C vs. C++

Proseminar C - „Grundlagen und Konzepte“, 2013

Sebastian Meyer

Betreuer: Nathanael Hübbe

Hamburg, 26. Juni 2013

Inhaltsverzeichnis

1 Einführung

1.1 Was ist C++?

C++ ist eine objektorientierte Weiterentwicklung der Sprache C. Sie beinhaltet die Sprache C. Allerdings kann man nicht sagen, dass C ein komplettes Subset der Sprache C++ ist, denn es gibt Inkompatibilitäten. Leider wurden bei der Weiterentwicklung nicht nur Ergänzungen und Aufweichungen, sondern auch Verschärfungen und Änderungen vorgenommen. Das führt dazu, dass C-Code nicht immer vom C++-Compiler kompilierbar ist. Dennoch hat man sich bei der Entwicklung bemüht ziemlich nah an der Sprache C zu bleiben. So ist es zumindest möglich C-Code zu benutzen, indem man ihn nicht in einer '.cpp"-Datei, sondern in einer ".c"-Dateien ablegt. Des Weiteren bietet C++ auch eine neue Standardbibliothek, die allerdings die C Bibliothek einschließt. Es ist aber empfehlenswert vorzugsweise die C++-Bibliothek zu benutzen, wenn man in C++ programmiert.

1.2 Historische Einordnung

Vor der Entwicklung von C++ nahm C eine sehr besondere Rolle. Neben Assembler, nahm sie aufgrund der Maschinennähe und der Effizienz, die Rolle der Programmiersprache für Systemprogrammierung, Treiber, etc. ein. Der Entwickler der Sprache C++, Bjarne Stroustrup, beschäftigte sich zu der Zeit in seiner Doktorarbeit an der Cambridge University mit der Sprache Simula. Dies ist eine Sprache die, wie der Name schon sagt, für Erstellung von Simulationen und somit für große Projekte eignete. Allerdings ließ die Sprache nicht zu, hocheffizient zu programmieren. Die Sprache BCPL ließ dies zu, war aber wiederum nicht für große Projekte geeignet. Stroustrup versuchte die beiden Eigenschaften der Sprachen zu vereinen. Dies gelang ihm als er beim nordamerikanischen Telekommunikationsunternehmen AT&T versuchte, die bereits in UNIX-Kreisen etablierte Programmiersprache C, zu erweitern. Seit 1979 beschäftigte er sich mit der Entwicklung von "C with Classes". Er brachte auch einen Compiler heraus, der zunächst "C with Classes"-Code in C-Code übersetzte. 1983 wurde "C with Classes" schließlich in C++ umbenannt und im laufe der zeit immer wieder erweitert. Somit eignete sich jetzt nicht nur C, sondern auch C++ für System und Anwendungsprogrammierung. Für den Bereich der Anwendungssoftwareprogrammierung wurden später Sprachen wie Java und C# entwickelt, die nun parallel neben C++ existieren. Java und C++ boten allerdings eine größere Laufzeit und einen standardmäßigen Garbage Collector an. Das machte die Sprachen zunächst attraktiv. Es führte aber auch dazu, dass sich diese Sprachen nicht für Anwendungen eignen, in denen es auf Effizienz und Schnelligkeit ankommt. Aufgrund eines Trends hin zu Green-IT und Ressourcensparsamkeit rückte C++ deshalb in der letzten Zeit auch im Gebiet der Anwendungssoftware wieder in den Vordergrund.

2 Vergleich der Spracheigenschaften

2.1 Vergleich der Konzepte von C / C++

Auch wenn man die in den Sprachen umgesetzten Konzepte anschaut, kann man sehr gut sehen, dass C++ eine objektorientierte Weiterentwicklung von C ist. Die Konzepte der imperativen, strukturierten und prozeduralen Programmierung wurden in beiden Sprachen berücksichtigt. C++ erweitert C um die Konzepte der objektorientierten- und generischen Programmierung. Für die objektorientierte Programmierung werden in C++ mehrere Elemente angeboten, die denen aus Java teilweise sehr ähnlich sind. Um die generische Programmierung zu ermöglichen gibt es in C++ Templates, die auch noch thematisiert werden. Dank der C++-Standardbibliothek ist es auch möglich funktional in C++ zu programmieren. Allerdings bieten echte funktionale Programmiersprachen weitaus größere Möglichkeiten an. Da Objektorientierung viel mehr ein Gedankenkonstrukt als ein Sprachfeature ist, ist es auch in C möglich mithilfe von Konventionen objektorientiert zu programmieren. Mögliche Vorgehensweisen werden jeweils bei der Erläuterung der neuen Sprachelemente von C++ thematisiert.

2.2 Vergleich der Spracheigenschaften

C++ bietet als Weiterentwicklung nicht nur die aus C bekannten Structs und Zeiger an sondern viele weitere Konzepte, die das objektorientierte Programmieren ermöglichen. Ein Konzept sind Klassen. Klassen haben im Gegensatz zu Structs nicht nur Attribute sondern auch Funktionen und bilden somit eine Einheit aus Attributen und zugehörigen Funktionen. Von diesen Klassen lassen sich dann Objekte erzeugen. Um den Anspruch einer objektorientierten Weiterentwicklung gerecht zu werden bietet C++ noch Vererbung (genauer: Mehrfachvererbung) an. Auch die Überladung von Funktionen wird unterstützt. Das objektorientierte Grundgerüst ermöglicht auch eine neue Art der Fehlerbehandlung mit "Fehlerobjekten" welches unter das Stichwort "Exceptions" fällt. C++ hat zudem eine neue Standardbibliothek welche die C-Standardbibliothek mit einschließt. Allerdings ist es empfehlenswert in C++ auch die C++-Standardbibliothek zu benutzen. Bei der Entwicklung von C++ wurde darauf geachtet, dass die Sprache weiterhin dem Grundsatz "You only pay for what you use" gehorcht. Wen man also dasselbe tut hat man in C denselben Overhead wie in C++. Allerdings sind konkrete Implementierungen in C++ meist nicht so sparsam. Außerdem verschleiert die Sprache C++ mehr als C. So kann es passieren, dass für die gleiche Tätigkeit Sachen mit eingebunden werden, die man eigentlich gar nicht braucht wodurch man in C++ doch einen größeren Overhead verursachen kann. Die nicht so sparsamen Implementierungen haben auch zur Folge, dass die Programme möglicherweise nicht mehr so schnell sind wie es vergleichbare C-Programmen wären. Als objektorientierter Programmierer neigt man dazu viele Objekte zu erzeugen und viele Funktionsaufrufe zu verursachen. Das verlangsamt das System minimal. In einigen Anwendungsbereichen, kann das aber entscheidend sein. Betrachtet man die Maschinennähe der Sprachen, so kann man sagen, dass beide Sprachen doch ziemlich maschinennah sind. Allerdings weiß man in C an einigen Punkten eher, was

gerade konkret gemacht wird. Z.B. ist man bei der Speicherreservierung mit ”malloc()” näher am System als bei der Objekterzeugung mit ”new”. Beide Sprachen haben nur eine minimale Laufzeit, die durch die Standardbibliothek definiert ist und bieten im Gegensatz zu Sprachen wie Java oder C# keinen standardmäßigen Garbage Collector an. Aufgrund dessen sind C- und C++-Programme allerdings sehr viel performanter. Ein Nachteil ist aber, dass C und C++ viele Fehler gar nicht oder erst sehr spät zur Laufzeit unterbinden und dass man sich manuell um die Speicherfreigabe kümmern muss. C++ besitzt zusätzlich noch Mechanismen, die zuverlässiger sind als malloc() und die Zeigearithmetik. Sofern man in C++ mit new und delete() arbeitet, kann man nicht so einfach eine ungültige Referenz erhalten. Auch die Zugriffsmodifizierer (public, private etc.) für Funktionen und Attribute schaffen zusätzliche Sicherheit. Da C++ bei der Verwendung der C++ Konzepte nicht ganz so maschinennah wie C ist, ist in C auch eine bessere Optimierung des Codes möglich. Dagegen gibt es für objektorientierte Sprachen viele Konzepte um Fachlichkeit zu modellieren. Das trägt dazu bei, dass man sich auch gut mit Kunden über die Software unterhalten kann, die weniger von den technischen Gegebenheiten verstehen. Hält man sich an die Trennung von Fachlichkeit und Technik sowie an lose Kopplung von Systemkomponenten sind nach der Meinung von vielen objektorientierten Entwicklern solche Systeme besser zu warten als nicht objektorientierte Systeme. Allerdings kostet objektorientierte Programmierung, bei der man sich streng an Entwurfsmuster hält auch sehr viel Zeit. Dagegen gibt es allerdings auch in C++ Elemente, wie z.B. Templates oder Mehrfachvererbung, die die Wartbarkeit reduzieren können. Die Übersichtlichkeit eines Softwaresystems ist also in beiden Systemen gefährdet. Man muss in beiden Fällen sehr sorgfältig vorgehen, wenn man eine Struktur in die Systeme hineinbringen möchte.

3 Neue Sprachelemente in C++

3.1 Ein- und Ausgabe

C++ gibt einem eine neue Möglichkeit Text oder primitive Datentypen auf der Konsole auszugeben. Dazu hat man die Bitshift-Operatoren überladen (was ja in C++ jetzt möglich ist). Das Einlesen übernimmt nun der Operator », das Auslesen «. Den Ein- und Ausgabeort legt man mit cin, cout, cerr und clog fest. So kann man mit cin und cout Eingaben von der Konsole einlesen und Text ausgeben. Cerr und clog sind für die Fehlerausgabe zuständig. Einen Zeilenumbruch erreicht man mit endl. Dieser Wert entspricht dem \n und ist auch in der Standardbibliothek so festgelegt. cout zusammen mit dem «- Operator entspricht also printf(). Mit den Operatoren kann man mehrere auszugebende Daten verketten. So wurde z.B. unten im Codebeispiel die Zeichenkette ”Du bist ” mit der Variablen jahre_alt verkettet. Ein deutlicher Vorteil gegenüber printf() ist die automatische Formatwahl. So ist es nicht mehr nötig mit Ausdrücken wie %i zu spezifizieren, um welchen Datentyp es sich bei den angehängten Werten handelt.

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

int main(int argc, char* argv[])
{
    int jahre_alt;
    cout << "Wie alt?" << endl;
    cin >> jahre_alt;
    cout << "Du bist " << jahre_alt << " Jahre alt." << endl;
    return 0;
}
```

3.2 Umgang mit Variablen

In älteren C Standards war es immer nötig, dass die Variablendeklarationen am Anfang des Blocks stehen. In C++ kann man nun Variablendeklarationen und Anweisungen mischen. Bei einigen Compilern kann man das bis heute beobachten, obwohl in C99 diese Aufweichung aus C++ übernommen wurde. Nutzt man den Microsoft-Compiler aus Visual Studio 2008 für C, so erhält man einen Übersetzungsfehler, wenn man Variablendeklarationen und Anweisungen mischt. Benutzt man nun den Microsoft C++ Compiler so erhält man diesen Fehler nicht. Des weiteren gibt es in C++ nun den neuen Operator "::", mit dem man auf globale Variablen zugreifen kann. Wurde eine globale Variable durch eine lokale Variable mit dem gleichen Namen verdeckt, kann man diesen Operator vor den Variablennamen schreiben und so auf die globale Variable zugreifen.

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

int a = 5;

int main(int argc, char* argv[])
{
    int a = 7;
    cout << a << endl; //Ausgabe a lokal: 7
    int b = 6; //In C nicht moeglich

    cout << a << endl; //Ausgabe a lokal: 7
    cout << ::a << endl; //Ausgabe a global: 5
}
```

3.3 Defaultparameter

In C++ ist es möglich Funktionsparameter zu definieren, die einen vordefinierten Standardwert annehmen, wenn dieser beim Aufruf nicht explizit angegeben wird. Dies kann z.B. bei Rekursionen hilfreich sein. Dort ist es manchmal notwendig, dass man zu Beginn einen Standardwert hat, der beim rekursiven Aufruf allerdings verändert wird. Defaultparameter müssen immer als letztes im Funktionskopf angegeben werden. Es ist also nicht möglich Defaultparameter und normale zu mischen. Wäre dies möglich, wäre zur Laufzeit nicht erkennbar, welche Parameter als Defaultparameter interpretiert werden sollen und welche nicht. Ein weiterer Anwendungsfall ist die Erweiterung der Funktionalität, die durch einen zusätzlichen Parameter gesteuert wird.

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  int potenzen(int a, int potenz=2)
6  {
7      return pow((float)a, potenz);
8  }
9
10
11 int main(int argc, char* argv[])
12 {
13     cout << potenzen(3) << endl; //Ergebnis: 9
14     cout << potenzen(3, 3) << endl; //Ergebnis: 27
15     return 0;
16 }
```

3.4 Überladung

In C++ ist Überladung möglich. Es können also mehrere Funktionen mit demselben Namen, aber anderen Parametern definiert werden. Welche Funktion verwendet wird, wird beim Funktionsaufruf anhand der übergebenen Parameter entschieden. Im Codebeispiel ist die Funktion potenzen() überladen. In C ist dies so nicht möglich. Man kann das Konzept allerdings anhand von Namenskonventionen eigenständig umsetzen. So kann man z.B. mehrere Methoden mit demselben Namen definieren, wenn man an diesen noch die Parameterdefinition anhängt, für die diese Implementation gültig ist. Also würde beispielsweise potenzen_i() für die Funktion mit einem Integer-Parameter und potenzen_ii() für die Funktion mit zwei Integer-Parametern stehen. In C++ kodiert der Compiler diesen Namensanhang selbst um die Überladung zu realisieren. Das führt dazu, dass die Namen in den Bibliotheken anders sind als im Quellcode. Da die Namen in C gleich bleiben, eignet sich C besser zum Entwickeln von Bibliotheken, sofern man diese

nicht nur in C++ benutzen will.

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

int potenzen(int a, int potenz)
{
    return pow((float)a, potenz);
}

int potenzen(int a)
{
    return pow((float)a, 2);
}

int main(int argc, char* argv[])
{
    cout << potenzen(3) << endl; //Ergebnis: 9
    cout << potenzen(3, 3) << endl; //Ergebnis: 27
    return 0;
}
```

3.5 Klassen

Klassen unterscheiden sich von den aus C bekannten Structs davon, dass sie nicht nur Attribute, sondern auch Funktionen enthalten. So wird eine Einheit aus Daten und Operationen geschaffen. Aber natürlich ist das nicht das einzige Merkmal von Klassen. Zwischen Klassen lassen sich wie in Java Beziehungen, wie z.B. die Vererbungsbeziehung oder die uses-Beziehung modellieren. Der Code für das Erstellen von Klassen sieht in C++ allerdings anders aus als in Java. Das liegt an der Trennung von Deklaration und Implementation, die aus C übernommen wurde. So wird die eigentliche Klassendeklaration (inklusive der Deklaration ihrer Attribute und Funktionen) in der Headerdatei vorgenommen. Die Implementation der einzelnen Funktionen wird üblicherweise in der Quelldatei vorgenommen. Dazu wird in C++ der Operator :: benutzt. Mit diesem Operator wird bei jeder Funktionsdeklaration die zugehörige Klasse angegeben, die bereits in der Headerdatei deklariert wurde. Zudem lassen sich mehrere Klassen in eine Datei schreiben. In Java hat man üblicherweise für jede Klasse eine eigene Datei. Die Zugriffsmodifizierer sind in C++ ebenfalls neu. Public, private und protected haben dieselbe Bedeutung wie in Java. Ein Unterschied ist allerdings, dass man nicht zu jeder Deklaration den Zugriffsmodifizierer einzeln angeben muss. Stattdessen sind sie für alle folgenden Deklarationen gültig, sofern kein neuer Zugriffsmodifizierer angegeben wird. C++-Klassen haben immer einen Konstruktor und einen Destruktor. Sofern dieser nicht im Quellcode

angegeben wird, wird er automatisch beim Kompilieren erzeugt. Der Konstruktor ist schon aus Java bekannt. Der Destruktor wird in C++ ebenfalls bei der dynamischen Speicherverwaltung gebraucht. Nähere Informationen dazu gibt es im Kapitel "Objekte und dynamische Speicherverwaltung". In C lassen sich Klassen auch mithilfe von Structs und Funktionen realisieren indem man z.B. vor den Funktionsnamen immer noch den Namen des zugehörigen Structs schreibt. Auch Vererbung lässt sich realisieren, indem man für die spezialisierten Structs und Funktionen geeignete Namensanhänge als Konvention festlegt.

Konto.h enthält nur Deklarationen:

```
1  class Bankkonto
2  {
3
4  protected:
5      int _kontostand;
6
7  public:
8      Bankkonto(); //Konstruktor
9      ~Bankkonto(); //Destruktor
10
11     int getKontostand();
12     int zahleEin(int wert);
13 };
```

Konto.cpp enthält die Implementierungen der Funktionen:

```
#include "Konto.h"

Bankkonto::Bankkonto() //Konstruktor
{
    _kontostand = 0;
}

Bankkonto::~Bankkonto() //Destruktor
{
}

int Bankkonto::getKontostand()
{
    return _kontostand;
}

int Bankkonto::zahleEin(int wert)
{
    if(wert >=0)
    {
    _kontostand = _kontostand + wert;
    return 0;
    }
    return 1;
}
```

3.6 Objekte und dynamische Speicherverwaltung

Die aus C bekannten Funktionen malloc() und free() sind in C++ zwar noch benutzbar, aber nicht mehr nötig. Dort kann man Objekte einer Klasse mit new auf dem Heap ablegen. Mit delete() kann man den Speicher wieder freigeben. Dies muss man in C++ aufgrund des fehlenden Garbage Collector manuell machen. Um ein zu Objekt erzeugen muss ein Pointer vom Typ des zu erzeugenden Objekts deklariert werden. Diesem Pointer kann dann mit ”new” und einem folgenden Konstruktoraufruf eine gültige Speicheradresse zugewiesen werden. Der Konstruktor sorgt wie in Java für die Initialisierung des Objekts. In C++ gibt es aber auch einen Destruktor. Der Destruktor wird aufgerufen, wenn man ein Objekt. mit delete() löscht. So können auch Objekte, die durch das Objekt erzeugt wurden, gelöscht werden. Diese Realisierung, die ohne Garbage Collector auskommt, ist in Vergleich zu anderen objektorientierten Sprachen wie Java oder C# sehr effizient. Nach dem Benutzen von delete() ist es wie in C sinnvoll alle Pointer auf NULL zu setzten. Es ist auch möglich ein Objekt auf dem Stack zu erzeugen. Dazu deklariert man keinen Pointer, sondern direkt das gewünschte Objekt und lässt new beim Konstruktoraufruf einfach weg. Das hat den Vorteil, dass das Objekt beim Verlassen der Routine automatisch gelöscht

wird. Allerdings kann das Objekt dann nicht von mehreren Objekten gemeinsam benutzt werden. Es ist also nicht erlaubt eine Referenz zu erzeugen und diese aus der Funktion hinauszureichen. Ein Nachteil von ”new” ist, dass es eine schlechtere Performance hat als malloc(). Wenn man hocheffiziente Programme schreiben möchte, könnte es also doch sinnvoll sein malloc() auch in C++ zu benutzen. Alternativ kann man in diesem Fall natürlich gleich C verwenden.

```
#include <iostream>
#include "Konto.h"
using namespace std;

int main(int argc, char* argv[])
{
    Bankkonto* konto1 = new Bankkonto(); //Objekt auf dem Heap
    konto1->zahleEin(2);
    cout << konto1->getKontostand() << endl;
    delete(konto1); //Destruktoraufruf und Speicherfreigabe
    konto1 = NULL;

    Bankkonto konto2 = Bankkonto(); //Objekt auf dem Stack
    konto2.zahleEin(10);
    cout << konto2.getKontostand() << endl;

    return 0;
}
```

3.7 Vererbung

Vererbung lässt sind in C++ realisieren, indem man ”:” hinter den Klassennamen schreibt und dort gefolgt von einem Zugriffsmodifizierer den Klassennamen der gewünschten Superklasse angibt. Der Zugriffsmodifizierer sorgt dafür, dass die schwächeren Zugriffsmodifizierer aus der Superklasse in dieser Klasse auf den angegebenen Zugriffsmodifizierer hochgesetzt werden. Mit ”schwächer” ist gemeint, dass mehr Zugriffe zugelassen werden. Also ist z.B. public schwächer als private. Ist ein Zugriffsmodifizierer bereits in der Superklasse stärker, verändert sich nichts. Wenn man bei der Vererbung keinen verändernden Zugriffsmodifizierer angibt wird standardmäßig ”private” angenommen. In vielen Fällen ist es aber sinnvoll nichts zu verändern, also ”public” zu benutzen. Da in Java bei der Vererbung auch nichts an den Zugriffsmodifizierern geändert wird, ist es für Einsteiger erstmal sinnvoll immer ”public” zu benutzen. In C++ ist nicht nur die einfache Vererbung, sondern auch die Mehrfachvererbung möglich. Man kann also von mehreren Klassen erben, indem man die gewünschten Superklassen wie im Beispiel mit einem Komma getrennt angibt. Mehrfachvererbung ist ein mächtiges Werkzeug. Z.B. lassen sich damit sehr gut Code-Doppelungen vermeiden. Aber manchmal ist es auch

einfach sinnvoll, wenn eine Klasse Eigenschaften von verschiedenen Klassen übernimmt. Allerdings kann Mehrfachvererbung unangenehme Folgen haben und den Code unübersichtlich machen. Probleme treten hauptsächlich beim Aufteilen und anschließenden Verschmelzen auf, insbesondere, wenn es sich bei der Vererbungshierarchie nicht mehr um eine Baumstruktur handelt. Ein bekanntes Problem ist z.B. das Diamond-Problem. In C++ kann man sogar selbst festlegen, wie sich das System in diesem Fall verhalten soll. Für weitere Informationen dazu möchte ich auf ergänzende Literatur verweisen. Um zu wissen, was die Mehrfachvererbung bewirkt, muss man sich gut mit Mehrfachvererbung auskennen. Oft sind Fehler auch nicht sofort ersichtlich.

Vererbung:

```
1 class Girokonto :public Bankkonto
2 {
3     //...
4 };
```

Mehrfachvererbung:

```
1 class Girokonto :public Bankkonto, public Finanzbehaelter
2 {
3     //...
4 };
```

Konten.h enthält eine Deklaration der erbenden Klasse Girokonto:

```
#include "Konto.h"

class Girokonto :public Bankkonto //Standardwert: private
{
protected:
    int _dispolimit;

public:
    Girokonto();
    int zahleAus(int wert);
};
```

Konten.cpp enthält die Implementierung der erbenden spezialisierten Klasse Girokonto: (Das Attribut _kontostand und die Funktion zahleEin() werden geerbt)

```
#include "Konten.h"

Girokonto::Girokonto()
{
    this;
    _dispolimit = 100;
}

int Girokonto::zahleAus(int wert)
{
    if((_kontostand - wert) >= (0 - _dispolimit))
    {
    _kontostand = _kontostand - wert;
    return 0;
    }
    return 1;
}
```

3.8 Virtuelle Funktionen

Mit virtuellen Funktionen kann man erreichen, dass die spezielle Implementation einer erbenden Klasse benutzt wird, auch wenn man einen Pointer vom Typ der Superklasse benutzt. Dazu muss die Funktion in der Superklasse als ”virtual” deklariert werden. Im Beispiel (siehe unten) wird in der main()-Funktion ein Girokonto-Objekt erzeugt. Obwohl man nur über einen Bankkonto-Pointer darauf zugreifen kann, wird die spezielle Implementation benutzt und somit der String ”Girokonto” ausgegeben. Wäre die Funktion keine virtuelle Funktion, würde an der Stelle die Ausgabe des Strings ”Bankkonto” erfolgen.

Konto.h enthält die Deklaration einer virtuellen Funktion:

```
using namespace std;

class Bankkonto
{
public:
    virtual string getKontoTyp(); //Virtuelle Funktion
};
```

Konto.cpp

```
#include <string>
#include "Konto.h"
using namespace std;

string Bankkonto::getKontoTyp()
{
    return "Bankkonto";
}
```

Konten.h

```
#include "Konto.h"
using namespace std;

class Girokonto :public Bankkonto
{
public:
    string getKontoTyp();
};
```

Konten.cpp

```
#include <string>
#include "Konten.h"
using namespace std;

string Girokonto::getKontoTyp()
{
    return "Girokonto";
}
```

Main.cpp

```
#include <string>
#include <iostream>
#include "Konten.h"
using namespace std;

int main(int argc, char* argv[])
{
    Bankkonto* konto1 = new Bankkonto();
    cout << konto1->getKontoTyp() << endl; //Ausgabe: Bankkonto

    Bankkonto* konto2 = new Girokonto(); //Ausgabe: Girokonto
    cout << konto2->getKontoTyp() << endl;

return 0;
}
```

3.9 Abstrakte Klassen

Setzt man hinter die Deklaration einer virtuellen Funktion noch ”= 0” muss die Klasse die Funktion nicht implementieren. Man hat dann eine abstrakte Klasse, von der sich keine Objekte erzeugen lassen. Diese eignet sich z.B. für Code-Injektion. Man kann also beispielsweise einen abstrakten Algorithmus implementieren, wobei die vom Algorithmus benutzten abstrakten Funktionen erst in der erbenden Klasse implementiert werden müssen. Da es in C++ keine Interfaces gibt, kann man bei Bedarf einfach das stärkere Element, also eine abstrakte Klasse benutzen.

Konto.h

```
using namespace std;

class Bankkonto
{
public:
    virtual string getKontoTyp() = 0;
};
```

3.10 Templates

Templates kann man sozusagen als Weiterentwicklung der Makros aus C bezeichnen. Sie funktionieren ebenfalls mit Ersetzen und sind Vorlagen für Funktionen und Klassen. Es gibt also Funktions- und Klassentemplates. Sie eignen sich, wenn Funktionen sich z.B. fast nur in den Parametertypen unterscheiden. Dann ist eine allgemeine Implementation möglich. Man nennt das generische Programmierung. So kann man beispielsweise eine

allgemeine Implementation für das Addieren von verschiedenen Datentypen schreiben. Wie die konkreten Parameter aussehen, wird beim Funktionstemplate erst beim Aufruf der Funktion entschieden. Man kann dann im Template Platzhalter für die spontan festgelegten Typen definieren (in diesem Beispiel: T) und so eine allgemeine Implementation für alle Typen schreiben. Templates sind ein mächtiges Tool, aber sie sind mit Vorsicht zu genießen. Es ist nämlich nicht gesagt, dass der Aufruf einer Funktion mit jeder Kombination von Parametern Sinn macht. Zudem können die Platzhalter so benutzt werden, dass nicht jeder beliebige Aufruf möglich ist. Man kann also schnell einen Aufruf tätigen, der eigentlich gar nicht zulässig ist. Demnach kann es beim Benutzen von Templates oft zu Fehlern kommen. Es ist also fraglich, ob man Templates wirklich benutzen sollte.

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

template <typename T> //Allgemeines Template
T addiere(T obj1, T obj2)
{
    return obj1 + obj2;
}

template <> //Spezialisierung
string addiere(string str1, string str2)
{
    return str1 + " " + str2;
}

int main(int argc, char* argv[])
{
    string a = "Hallo";
    string b = "Welt";
    cout << addiere(2, 3) << endl; //Ergebnis: 5
    cout << addiere(2.1, 2.1) << endl; //Ergebnis: 4.2
    cout << addiere(a, b) << endl; //Ergebnis: Hallo Welt
    //cout << addiere(2.3, 3) << endl; --> ist nicht moeglich
    return 0;
}
```

3.11 Namespaces

Mit Namespaces lassen sich globale Namen für Bereiche festlegen. Diese Bereiche sind allgemeingültig und können viele Dateien umfassen. Ein Beispiel dafür ist die C++-

Standardbibliothek. Die Standardbibliothek von C++ befindet sich im Namespace ”std”. Um auf Funktionen oder Variablen zugreifen zu können, muss man also ”std” und den Operator ”::” vor den Funktions- oder Variablennamen schreiben. Alternativ kann man am Anfang des Quelltextes mit dem Schlüsselwort ”using” und den Namen des Namespaces dafür sorgen, dass man direkt auf die Funktionen und Variablen zugreifen kann. Durch ”using” wird aber nichts importiert. Dies geschieht weiterhin durch ”#include”. Es verkürzt sozusagen nur den längeren Namen. Das Schlüsselwort ”using” ist vergleichbar mit dem Schlüsselwort ”import” aus Java. Man muss also immer erst die Headerdateien einbinden, in denen der Namespace vorhanden ist. Mithilfe von Namespaces kann man zusammengehörige Bereiche besser abgrenzen.

Namespace festlegen:

```
1 namespace bank
2 {
3     int a;
4 }
```

Auf Namespace zugreifen:

```
1 #include "Bank.h"
2 using namespace bank;
3
4 int main(int argc, char* argv[])
5 {
6     a = 0;
7     return 0;
8 }
```

oder

```
1 #include "Bank.h"
2
3 int main(int argc, char* argv[])
4 {
5     bank::a = 0;
6     return 0;
7 }
```

3.12 Neue C++ Standardbibliothek

C++ bietet eine neue C++ Standardbibliothek, die auch einige Elemente von C einschließt. Einige Header der C++ Standardbibliothek sind ”iostream” (Ein- und Ausgabe), ”string” (komfortabler C++ String), ”list”, ”set”, ”stack”, ”vector”, ”functional” (funktionale Programmierung). In C++ haben die Haederdateien der Standardbibliothek im Gegensatz zu denen aus C keine ”.h”-Endung. Außerdem wurden die Pakete der

C-Bibliothek übernommen. Sie sind weiterhin unter ".h"-Endung zu erreichen. Es gibt aber auch für die C-Standardlibrary-Dateien jeweils neue Versionen ohne ".h". Bei diesen Paketen wurde allerdings an den ursprünglichen Namen (aus C) der Buchstabe "c" vorne angehängt. Benutzt man die neue Version ohne Header, so sind die Elemente ebenfalls im Namespace "std" zu finden, ansonsten nicht. Bei neueren Versionen, wird aber die zugehörige ".h"-Datei automatisch eingebunden, wenn kein Namespace festgelegt wurde. Einbindung einer Headerdatei aus der C++ Standardbibliothek:

```
#include <paket-headerdatei>
using namespace std;
```

3.13 Vom C++-Compiler nicht kompilierbarer C-Code

Leider handelt es sich bei C++ nicht um eine reine Weiterentwicklung von C. Es werden also nicht nur Regeln aufgeweicht und ergänzt, sondern auch verschärft. Somit gibt es auch C-Code, der sich nicht von einem C++-Compiler kompilieren lässt. Eine Einschränkung in C++ ist, dass im Allgemeinem oft explizite Typumwandlung nötig ist, obwohl das in C an diesen Stellen nicht nötig war. Will man z.B. einen void-Pointer durch eine Zuweisung implizit in einen Pointer eines anderen Typs umwandeln, gibt der Compiler in C++ einen Fehler aus. Man muss hier explizit eine Typumwandlung durchführen. So ein Fall tritt z.B. beim Benutzen von malloc() auf, da malloc() einen void-Pointer zurückgibt, der dann in C üblicherweise in einen speziellen Pointer umgewandelt wird:

```
#include <stdlib.h>

int main(int argc, char* argv[])
{
    int* a1 = malloc(sizeof(int)); //gibt void Pointer zurueck
    int* a2 = (int*) malloc(sizeof(int)); //Gueltiger C++ Code
    return 0;
}
```

Zudem war es in C möglich main() rekursiv aufzurufen, was jetzt in C++ verboten ist. Durch die Einführung von neuen Schlüsselwörtern in C können diese reservierte Namen nicht mehr als Bezeichner verwendet werden. So ein reservierter Name ist z.B. "template" (denn Templates wurden ja in C++ eingeführt) In C++ dürfen also z.B. Variablen niemals den Namen "template" haben. Eine weitere Änderung, die Inkompatibilitäten verursacht, ist, dass Zeichenliterale (also direkt im Quelltext codierte Zeichen) in C vom Typ int und in C++ vom Typ char sind. Benutzt man sizeof() führt dass in den Sprachen zu unterschiedlichen Ergebnissen.

```c
int main(int argc, char* argv[])
{
    printf("%i", sizeof('a')); //Verschiedene Ergebnisse
    return 0;
}
```

Eine andere relevante Einschränkung ist die unterschiedliche Bedeutung von fehlenden Parameterangaben bei der Funktionsdeklaration. In C bedeutet das "unspezifiziert". D.h. in C sind dann ein oder mehrere Parameter möglich. In C++ bedeutet das allerdings, dass die Funktion keine Parameter hat.

```c
int funktion1() //C++: keine Parameter, C: unbestimmt
{
    return 0;
}

int main(int argc, char* argv[])
{
    funktion1(1); //In C++ ungueltig
    return 0;
}
```

Dies sind allerdings nicht die einzigen Einschränkungen, die zu Inkompatibilitäten führen. Für mehr Informationen über solche Inkompatibilitäten verweise ich auf das Buch "C / C++ - Die praktische Referenz" von Dirk Louis.

4 Fazit

Nun ist noch die Frage offen, ob man eher den C++- oder den C-Compiler verwenden sollte. Aufgrund der Inkompatibilitäten sollte man C -Code allerdings immer durch einen C-Compiler kompilieren lassen und C++-Code durch einen C++ Compiler. Da viele Compiler sich anhand der Dateiendung der Quelldateien für eine Sprache entscheiden, ist es wichtig darauf zu achten, ob eine Datei die Endung ".c" oder ".cpp" besitzt. Man sollte C-und C++-Code nur vermischen, wenn man C-Code in C++-Code verwenden möchte. Allerdings ist es in diesem Fall besser, den C-Code in ".C" Dateien abzulegen, die man dann in eine ".cpp"-Datei einbindet. Dann wird nämlich der C-Code vom C-Compiler und der C++-Code vom C++-Compiler kompiliert. Diese Lösung ist sehr viel eleganter und einfacher als das direkte Vermischen von C- und C++-Code in einer Datei. Zum Schluss werde ich mich noch der Frage widmen, welche Programmiersprache man in einer bestimmten Situation einsetzen soll. Diese Frage lässt sich allerdings nicht eindeutig beantworten, da vieles subjektiv zu beurteilen ist. Ein wichtiger Punkt

bei der Einscheidung ist nämlich der bevorzugte Programmierstil. Nicht jeder mag objektorientiert programmieren. Das gilt auch umgekehrt für nicht objektorientierte Sprachen. Es gibt für beide Seiten viele Pro- und Contra-Argumente. Einige davon wurden ja schon beim Vergleich der Spracheigenschaften erwähnt. Allerdings kann man sagen, dass bei Projekten, in denen es auf Effizienz, Schnelligkeit und Sparsamkeit (wie z.B. im Bereich der Mikrocontroller) ankommt, sich C besser eignet als C++. Aufgrund der Maschinennähe, und den oft sparsameren Implementierungen hat man hier einen entscheidenden Vorteil. Für größere Projekte eignet sich allerdings eher C++, da man durch die Objektorientierung besser dafür sorgen kann, dass ein Projekt überschaubar bleibt.

5 Quellen

C++ - Programmieren mit einfachen Beispielen - Dirk Louis, Markt+Technik Verlag
C / C++ - Die praktische Referenz - Dirk Louis, Markt+Technik Verlag
http://de.wikibooks.org/wiki/C++-Programmierung
http://de.wikipedia.org/wiki/C%2B%2B
http://en.wikipedia.org/wiki/Compatibility_of_C_and_C%2B%2B
http://www.asdala.de/algorithmik/cpp/non-oop.html
http://www.textarchiv.alojado.de/text/vorteile_und_nachteile_der_objektorientierung-ap645.html
http://www.mikrocontroller.net/articles/C_vs_C%2B%2B
http://www.willemer.de/informatik/cpp/cincout.htm
http://www.willemer.de/informatik/cpp/pointer.htm
http://docwiki.embarcadero.com/RADStudio/XE4/de/Virtuelle_Funktionen
http://www.namespace-cpp.de/std/doku.php/kennen/header
http://de.over-blog.com/C_vs_Java_Vor_und_Nachteile-1228321776-art185420.html