`), Linker (`CC`) und Objektdateien (`OBJS`). Unser Programm liest sich nun

```
OBJS = programm.o source.o
CXX = g++
CC = $(CXX)
CXXFLAGS = -Wall -g -O2

all : programm

programm : $(OBJS)

programm.o source.o : source.h

clean :
	rm -f programm $(OBJS)

rebuild: clean programm
```

Kapitel 6

Vererbung

Ein wichtiges Prinzip der OOP ist die Vererbung. Ziel ist es, Bestandteile eines Programms so zu konzipieren, dass sie leicht wiederverwendbar sind. Man möchte in neuen Klassen alte Deklarationen verwenden und leicht variieren, das heißt um einige Komponenten erweitern oder bestimmte Methoden anders definieren. Man nennt die Ausgangsklasse **Basisklasse**, die neue Klasse **abgeleitete Klasse**. Eine Klasse kann auch aus mehreren Basisklassen abgeleitet werden, man spricht dann von **multipler Vererbung**. eine abgeleitete Klasse besitzt alle Eigenschaften ihrer Basisklasse und auch noch die, die sie selbst definert. Ausnahmen sind Konstruktoren, Destruktoren und `operator=()`. Diese werden grundsätzlich nicht vererbt. Eine abgeleitete Klasse kann wieder als Basisklasse dienen.

6.1 Syntax der Ableitung

Neben `public` und `private` kann man Komponenten einer Klasse noch als `protected` deklarieren. `protected`-Komponenten einer Klasse verhalten sich nach außen hin wie private, sind aber in einer abgeleiteten Klasse verfügbar, im Gegensatz zu `private`-Komponenten.

Es gibt außerdem drei Vererbungsarten: `public`, `protected` und `private`.

Um die Klasse `X` öffentlich aus der Klasse `Y` abzuleiten, schreibt man

```
class X: public Y {...}
```

Die öffentliche Ableitung `public` übernimmt die Zugriffsrechte unverändert aus der Basisklasse. Alle Komponenten von `Y`, die als `public` oder `protected` deklariert sind, können in `X` angesprochen werden, solche, die `private` sind, nicht.

Bei der `protected` Ableitung

```
class X: protected Y {...}
```

haben die Komponenten, die in `Y` als `public` deklariert wurden in `X` den Status `protected`. `private` und `protected`-Komponenten von `Y` sind in `X` privat.

Bei der `private` Ableitung

```
class X: private Y {...}
```

sind Komponenten, die in `Y` `public` oder `protected` sind, in `X` `private`.

Das Programm `inherit.cc` soll das Prinzip der Vererbung verdeutlichen:

```
1:   #include <iostream>
2:
3:   class Mammal
4:   {
5:   public:
6:       Mammal();
7:       ~Mammal();
8:
9:       int GetAge() const { return itsAge; }
10:      void SetAge(int age) { itsAge = age; }
11:
12:      void Eat() const { std::cout << "Das Tier frisst." << std::endl; }
13:
14: protected:
15:      int itsAge;
16:  };
17:
18:  class Dog : public Mammal
19:  {
20:  public:
21:      Dog();
22:      ~Dog();
23:
24:      void WagTail() { std::cout << "Schwanzwedeln." << std::endl;  }
25:  };
26:
27:  Mammal::Mammal()
28:  { }
29:
30:  Mammal::~Mammal()
31:  { }
32:
33:  Dog::Dog()
34:  { }
35:
```

```
36:  Dog::~Dog()
37:  { }
38:
39:  int main()
40:  {
41:      Dog Nero;
42:      Nero.Eat();
43:      Nero.WagTail();
44:      Nero.SetAge(3);
45:      std::cout << "Nero ist "<< Nero.GetAge() << " Jahre alt." << std::endl;
46:      return 0;
47: }
```

Zeile drei bis 16 deklarieren die Klasse `Mammal`. In unserem Beispiel ist dies die Basisklasse. Die Klasse `Mammal` hat die Funktionen zum Setzen und Zurückgeben des Alters. Darüberhinaus hat sie noch die Funktion `Eat` (alle Säugetiere essen). Die Klasse `Dog` leitet sich öffentlich von der Klasse `Mammal` ab (in Zeile 18-25). Konstruktoren und Destruktoren von `Mammal` vererben sich nicht, müssen daher also explizit angegeben werden (Zeile 27-38)(wir hätten hier auch die Standardkonstruktoren und Destruktoren nehmen können). Auch jedes `Dog` Objekt besitzt die Elementvariable `itsAge`, sowie die Methoden zum setzen und zurückgeben des Alters und die Funktion `Eat`. Darüberhinaus besitzen `Dog`-Objekte noch die Methode `WagTail`.

6.2 Namenskollisionen

Der erste mögliche Fall einer Namenskollision kann auftreten, wenn eine abgeleitete Klasse eine Komponente definiert, deren Name bereits in einer Basisklasse verwendet wurde. Ist von einer Komponente nur der Name angegeben, so durchsucht der Compiler die Vererbungshirarchie, beginnend mit der Klasse des angesprochenen Objekts bis hin zur Basisklasse. Die erste Gefundene wird verwendet. Falls man die Komponente einer anderen Basisklasse meint, so muss man dies explizit über `Klasse::name` verwenden.

```
class X           { public: int wert; };
class Y: public X     { public: float wert; };

Y zahl;

zahl.wert = 12.34;       //Komponente von Y
zahl.Y::wert = 12.34;    //Komponente von Y
zahl.X::wert = 1234;     //Komponente von X
```

Ein weiterer Kollisionsfall liegt vor, wenn ein und derselbe Komponentenname in mehreren Basisklassen vorkommt. Hier muss immer die vollständige Bezeichung angegeben werden.

```
class X                         { public: int wert; };
class Y                         { public: int wert; };
class Z: public Y                { };
class A: public X, public Z     { };

A a;
```

Der Ausdruck `a.wert` führt zu einer Fehlermeldung, es muss entweder `a.X::wert` oder `a.Y::wert` angesprochen werden.

6.3 Initialisierung von Basisklassen

Im Konstruktor einer abgeleiteten Klasse müssen die Basisklassenkonstruktoren aufgerufen werden. Die Initialisierungen werden in der folgenden Reihenfolge durchgeführt: Zuerst werden alle Basisklassen in der Reihenfolge ihrer Deklaration initialisiert. Falls für eine Klasse kein Konstruktor angegeben ist, wird der Default-Konstruktor verwendet. Danach werden die Komponenten der abgeleiteten Klasse in der Reihenfolge ihrer Deklaration initialisiert.

Es ist natürlich auch möglich, Konstruktoren von Basisklasse und/oder abgeleiteter Klasse zu überladen. Wir erweitern in unserem Beispiel die Klasse `Dog` um die geschützte Komponente `itsWeight` und überladen die Konstruktoren für die Klassen `Mammal` und `Dog` und erzeugen verschiedene `Dog` Objekte. Das Programm `inherit_overload.cc` verdeutlicht dieses.

```
#include <iostream>

class Mammal
{
public:
    Mammal();
    Mammal(int age);
    ~Mammal();

    int GetAge() const { return itsAge; }
    void SetAge(int age) { itsAge = age; }

    void Eat() const { std::cout << "Das Tier frisst." << std::endl; }
```

```
protected:
    int itsAge;
};

class Dog : public Mammal
{
public:
    Dog();
    Dog(int age);
    Dog(int age, int weight);
    ~Dog();
    int GetWeight() const { return itsWeight; }

    void WagTail() { std::cout << "Schwanzwedeln." << endl; }

protected:
   int itsWeight;
};

Mammal::Mammal()
{ }

Mammal::Mammal(int age):
itsAge(age)
{ }

Mammal::~Mammal()
{ }

Dog::Dog():
Mammal(),
itsWeight(5)
{ }

Dog::Dog(int age):
Mammal(age),
itsWeight(6)
{}

Dog::Dog(int age, int weight):
Mammal(age),
itsWeight(weight)
{}
```

```
Dog::~Dog()
{ }

int main()
{
    Dog Nero;
    Nero.Eat();
    Nero.WagTail();
    Nero.SetAge(3);
    Dog Caesar(4);
    Dog Pluto(7,8);
    std::cout << "Nero ist " << Nero.GetAge() << " Jahre alt und ";
    std::cout << Nero.GetWeight() << " Kilo schwer.\n";
    std::cout << "Caesar ist " << Caesar.GetAge() << " Jahre alt und ";
    std::cout << Caesar.GetWeight() << " Kilo schwer.\n";
    std::cout << "Pluto ist " << Pluto.GetAge() << " Jahre alt und ";
    std::cout << Pluto.GetWeight() << " Kilo schwer.\n";
    return 0;
}
```

Die Ausgabe sieht so aus:

```
Das Tier frisst.
Schwanzwedeln.
Nero ist 3 Jahre alt und 5 Kilo schwer.
Caesar ist 4 Jahre alt und 6 Kilo schwer.
Pluto ist 7 Jahre alt und 8 Kilo schwer.
```

6.4 Funktionen redefinieren

Redefinieren einer Methode bedeutet, dass die abgeleitete Klasse die Implementierung einer Funktion der Basisklasse ändert. Wenn man ein Objekt der abgeleiteten Klasse erstellt, wird die korrekte Funktion aufgerufen. Die redefinierte Funktion muss denselben Rückgabetyp und dieselbe Signatur wie die Funktion der Basisklasse haben. Beispielsweise könnte man in obigem Beispiel die Funktion `Eat` redefinieren, indem man der Klasse `Dog` die Methode

```
void Eat() const {std::cout << "Der Hund frisst." << endl; }
```

hinzufügt.

6.5 Methoden der Basisklasse verbergen

Methoden können auch verborgen werden. Dies geschieht, wenn eine Basisklasse über eine überladene Methode verfügt und die abgeleitete Klasse diese Methode redefiniert. Dann werden alle Methoden der Basisklasse mit diesem Namen verborgen. Die Basismethode kann man aufrufen, wenn man den vollständigen Namen der Methode angibt, z.B.:

```
Mammal::Eat ()
```

Falls ein Objekt diese Methode ausführen soll, geschieht dies folgendermaßen:

```
Nero.Mammal::Eat();
```

Ein Beispiel:

```
#include <iostream>

class Mammal
{
 public:
    void Move() const { std::cout << "Das Tier geht." << std::endl; }
    void Move(int distance) const { std::cout << "Das Tier geht " << distance << " Schrit
};

class Dog : public Mammal
{
 public:
    void Move() const { std::cout << "Der Hund geht." << std::endl; }

 };

int main()
{
   Mammal Tier;
   Dog Nero;
   Tier.Move();
   Tier.Move(5);
   Nero.Move();
   //Nero.Move(4); führt zu einem Compilerfehler
   Nero.Mammal::Move(4);//so funktioniert es
  return 0;
}
```

Kapitel 7

Virtuelle Funktionen

Im Zuge der Vererbung kann es zu Problemen kommen: Wenn wir eine Basisklasse A haben, die an B_1 und B_2 vererbt, und noch eine weitere abgeleitete Klasse C, die von B_1 und B_2 erbt, so sind die Datenkomponenten von A in C doppelt vorhanden. Dies ist unerwünscht, denn Komponentennamen sind nun nicht mehr eindeutig. Es wird doppelt Speicher verbraucht und die gleichnamigen Komponenten können sogar verschiedene Werte haben. Man kann dies verhindern, indem man die Ableitung als `virtual` kennzeichnet.

```
class Dog: virtual public Mammal                {/*...*/};
```

Der Zusatz darf vor oder hinter `public`/ `private` stehen. Alle mit dem Attribut `virtual` versehenen Basisklassen innerhalb einer Vererbungshierarchie werden zu einem einzigen Komponentensatz zusammengefasst. Die Virtualität ist eine Eigenschaft der Ableitung, nicht der Basisklasse. Virtuelle Basisklassen werden zeitlich vor allen anderen initialisiert.

Es ist im Zuge der Vererbung auch möglich, virtuelle Funktionen zu definieren. Diese erlauben die „späte Bindung“: Wir weisen ein abgeleitetes Objekt an einen Zeiger auf die Basisklasse zu. Das Programm darf nicht die Memberfunktion der Basisklasse, sondern die des abgeleiteten Objekts aufrufen, auch wenn der Zeiger vom Typ „Zeiger auf Basisklasse“ ist. Wir nehmen an, dass wir eine Basisklasse `Mammal` haben, und eine abgeleitete Klasse `Dog`. Die Klasse `Mammal` besitzt eine virtuelle Funktion `Eat`, die von der Klasse `Dog` redefiniert wird. Nach Erzeugung des neuen Objekts

```
Mammal* pDog = new Dog;
```

und dem Aufruf

```
pDog->Eat():
```

wird die korrekte redefinierte Funktion der `Dog`-Klasse aufgerufen.

Das folgende Programm `late_binding.cc` demonstriert die „späte Bindung“:

```
#include <iostream>

class Mammal
{
public:
    Mammal():itsAge(1) {  }
    ~Mammal() { }
    virtual void Speak() const { std::cout << "Das Tier gibt Laut.\n"; }
protected:
    int itsAge;
};

class Dog : public Mammal
{
public:
    void Speak()const { std::cout << "Der Hund bellt.\n"; }
};


class Cat : public Mammal
{
public:
    void Speak()const { std::cout << "Die Katze miaut.\n"; }
};

int main()
{
    Mammal* Mammal_Array[3];
    Mammal* ptier;
    int choice;
    for ( int i = 0; i<3; i++)
    {
        std::cout << "(1)Hund (2)Katze: ";
        std::cin >> choice;
        switch (choice)
        {
        case 1:
            ptier = new Dog;
            break;
        case 2:
            ptier = new Cat;
            break;
        default:
            ptier = new Mammal;
```

```
            break;
        }
        Mammal_Array[i] = ptier;
    }
    for (int i=0;i<3;i++)
        Mammal_Array[i]->Speak();
    return 0;
}
```

Ausgabe:

```
(1)Hund (2)Katze: 1
(1)Hund (2)Katze: 2
(1)Hund (2)Katze: 3
Der Hund bellt.
Die Katze miaut.
Das Tier gibt Laut.
```

Obiges Programm leitet die Klassen `Dog` und `Cat` von der Basisklasse `Mammal` ab. Die Methode `Speak` von `Mammal` wird als `virtual` deklariert und in den Klassen `Dog` und `Cat` redefiniert. Der Benutzer entscheidet, welche Objekte erzeugt werden. Diese werden in einem Array von Zeigern auf `Mammal` gespeichert. Für jedes Objekt wird schließlich die jeweilige redefinierte Methode aufgerufen. Zur Kompilierzeit ist noch überhaupt nicht bekannt, welches Objekt und welche der `Speak` Methoden aufgerufen wird. Der Zeiger `ptier` wird an sein Projekt zur Laufzeit gebunden. Dies wird als dynamisches Binden bezeichnet.

Ein Objekt, welches mindestens eine virtuelle Memberfunktion enthält, nennt man auch **polymorph**. Die Aktion, welche ausgelöst wird, entscheidet sich erst zur Laufzeit. Die späte Bindung wird nur wirksam, wenn der Zugriff auf das Objekt über einen Zeiger oder eine Referenz auf den Basistyp erfolgt. Der Aufruf einer virtuellen Funktion braucht etwas mehr Rechenzeit, denn er erfolgt über einen zusätzlichen Zeiger, der eine Tabelle braucht. Diese Tabelle enthält ihrerseits Zeiger auf die virtuellen Funktionen. Für die Redefinition müssen Signatur und Returntyp der redefinierten Funktion übereinstimmen. Die abgeleitete Klasse, die eine virtuelle Funktion definieren will, muss diese sowohl deklarieren als auch definieren.

Falls man in einer Klasse eine Funktion als virtuell definiert, sollte der Destruktor ebenfalls virtuell sein.

Funktionen werden manchmal nur definiert, um abgeleiteten Klassen eine gemeinsame Schnittstelle zur Verfügung zu stellen und nicht, um tatsächlich aufgerufen zu werden. Die Tatsache, dass man eine virtuelle Funktion nur deklarieren, aber nicht definieren möchte, kann man deutlich machen, dadurch dass man die Implementation ganz weglässt. Sie wird dann durch folgende Initialisierung ersetzt:

```
virtual void Speak() = 0;
```

Die Angabe `= 0` besagt, dass die Klasse in dieser Ableitungsstufe darauf verzichtet, die Memberfunktion zu implementieren. So eine Funktion heißt rein virtuell (oder pure virtual). Klassen mit mindestens einer rein virtuellen Funktion werden **abstrakte Klassen** oder **abstrakte Datentypen (ADT)** genannt. Sie dienen lediglich als Basis für weitere Ableitungen. Man kann keine Objekte einer abstrakten Basisklasse erzeugen. Für rein virtuelle Funktionen muss die Definition der Funktion in der abgeleiteten Klasse nachgeholt werden.

Kapitel 8

Templates

Templates sind Schablonen oder Vorlagen für Klassen oder Funktionen, in denen der Typ nicht explizit angegeben ist. Dieser wird erst später angegeben. Ein Template ist eine Schablone für eine Funktions- oder Klassendefinition. Die Deklaration eines Templates wird eingeleitet durch

```
template <parameter,...>
```

Template-Parameter sind aufgebaut wie formale Parameter, aber mit dem Typspezifikator `typename` erweitert. Ein Template kann mehr als einen Typen übernehmen. Templates können auch Klassen als Parameter haben. Das Erzeugen eines bestimmten Typs von einer Template nennt man **Instanzbildung**, einzelne Klassen heißen **Instanzen** der Template.

8.1 Funktionentemplates

Bei Funktionentemplates folgt auf den Templatekopf eine ganz normale Funktionsdeklaration bzw. -definition.

```
template<typename T> void swap (T &rx, T &ry)
{
    T temp;
    temp = rx;
    rx = ry;
    ry = temp;
}
```

Diese Deklaration ist nur ein Muster für spätere konkrete Funktionsdeklarationen. Ruft man schließlich die Funktion `swap` auf, so schließt der Compiler aus dem Typ der Aufrufparameter auf den Typ `T`. Dies demonstriert das Programm `template_funktion.cc`:

```
#include <iostream>

template<typename T> void swap (T &rx, T &ry);

int main()
{
    int x = 5, y = 10;
    std::cout << "Vor der Vertauschung x: " << x << " y: " << y << std::endl;
    swap(x,y);
    std::cout << "Nach der Vertauschung x: " << x << " y: " << y << std::endl;
}

template<typename T> void swap (T &rx, T &ry)
{
    T temp;
    temp = rx;
    rx = ry;
    ry = temp;
}
```

Der Compiler ersetzt also überall `T` durch `int` und erzeugt die Funktionsdefinition

```
void swap (int &rx, int &ry)
{
    int temp;
    temp = rx;
    rx = ry;
    ry = temp;
}
```

Eine solche Funktion wird Templatefunktion genannt. Im Fall

```
int x = 3;
double y = 3.4;
swap(x, y);
```

bekommt man einen Compilerfehler, da beide Parameter denselben Typ haben müssen. Ein Funktionstemplate ist also eine Vorlage zur automatischen Erzeugung einer im Prinzip unbegrenzten Menge überladener Funktionen.

Die Templateparameter müssen sich auf den Typ der Funktionsparameter auswirken, wenn der Compiler automatisch auf den Typ der Parameter schließen soll. Deshalb ist folgender Programmteil falsch:

```
template <int n> double f(double)
```

```
{
  double array[n];
}
```

Hier kann der Compiler aus dem Aufruf von `f()` keinen Rückschluss auf `n` ziehen. Es muss pro Templatefunktion genau eine Definition geben.

8.2 Klassentemplates

Man kann auch für Klassendefinitionen Templates schreiben. Die Schablonen heißen **Klassentemplates**, die generierten Definitionen heißen **Templateklassen**. Die Syntax der Deklaration entspricht der von Funktionentemplates.

```
template <typename T> class Dog
 {
 public:
     T GetAge();
     void SetAge(T age);
 private:
     T itsAge;
 };
```

Eine Templateklasse wird aus einem Klassentemplate gebildet, indem man die aktuellen Parameter in spitzen Klammern hinter dem Templatenamen angibt.

```
Dog<int> Nero;
```

Bei der ersten Verwendung erzeugt der Compiler die Definition aus dem Template `Dog`, indem er dort überall `T` durch `int` ersetzt. Memberfunktionen einer Templateklasse, die außerhalb der Klassendefinition definiert werden, müssen ebenfalls über `template<...>` angesprochen werden. Hier ein kompletttes Beispiel `template_klassen.cc`:

```
#include <iostream>

  using namespace std;

template <typename T> class Dog
 {
 public:
     T GetAge {return itsAge;}; //inline
     void SetAge(T age); //nicht inline
 private:
     T itsAge;
 };
```

```
template <typename T> void Dog<T>::SetAge(T age)
{
    itsAge = age;
}

int main()
{
    Dog<int> Nero;
    Nero.SetAge(7);
    cout << "Nero ist " << Nero.GetAge() << " Jahre alt." << endl;
    return 0;
}
```

Bei Klassentemplates sind auch Parameter erlaubt, die keine Typen sind:

```
template <int n, typename T> class X
{
  private:
  T array[n];
}
```

Nun erzeugt

`X<2, int> test`

eine Klasse mit privater Komponente `int array[2]`. Man kann Templateklassen auch benennen, und zwar mit dem Schlüsselwort `typedef`.

```
typedef X<50, int> X50i;
X50i test;
```

Nun muss man die Parameter nicht mehr ständig angeben.

Kapitel 9

Exceptions

Mit **Exceptions** lassen sich Ausnahmesituationen sauber behandeln. Eine Exception ist ein Objekt, das aus dem Codebereich, in dem ein Problem auftritt, an den Codebereich übergeben wird, welcher das Problem behandeln soll. Der Programmierer legt selbst fest, was eine Ausnahmesituation ist. Allgemein ist dies ein Zustand, der im Programmablauf nicht auftreten sollte. Für einen Block mit Anweisungen (`try`-Block) legt der Programmierer mittels beliebig vieler `catch`-Blöcke fest, welche Reaktion auf welche Ausnahme zu erfolgen hat. Die Ausnahme ist ein Objekt beliebigen Typs.

```
try
{
   //beliebige Anweisung
}
catch (Typ_1 var_1)
{
   //Behandlung der Ausnahmen vom Typ_1
}
catch (Typ_2 var_2)
{
   //Behandlung der Ausnahmen vom Typ_2
}
```

Wenn im `try`-Block mittels

```
throw obj;
```

eine Ausnahme ausgeworfen wird, geschieht dabei folgendes:

1. Der `try`-Block wird verlassen.
2. Die angegebenen `catch`-Blöcke werden der Reihe nach geprüft. Falls das Objekt `obj` der Deklaration `Typ_i` entspricht, wird `var_i` mit `obj` initialisiert. Die

Anweisungen im entsprechenden Block werden ausgeführt. Anschließend fährt das Programm hinter dem letzten `catch`-Block fort.

3. Passt `obj` zu keinem `catch`-Typ, wird die Ausnahme nicht abgefangen. Dafür kann meinen Default-Zweig vorsehen.

Die Besetzung von `var_i` geschieht so, als sei der `catch`-Kopf eine Funktionsdeklaration mit dem formalen Parameter `var_i` und die `throw`-Anweisung ein Funktionsaufruf mit dem Parameter `obj`. Einige Beispiele (es handelt sich hierbei nicht um eine echte Datumsklasse, sondern nur um ein einfaches Beispiel zu Demonstration von Exceptions):

```
 #include<iostream>

  using namespace std;

class XTag {};
class XMonat {};

class Datum
{
  public:
   Datum(int ptag, int pmonat, int pjahr):
    tag(ptag), monat(pmonat), jahr(pjahr)
    {
      if (monat > 12)
      throw XMonat();

      if (tag > 31)
      throw XTag();
    }
    int tag, monat, jahr;
};

int main()
{
  try
  {
    Datum d1(24,22,1994), d2(24,11,1984);
    cout << d1.tag << "." << d1.monat << "." << d1.jahr << endl;
    cout << d2.tag << "." << d2.monat << "." << d2.jahr << endl;
  }

  catch (const XTag)
```

```
  {
    cerr << "Tag falsch!" << endl;
  }
  catch (const XMonat)
  {
    cerr << "Monat falsch!" << endl;
  }
  Datum d3(02,05,1994);
  cout << d3.tag << "." << d3.monat << "." << d3.jahr << endl;
  return 0;
};
```

Ausgabe:

```
Monat falsch!
2.5.1944
```

Ändert man das erste Datum auf `(24,12,1994)`, so lautet die Ausgabe:

```
24.12.1994
24.11.1984
2.5.1944
```

Man erreicht ein unspezifisches Ausgeben aller Ausnahmen durch:

`catch (...)`

In obigem Beispiel müsste der `catch`-Block folgendermaßen verändert werden.

```
   catch (...)
   {
     cerr << "Irgendein Fehler!" << endl;
   }
```

Es ist auch möglich, mit der Exception-Behandlung zu erfahren, welcher Wert falsch ist. Im folgenden Programm wird der falsche Wert mit ausgegeben. Außerdem werden die Klassen `XTAg` und `XMonat` von einer gemeinsamen Basisklasse abgeleitet.

```
 #include<iostream>

  using namespace std;

class XDatum
{
  public:
```

```
  XDatum(int tm): xtm(tm) {}
  int wert() {return xtm;}
  private:
  int xtm;
};

class XTag:public XDatum
{
  public:
  XTag(int t): XDatum(t) {}
};

class XMonat:public XDatum
{
  public:
  XMonat(int t): XDatum(t) {}
};
class Datum
{
  public:
    Datum(int ptag, int pmonat, int pjahr):
      tag(ptag), monat(pmonat), jahr(pjahr)
    {
      if (monat > 12)
      throw XMonat(monat);

      if (tag > 31)
      throw XTag(tag);
    }
   int tag, monat, jahr;
};

int main()
{
  try
  {
    Datum d1(12,14,1994), d2(02,05,1984);
    cout << d1.tag << "." << d1.monat << "." << d1.jahr << endl;
    cout << d2.tag << "." << d2.monat << "." << d2.jahr << endl;
  }

  catch ( XTag &xt)
  {
    cerr << "Tag falsch: " << xt.wert() << endl;
```

```
  }
  catch ( XMonat &xm)
  {
    cerr << "Monat falsch: " << xm.wert() << endl;
  }
  catch ( XDatum&)
  {
    cerr << "Datum falsch! " << endl;
  }
  return 0;
};
```

Ausgabe:

```
Monat falsch: 14
```

Der dritte `catch`-Zweig wird als Default-Zweig verwendet, er fängt alle `XDatum`-Exceptions ab, die nicht vorher abgefangen wurden.

Exceptions sind normale Objekte, daher stehen auch Klasseninstumente wie multiple Vererbung und virtuelle Funktionen zur Verfügung. Man kann `try`-Blöcke auch ineinander verschachteln, sie werden dann von innen nach außen abgehandelt.

Falls für eine Exception kein passender `catch`-Block existiert, wird automatisch die Funktion

```
void terminate()
```

aufgerufen, die das Programm beendet. Man kann die Funktion, die an dieser Stelle auftreten soll, auch selbst definieren. Falls die benutzerdefinierte Funktion `void f()` aufgerufen werden soll, meldet man sie folgendermaßen an:

```
set_terminate(f);
```

Die Funktion `f` muss das Programm beenden.

Falls eine nicht zulässige Exception ausgeworfen wird, wird

```
void unexpected();
```

aufgerufen. Diese Funktion ruft normalerweise `terminate()` auf. Auch hier kannn man eine eigene Funktion benutzen:

```
set_unexpected(f);
```

Kapitel 10

Die Standard Template Library

10.1 Container und Iteratoren

Unter Containern werden ganz allgemein Klassen verstanden, die als Behälter für Datenelemente fungieren, zum Beispiel Listen, Arrays und Vektoren. Die Bibliothek stellt hierfür ein System von Templates zur Verfügung. Alle Container der STL besitzen das gleiche Interface, das macht es einfach, bei Bedarf einen verwendeten Containertyp gegen einen anderen auszutauschen.

Iteratoren sind abstrakte Zeiger (sie lassen sich zum Beispiel inkrementieren und dereferenzieren), mit deren Hilfe man auf die Elemente eines Containers zugreifen kann. Iteratoren sind nur über ihre generellen Eigenschaften definiert. Durch den Zugriff über Iteratoren muss man sich nicht mit den Details zum Datenzugriff beschäftigen. Zu jeder Kombinaton von Container- und Elementtyp gehört ein spezieller Iteratortyp. Iteratoren sind wie die Container Templates und verfügen über eine einheitliche Schnittstelle. Hinter einem speziellen Iteratortyp verbirgt sich je nach Container ein Klassenobjekt oder ein einfacher Zeiger.

10.2 Die Standard Template Library

Die **Standard Template Library** ist eine Bibliothek mit Template-basierten Container-Klassen. Die STL enthält auch Routienen zum Sortieren und Suchen. Ein typisches Beispiel zur Benutzung der STL ist die Erstellung eines Vektors (`vector.cc`):

```
1:  #include <iostream>
2:  #include <vector>
3:  using namespace std;
4:
5:  int main()
6:  {
```

```
7:    vector<int> vec;
8:
9:    for (int i=0; i<9; i++)
10:   vec.push_back(i);
11:   for (vector<int>::iterator it=vec.begin(); it != vec.end(); ++it)
12:     cout << *it << endl;
13:
14:   return 0;
15: }
```

In Zeile zwei wird die in der STL enthaltene `Vector`klasse eingebunden. Die Elemente des Vektors werden in Zeile zehn mit den Werten von null bis acht besetzt. Zeile zwölf gibt die Elemente des Vektors aus. Die Dereferenzierung des Iterators erfolgt wieder mit dem `*` Operator. Der in der Templateklasse `vector<int>` enthaltene `iterator` ist ein `int`-Zeiger. Die Funktion `begin()` liefert einen Iterator, der auf das erste Element des Containers zeigt, `end()` liefert einen, der auf ein fiktives Element hinter dem letzten zeigt.

Der Elementtyp eines Containers ist im Prinzip beliebig, aber es müssen einige Eigenschaften erfüllt sein. Zum Beispiel müssen die Operatoren `<` und `==` definiert sein. Ein weiteres Beispiel zur Nutzung einer Templatefunktion der STL ist das Finden bestimmter Elemente:

```
vector<int>::iterator it = find(vec.begin(), vec.end(), 3);
```

Diese Funktion sucht im Vektor nach einem Element mit Wert drei und liefert einen Iterator zurück, der auf dieses Element zeigt. Falls ein solches Element nicht vorhanden ist, wird der Wert von `vec.end()` zurückgegeben.

Weitere Containertypen neben `vector` sind `list` (Liste), `queue` (Warteschlange), `deque` (double-ended queue) und `stack` . In einer Liste ist das Einfügen und Löschen von Elementen möglich. `queue` erlaubt das Einfügen vom Elementen an einem Ende und das Löschen von Elementen an der anderen Ende. `deque` erlaubt Einfügen und Löschen von Elementen an beiden Enden. Beim `stack` kann nur das Element, was als letztes hinzugefügt wurde (nur an einem Ende möglich), wieder gelöscht werden. Weitere Operationen für Container sind:

Elementzugriff:

- `[]` : ungeprüfter Indexzugriff (nicht für Listen)
- `at()` : geprüfter Indexzugriff (nicht für Listen)
- `front()` : Zugriff auf das erste Element

Iteratoren:

- `begin()` : zeigt auf das erste Element

- `end()` : zeigt hinter das letzte Element

Randelemente:

- `push_back(p)` : Element `p` am Ende anfügen
- `pop_back()` : letztes Element entfernen
- `push_front(p)` : Element `p` am Anfang einfügen (nur `list`)
- `pop_front()` : erstes Element entfernen (nur `list`)

Operationen zum Einfügen und Löschen von Elementen:

- `insert(p,x)` : fügt `x` vor `p` ein
- `insert(p,n,x)` : fügt `n` Kopien von `x` vor `p` ein
- `erase(p)` : löscht Element bei `p`
- `clear()` : löscht alle Elemente

Beispiel zur Benutzung:

```
vector<int> vec;
vec.erase(vec.begin()+3);
```

Als Konstruktor kann man `container(n)` verwenden, um einen Container mit n Default-Elementen zu erzeugen.

```
vector<int> vec(10);
```

10.2.1 Die Klasse `string`

Mit der Klasse `string` können allgemeine Strings bearbeitet werden. Eigentlich ist `string` keine einfache Klasse, sondern ein typedef für das Template `basic_string<char>`. Die Deklaration steht im Header `<string>`. Einige Operationen mit Strings:

- `a = b` : Der String `b` wird auf `a` kopiert.
- `a = "xyz"` : Die Zeichenfolge `xyz` wird auf `a` kopiert.
- `a.c_str()` : Anhängen von '\0' und Rückgabe als C-String.
- `a += b` : `b` wird an `a` angehängt.
- `a.find(b)` : Sucht `b` in `a` und gibt den Index zurück.

- `a.replace(i, n, b)` : Ersetzt n Zeichen ab i-tem durch b

Längenparameter haben im Allgemeinen den Typ `string::size_type`, der verwendet werden sollte, auch wenn der Typ bekannt ist. Das Programm `klasse_string.cc` zur Bearbeitung von strings

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

int main()
{
string s("Hallo");
  s+= " World";
s.replace (7,2,"e");                      // ersetze Zeichen 7-8 durch e
string::size_type index_d = s.find('d'); // suche d
s[index_d] = 't';                         // ersetze d durch t
s.insert(10,"!");                         // füge hinter 10-tem Zeichen ! ein
cout << s << endl;
return 0;
}
```

hat die Ausgabe

```
Hallo Welt!
```

10.3 Externe Bibliotheken

Es gibt nicht nut die Standard Template Library, sondern auch externe Bibliotheken, wie zum Beispiel BLAS (Basic Linear Algebra Subprograms) und LAPACK (Linear ALgebra PACKage). BLAS stellt einige Routinen, die in der Linearen Algebra genutzt werden wie Vektor- oder Matrixmultiplikationen zur Verfügung. LAPACK kann genutzt werden, um Systeme linearer Gleichungen, kleinste-Quadrate Probleme und Eigenwertprobleme zu lösen. Fast alle LAPACK Routinen benutzen BLAS.

Kapitel 11

Weiterführende Themen

11.1 Curiously Recurring Template Pattern

Die externe Bibliothek FLENS (Flexible Library for Eficient Numerical Solutions) realisiert ihre Klassen mit CRTP (Curiously Recurring Template Pattern). Dieses Muster ist sehr effizient und wird daher gerne verwendet. Es erlaubt statische Polymorphie ohne virtuelle Funktionen. Wir betrachten die Basisklasse `BasicClass` und deklarieren die abstrakte Methode `aMethod`, welche in der abgeleiteten Klasse `Implementation` implementiert wird.

```
template <typename Impl>
class BasicClass
{
  public:
     Impl &impl()
     {
       return static_cast<Impl &>(*this);
     }

     void aMethod()
     {
      //...
     }
};

class Implementation : public BasicClass<Implementation>
{
 public:

   void aMethod()
```

```
  {
    //..
  }
};
```

Der Punkt ist, dass die Basisklasse `BasicClass` den Typ der abgeleiteten Klasse als Parameter enthält. Deshalb kann die Basisklasse eine Methode `impl` bereitstellen, die statisch castet. Beim Aufruf der Methode `impl` wird `aMethod` aufgerufen, deren Implementation in der abgeleiteten Klasse statt findet. Diese Technik ist schneller als die Benutzung von virtuellen Funktionen.

11.2 Template Metaprogramming

Die Template Metaprogrammierung ist eine Programmtechnik, in der Templates benutzt werden, um während der Compilierung des Codes schon Funktionen zu berechnen. Ein Beispiel (`factorial.cc`):

```
1:  #include <iostream>
2:
3:  template <int N>
4:  struct Factorial
5: {
6:   static const int value = Factorial<N-1>::value * N;
7:  };
8:
9:  template <>
10: struct Factorial<1>
11: {
12:  static const int value = 1;
13: };
14:
15: int main()
16: {
17:   std::cout << Factorial<10>::value << std::endl;
18: }
```

In Zeile drei bis sieben wird ein Template definiert. Zeile neun leitet eine Template Spezialisierung ein. Diese wird als Abbruchbedingung für die Berechnung der Fakultät benutzt. Dieses Template wird in Zeile 17 mit der Zahl 10 verwendet. Der Compiler erzeugt die Definition aus dem Template beim compilieren. Das obige Programm berechnet die Fakultäten zur Compilierungszeit und benutzt die Werte, als wären es vordefinierte Konstanten.

Literaturverzeichnis

[1] P. Bastian. *Informatik 1, Programmieren und Softwaretechnik*,
Skript, Universität Heidelberg (2002)
http://hal.iwr.uni-heidelberg.de/lehre/inf1-ws02/download/inf1.pdf

[2] H. Burkhardt. *Kurzdokumentation zum Softwarepraktikum SS 2001*,
Skript, Albert-Ludwigs-Universität Freiburg (2001)
http://wwwmath.uni-muenster.de/num/Vorlesungen/Praktikum_WS07/Dokumente/doku.pdf

[3] B. Eggink. *C++ für C-Programmierer*,
RRZN/ Universität Hannover (2005)

[4] M. Lehn. *Flens, A Flexible Library for Efficient Numerical Solutions*,
Dissertation, Universität Ulm (2008)
http://vts.uni-ulm.de/docs/2008/6419/vts_6419_8661.pdf

[5] J. Liberty. *C++*,
Markt und Technik (2004)

[6] B. Stroustrup. *Die C++ Programmiersprache*,
Addison-Wesley (1998)

[7] S. Wiedenbauer, B. Haasdonk, C. Link. *Einführung in die wissenschaftliche C-Programmierung*