C / C++ Syntax: Eine kurze Zusammenfassung

Informatik für Physiker

SS2002

R. Bernet, U. Straumann

24. März 2002

Diese Zusammenfassung soll dem Anfänger als Nachschlagehilfe beim Programmieren behilflich sein. Sie ist weder vollständig noch systematisch konsequent aufgebaut, sie ist vielmehr nach praktischen Kriterien zusammengestellt.

Dem Anfänger wird empfohlen, zuerst die Kapitel 1 bis 3 durchzulesen. Anschliessend können die Uebungen der Reihe nach durchgearbeitet werden, wobei jeweils die notwendigen Befehle und deren Syntax in diesem Skript nachgelesen werden sollen. Die Uebungen, ihre Lösungen und weitergehende Informationen befinden sich auf:

http://www.physik.unizh.ch/teaching/Informatik/Informatik2.html

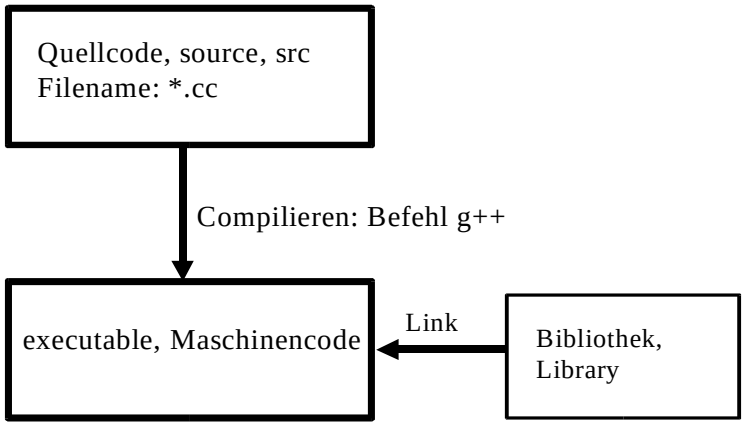
Literatur: "The C++ Language" von B. Stroustrup, Addison Wesley 1987 (und neuere Auflagen, auch in deutscher Uebersetzung) .

Dieser meines Erachtens immer noch ungeschlagene Klassiker ist die wichtigste Referenz zum Thema.

Inhaltsverzeichnis

1. Das Programmieren

"Programmieren" bedeutet, dem Computer in einer wohldefinierten Sprache zu sagen, was wir von ihm erwarten. Im engeren Sinne bedeutet Programmieren das Erstellen von files mit ausführbaren Maschinenbefehlen, sogenannten "executables". Das Vorgehen ist wie folgt:



1.1. Arbeitsablauf beim Programmieren

Der Programmierablauf besteht normalerweise aus drei Schritten:

1. Der Quellcode wird in einer Sprache formuliert (z.B. Pascal, C, C++, Fortran, Basic), dieses File wird mit Hilfe eines editors erstellt. Wir empfehlen den nedit zu verwenden. Setze in nedit Deine Präferenzen so, dass der C Code gut dargestellt wird:

   Preferences -> default -> highlight syntax

   Preferences -> default -> Language Mode -> C++

   Preferences -> default -> show line numbers

   Preferences -> save defaults

2. Die source wird mit Hilfe des Compilers in Maschinensprache übersetzt. Wir verwenden den GNU C++ Compiler, der Befehl lautet g++ *filename* -o *executable-filename*. Wurden im Quellcode Prozeduren aus Bibliotheken verwendet, so werden diese nun mit dem executable verknüpft (link, verlinkt), siehe die Figur oben.

   Hat der Quellcode syntaktische Fehler, die der compliler nicht versteht, meldet er dies unter Angabe der Zeilennummer, -> gehe zu 1. um zu korrigieren. Ist der Code syntaktisch fehlerfrei, dann gehe zu 3.

3. Testen: Das executable wird laufengelassen und seine Funktionalität getestet. Es ist nicht trivial, das richtige Funktionieren vollständig zu testen, vor allem bei Programmen mit vielen unterschiedlichen Abläufen.

Beispiel: Programm u1

1. `nedit u1.cc`	Den Quellcode editieren
2. `g++ u1.cc -o u1`	Compilieren, Syntax-Fehler?
3. `u1`	Laufen lassen: Teste, ob alles richtig ist.

1.2.Sequenz, Selektion, Repetition

"Einfache" Programme laufen grundsätzlich sequentiell ab. Der Programmcode enthält Befehle (statements), die in der aufgeschriebenen Reihenfolge ausgeführt werden. Von dieser Regel kann abgewichen werden durch

1. Selektionen: Eine Ansammlung von statements wird nur unter bestimmten Bedingungnen ausgeführt: Befehle `if` und `switch`.
2. Repetitionen: Statements werden mehrfach ausgeführt: Befehle `for, while, do while`

Sequenzen von mehreren statements können mit Hilfe von geschweiften Klammern { ... } zu Blocks zusammengefasst werden. Dies ist zum Beispiel für die statements bei Selections und Repetitions notwendig.

1.3.Daten

Programme enthalten Konstanten und Variablen. Jede dieser Variablen ist von einem bestimmten Typ (z.B. integer, real, character) und muss entsprechend vor deren Benützung deklariert werden.

Es können auch Vektoren (arrays) oder kompliziertere Daten-Strukturen verwendet werden.

1.4.Subprogramme

Subprogramme oder Prozeduren sind Teil - Programme, die entweder vom Benützer selbst fomuliert werden, oder vorgängig in einer Bibliothek abgelegt worden sind. In C heissen diese functions. Sie können vom Hauptprogramm oder von anderen functions aufgerufen werden, Die dadurch entstehende Hierarchie von Prozeduren erlaubt erst eine effiziente Formulierung von komplexen Programmen.

2. Struktur des Quellcodes

Jedes file mit einem Hauptprogramm ist wie folgt aufgebaut:

```
# include <library-header-filename>     // Filenamen der verwendeten Bibliothek
int main()
{
 statement-sequence                     // Statements des Hauptprogrammes
}
```

Beispiel: Programm `u1`:

```
#include <iostream.h>
int main()
{
    cout << "hello world" << endl;
}
```

Jedes statement muss mit einem Strichpunkt enden. Es kann sich aber über mehrere Zeilen erstrecken, das Zeilenende hat keine Bedeutung.

3. Kommentare

Längere Programme, oder solche, die nicht offensichtlich selbsterklärenden Code enthalten,

sollen mit Kommentaren erläutert werden.

Es gibt folgende beiden Möglichkeiten:

```
/* hier schreibe ich mein Kommentar,
   auch ueber mehrere Zeilen  */

     // hier gilt alles als Kommentar bis zum Ende der Zeile.
```

Die zweite Form gibt es nur in C++. Für sie ist ausnahmsweise das Zeilenende von Bedeutung.

Kommentare sollen vor allem folgende Informationen enthalten:

- Hinweise auf Manuals oder andere Literatur, die als Grundlage für den Code verwendet wurde oder die für das Verständnis notwendig sind.
- Am Anfang jeder function: Bedeutung der Eingangsparameter und der Returnvalues.
- Am Anfang jedes files der Name des Autors für Rückfragen.
- Zusammenhang in dem das Programm verwendet werden soll.

Der Nachteil des Kommentierens besteht darin, dass der Kommentar bei Modifikationen des Programmes falsch werden kann, ohne dass es jemand merkt. Eine Grundregel guten Programmierens lautet deshalb, dass man den Code so selbsterklärend wie möglich schreiben soll. Das heisst man soll klare (möglicherweise relativ lange) Variablenamen verwenden und auf die vielen Abkürzungstricks, die C erlaubt, möglichst verzichten. In diesem Skript sind die Abkürzungstricks aber doch (fast) alle erwähnt, damit man auch schlechte C Programme lesen lernt.

4. Datentypen

4.1. Simple Typen

Es gibt folgende simple Datentypen, das heisst solche, die eine Variable darstellen können:

Typ	***# Bytes***	***Beschreibung***	***Wertebereich***
char	1	einzelne Buchstaben	Alle ASCII Zeichen***
signed char	1	Integer Zahlen	-127 ... +127
unsigned char	1	Natuerliche Zahlen	0 ... +255
short	2	Integer	-32768 ... 32767
unsigned short	2	Natuerliche Zahlen	0 ... 65535
long	4	Integer	-2147483648 ... 2147483647
unsigned long	4	Natuerliche Zahlen	0 to 4294967295
int	*		
float	4	Fliesskommazahl	3.4e + / - 38 (7 digits)
double	8	Fliesskommazahl	1.7e + / - 308 (15 digits)
long double	10	Fliesskommazahl	1.2e + / - 4932 (19 digits)

Typ	*# Bytes*	*Beschreibung*	*Wertebereich*
bool	1	Logische Variable **	`true` oder `false`
void	-	leer	function ohne return value

* int hängt von der Implementation ab. Meistens ist es das gleiche wie long, auf kleinen Systemen manchmal nur short.

** Beim Typ bool wird intern true als 1 und false als 0 gespeichert. Bei der Interpretation einer bool Variable, werden jedoch alle Werte, die verschieden von null sind, als true betrachtet. Der Typ bool ist nicht immer implementiert.

***Computer arbeiten mit Zahlen in der binären Darstellung. Damit man auch Texte verarbeiten kann, werden die Buchstaben mit je einer bestimmten Zahl dargestellt. Es braucht eine Uebereinkunft, welche Zahl welchen Buchstaben darstellt. Es wird meist der ASCII Zeichensatz verwendet (siehe Anhang). Hier wird für jeden Buchstaben ein Byte reserviert. Wird die Variable als `char` deklariert, erfolgt die Interpretation als Zeichen nach der ASCII Tabelle.

Siehe auch die Beispiele in den folgenden Kapitel.

4.2.Konstanten

Ganze Zahlen werden in Dezimaldarstellung angegeben: 3, 17, 28376489903749, -57 sind erlaubte Konstanten. Konstanten in Hexadezimaldarstellung werden mit einem vorangestellen 0x dargestellt, Ziffern über 9 werden mit den ersten Buchstaben des Alphabetes dargestellt (Gross- oder Kleinbuchstaben) : 0x57, 0x3f, 0xA, 0xFFFF usw. Achtung! Konstanten mirt Vornullen werden als Oktalzahlen interpretiert: 010 hat den dezimalen Wert 8.

Fliesskommazahlen: 1.5e-15, wobei innerhalb der Zahl kein Abstand sein darf: 1.5 e-15 ist falsch.

Einzelne Buchstaben (character-Konstanten) müssen in einfache Anführungszeichen gesetzt werden: ‘a‘, ‘A‘ usw. Die ASCII Spezialzeichen werden mit \ dargestellt: ‘\n‘ für linefeed, ‘\t‘ für Tabulator usw. Siehe Tabelle im Anhang.

string Konstanten stehen in doppelten Anfuehrungszeichen ”...” (siehe Abschnitt 6.1)

4.3.Type Casting (Typenumwandlung)

Grundsätzlich ist es nur möglich Variablen gleichen Typs miteinander durch arithmetische oder logische Operatoren zu verknüpfen. Manchmal sind aber auch Verknüpfungen verschiedener Typen notwendig. Meisten macht der Compiler dabei automatisch die richtige Typenkonversion. Im Beispiel:

```
double x = 3.14;
int y = x * 10;
```

wird die ganze Zahl 10 zuerst in den double Wert 10.0 konvertiert, dann wird das Produkt 3.14*10.0=31.4 als Fliesskommaoperation berechnet und schliesslich das Ergebnis wieder in eine ganze Zahl umgewandelt, wobei die Kommastellen verloren gehen: Es wird y=31

Man kann aber auch eine explizite Umwandlung programmieren. Die Syntax lautet in der funktionalen Form:

neuerTyp (alteVariable)

Die ‘casting‘ Form *(neuerTyp) alteVariable* wird auch noch verwendet, gilt aber als unübersichtlich und sollte vermieden werden.

Beispiel:

```
char c = 'a';
cout << int (c) << endl;
```

gibt den ASCII code aus, mit dem der Buchstabe a im Computer codiert wird (97).

Das folgende Programm druckt eine ASCII Tabelle:

```
#include <iostream.h>
int main()
{
  for (int i=32;i<127;i++)
     {char c=char(i);
      cout << i << ' ' << c << endl;
      }
}
```

Man findet das explizite type casting oft in Zusammenhang mit Pointers. Dies ist allerdings sehr fehleranfällig. Guter Programmierstil vermeidet explizites type casting, was in C++ eigentlich immer möglich ist.

4.4.Pointer

Der Speicher eines Computers besteht aus aneinandergereihten Speicherplätzen von jeweils einem Byte (8 bit). Jeder Speicherplatz hat eine Bezeichung, eine Hausnummer, die man Adresse nennt. Die meisten Speicher sind "Byte-adressierbar", das heisst jedes einzelne Byte hat seine eigene Adresse. Mit Pointer oder Zeiger werden Variablen bezeichnet, die solche Adressen anderer Variablen gespeichert haben.

Eine Variable, die Adressen speichern soll, also ein Pointer, wird wie folgt deklariert:

*type *name;*

zum Beispiel:

```
int * aptr;            // aptr ist eine pointer-Variable zu einer Integerzahl
```

Die Leerzeichen um das * herum sind fakultativ. Der Adress- oder Referenzierungsoperator & gibt die Adresse einer Variablen an:

```
int * aptr;
int a;
aptr = &a;
```

Damit ist also in aptr nun die Adresse von a gespeichert. Der Dereferenzierungsoperator * macht das umgekehrte von &, er gibt den Inhalt des Speichers mit der Adresse aptr zurück:

```
int a;
int * aptr = &a;
cout << *aptr;
```

gibt den Wert der Variable a aus (!), das heisst der Variablen, "auf die aptr zeigt".

Pointer werden vor allem im Zusammenhang mit Strukturen verwendet. Mit der Definition des Types personalInfo wie im Abschnitt 6.2 können wir also pointers zu dieser Struktur deklarieren:

```
personalInfo * Iptr;       // deklariere Pointer zur struktur
Iptr = &Reto;              // Pointer kriegt den richtigen Wert
cout << (*Iptr).Alter;     // Dereferenzierung Version 1
```

```
cout << Iptr->Alter;        // Dereferenzierung Version 2
```

Die beiden letzten statements liefern identische Resulate. Die zweite Form des Dereferenzierungsoperator -> wird am häufigsten verwendet.

Beachte: Der Operator * hat also drei Bedeutungen: Multiplikation, pointer Deklaration und Dereferenzierung.

Die Verwendung von Pointers führt oft zu unübersichtlichen und schwer lesbaren Programmen, und ist sehr fehleranfällig. Andererseits erlauben es Pointers, die vorhandenen Ressourcen an Speicher und Rechenleistung effizient zu verwenden.

5. Abgeleitete Typen

5.1. Symbolische Konstanten

Durch das statement

```
const int Anzahl = 100;
```

wird eine Konstante definiert. Sie kann im Programm nicht verändert werden, der Compiler verhindert das mit entsprechender Fehlermeldung. Es handelt sich also nur darum der Zahl (im Beispiel 100) einen Namen zu geben. Anwendung: Man braucht mehrere Felder, die alle die gleiche Länge haben, zum Beispiel:

```
const int Anzahl = 100;
int a[Anzahl];
double r[Anzahl];
if ((i<Anzahl) && (i>=0)) r[i]=55; else Fehler_melden();
```

Will man nun in einer neuen Version des Programmes die Länge der Felder ändern, braucht man nur die Deklaration der Konstante zu ändern, der Rest des Codes ist dann automatisch richtig.

5.2. Aufzähltypen

Oft hat man es mit Aufzählungen zu tun, z.B. Wochentage, Zivilstand, usw. Man könnte natürlich jedem Fall eine Zahl zuordnen, z.B. 1=ledig, 2=verheiratet, 3=geschieden. Stattdessen kann man auch einen Aufzähl -Typ deklarieren:

```
enum Zivilstand {ledig, verheiratet, geschieden};
Zivilstand s=ledig;
```

Die erste Zeile definiert einen neuen Datentyp mit Namen Zivilstand, der die drei aufgezählten Werte annehmen kann. Die zweite Zeile deklariert eine Variable mit Namen s und vom Typ Zivilstand.

Aufzähltypen werden besonders im Zusammenhang mit dem switch statement eingesetzt und können wesentlich zur Lesbarkeit eines Programmes beitragen.

5.3. Typedef

Mit dem statement

Typedef BekannterType NeuerTypeName;

können eigene neue Typen definiert werden. Das folgende Programm verwendet einen eigens definierten Typ namens Orts_vector, der aus einem array (siehe 6.1) mit 3 Elementen vom Typ double besteht:

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

```
typedef double Orts_vector[3];
int main()
     {
     Orts_vector a = {1,2,3}, b = {2,4,6};
     cout << "a b" << endl;
     for (int i=0; i<3; i++)
          {
          cout << a[i] << " " << b[i] << endl;
          }
     }
```

Die durch typedef definierten neuen Typen sind also eigentlich eine Abkürzung schon bekannter Typen.

Die Sache wird natürlich erst richtig effizient, wenn man nun für diese neuen Typen auch neue Operatoren definieren könnte, zum Beispiel den Abstand ausrechnen, oder das Skalarprodukt. So was nennt man objektorientiertes Programmieren. Man kann das mit Hilfe von Klassen und Operatoroverloading leicht machen.

6. Zusammengesetzte Typen

6.1. Arrays

sind Felder oder Vektoren, also eine eindimensionale Ansammlung von Daten des gleichen Typs. Man deklariert sie mit einer eckigen Klammer, die die Zahl der Elemente enthält. Will man ein Element des Vektors indizieren, muss man ebenfalls die eckige Klammer verwenden. Der Index beginnt immer bei 0. Beispiel:

```
int a[100];                          // a hat 100 Elemente
for (i=0; i<100; i++) a[i]=0;        // der index läuft von 0 bis 99
```

Achtung! Das Einhalten der Indexgrenzen wird normalerweise nicht überprüft. Der Programmierer ist dafür verantwortlich, dass nicht über die Arraygrenzen hinaus geschrieben wird. Das Programm

```
int a[100];
for (int i=0; i<1000; i++) a[i]=0;     // FALSCH! array index overflow
```

produziert nur die lapidare Fehlermeldung "segmentation fault".Gute Programme kontrollieren deshalb explizit bei jeder Verwendung von Indizes, ob diese die erlaubten Arraygrenzen nicht überschreiten. Siehe Beispiel im Abschnitt 5.1 über symbolische Konstanten.

Mehrdimensionale Felder: Beispiel eine 2 mal 3 Matrix:

```
double x[2][3];
x[0][0] = 3;
```

Initialisierung von Feldern erfolgt mit geschweiften Klammern:

```
int a[] = {1, 4, 2};
int b[][2] = { {3,4} , {7,9} }
```

dabei wird b[0][0]=3, b[0][1]=4, b[1][0]=7 und b[1][1]=9. Da hier durch die Initialisatierung die Dimension klar ist, kann diese Zahl in den [ ] Klammern weggelassen werden, für multidimensionale Arrays allerdings nur bei der ersten Dimension.

Bei Feldern aus Pointer muss man bei der Deklaration aufpassen:

```
int* v[10];      // v ist ein array von Pointers
int (*p)[10];    // p ist ein Pointer zu einem Array aus Int
```

Strings sind arrays aus dem Typ char. Strings werden standardmässig mit einer null beendet. Das heisst ein string hat immer ein Element mehr als Buchstaben:

```
char a[]="Ueli";      // Stringkonstanten immer in Doppelquotes
```

a[0] ist also gleich 'U', a[1] gleich 'e', usw., a[4]=0. Die null kann auch als ASCII null geschrieben werden, das heisst a[4]='\0'. Alle Funktionen in der libray string.h erwarten nullterminierte strings.

Der Name eines Vektors ohne Klammern bezeichnet den Pointer zu seinem ersten Element:

```
char a[]="Ueli";
char *p = a;     // dasselbe wie: char *p = &a[0];
while (*p) {cout << *p; *p++; }
```

Erklärung: In der ersten Zeile defniert "..." ein nullterminierter String. p ist der Pointer zum ersten Element von a. *p hat also am Anfang der letzten Zeile den Wert 'U'. Nach der Ausgabe wird der Pointer um eins erhöht, im nächsten Durchgang wird 'e' ausgegeben, usw. Am Schluss des Strings wird *p null, was als false interpretiert wird, die while Bedingung ist nicht mehr erfüllt und der loop bricht wie gewünscht ab (man kann statt dem while loop in diesem Fall einfach schreiben cout << a; das Beispiel mit dem while loop soll nur zeigen, wie man die null terminierten strings verwenden kann).

Diese Eigenschaft wird auch bei der Uebergabe von arrays an Funktionen verwendet. Mit obiger Deklaration ist dann f(a) dasselbe wie f(p), siehe auch Kapitel 12.

6.2.Strukturen und Klassen

Klassen und Strukturen erlauben das Speichern von vielen Elementen mit unterschiedlichen Typen unter demselben Namen. Beispiel:

```
struct personalInfo
{
     int idNummer;
     char Name[80];
     int Alter;
     void printInfo()
          {cout << idNummer << Name << Alter << endl;}
};
```

definiert eine Struktur, die einen Namen (string), zwei Zahlen sowie eine function für jede damit deklarierte Variable ins Leben ruft. Es handelt sich um die Definition eines neuen Typs ähnlich wie bei typedef (vgl. Kapitel 5.3.). Die Funktion printInfo wird genau so deklariert, wie jedes andere Unterprogramm, siehe Kapitel 12).

Beachte den Strichpunkt nach } !

Strukturen, die nur aus Variablen bestehen, heissen Records.. Strukturen, die nicht nur Variablen, sondern auch Funktionen enthalten, werden Objekte genannt. Solche Element-Funktionen nennt man auch Methoden des Objekts. Statt mit `struct` kann man Objekte auch mit `class` definieren, der Unterschied besteht nur in der Zugriffskontrolle (siehe 6.3).

Mit

```
personalInfo Peter, Reto, AlleAnderen[100];
```

(Deklaration in der Form *type variable-name)* werden mehrere Variablen vom Typ personalInfo erzeugt, man nennt sie Instanzen des Objektes personalInfo.

Zugriff erhält man auf die Elemente der Klasse wie folgt:

```
Peter.idNummer = 12345;
Reto.Alter = 48;
AlleAnderen[i].Alter = 50;
```

Das folgende statement druckt die drei Informationen des 31. Elementes aus:

```
AlleAnderen[30].printInfo();
```

Definiert man in einem Objekt eine Methode mit dem gleichen Namen, wie das Objekt selber, erhält man einen Konstruktor: z.B. personalInfo.personalInfo(). Ein Konstruktor wird automatisch am Anfang ausgeführt, immer wenn eine Instanz des Objektes aktiviert wird (zum Beispiel am Anfang der in {...} gesetzten statement - Sequenz, in der eine Instanz dieses Objektes deklariert ist).

Stellt man dem Methodennamen zusätzlich ein ~ davor, handelt es sich um den Destruktor, der automatisch am Ende aufgerufen wird, wenn die Instanz zerstört wird.

6.3.Zugriffskontrolle: public, private, friend, Zugriffsoperator ::

Jedes Element eines Objektes kann durch Zugriffsrechte geschützt werden. Man unterscheidet zwischen `public` und `private`. In einem mit class definierten Objekt ist das default Zugriffsrecht private, bei struct ist der default public (das ist der einzige Unterschied zwischen den Schlüsselwörtern struct und class). Den Deklarationen der Elemente eines Objektes können die Keyworte public: oder private: vorangestellt werden.

Privat deklarierte Elemente eines Objektes können nur in den Funktionen (Methoden) dieses Objektes verwendet werden, während die public deklarierten Elemente von überall her mit dem . oder dem -> zugreifbar sind.

Diese Zugriffskontrolle lässt sich durch `friend` functions umgehen: Sei eine Funktion int g() global im file deklariert. Erwähnt man nun innerhalb der Objekt - Definition

```
public: friend int g();
```

Dann erhält die globale Funktion g() die Zugriffsrechte einer Elementfunktion des Objektes, sie kann also auch auf die private deklarierten Elemente des Objektes zugreifen.

Schliesslich hat man die Möglichkeit, mit Hilfe des Zugriffsoperators :: (Scope Operator) die Zugehörigkeit eines Elementes zu einem Objekt eindeutig zu bestimmen. Sei zum Beispiel int f() eine Methode, die im Objekt abc als Funktionsprototyp deklariert sei (private oder public). Möchte man nun im globalen Teil des Programmes die Implementation von f() beschreiben, muss man diese mit Hilfe von :: bezeichnen, um Zugriff auf die privaten Elemente des Objektes zu bekommen:

```
struct abc
{     ...
      private:
            int f(int);
}
...
int abc::f(int i) { ... }
```

Man beachte auch den Unterschied: Zugriff auf ein Element eines Objektes erfolgt mit dem Zugriffsoperator ::, Zugriff auf ein Element der Instanz eines Objektes erfolgt mit Hilfe des . Operators (oder des ->).

6.4.Vererbung

Hierarchien von Objekten werden mit Hilfe der Vererbung definiert. Ist ein Objekt B definiert:

```
struct B {
      double f(){}
      double b;
}
```

Dann kann man ein neues von B abgeleitetes Objekt A definieren:

```
struct A : B {
      double x
      double g(){...}
}
```

Das Objekt A besteht nun also aus den Variablen b und x und ausserdem aus den Funktionen f() und g(). B heisst die Basisklasse, A die abgeleitete Klasse. Der Vorgang heisst Vererbung.

Man darf im vererbten Objekt auch Elemente umdefinieren:

```
struct C : B {
      double f(){cout << "ich bin das neue C.f"};
}
```

6.5.Operator overloading

Oft erschiene es logisch, wenn man die Standardoperatoren für einen bestimmten zusammengesetzten Typ umdefinieren könnte. Zum Beispiel möchte man gerne den Typ Ortsvektor definieren, der aus drei Komponenten x, y, z besteht. Seien v und w solche Ortsvektoren. Dann möchte man , dass der Ausdruck v*w die Bedeutung des Skalarprodukts erhält. Das folgende Beispielprogramm definiert eine Struktur, die das leistet:

```
#include <iostream.h>
int main()
{
    struct Ortsvektor
    { double x, y, z;
        double operator*(Ortsvektor a)
            {return a.x*x + a.y*y + a.z*z;}
    };

    Ortsvektor v,w;
    cout << "enter v = (x, y, z) ";
    cin >> v.x >> v.y >> v.z;
    cout << "enter w = (x, y, z) ";
    cin >> w.x >> w.y >> w.z;
    cout << "Skalarprodukt = " << v*w << endl;
```

```
}
```

Dem keyword `operator` wird dabei das Operatorzeichen angehängt, das umdefiiniert werden soll, in diesem Fall die Multiplikation. `Double operator*(...) {...}` verhält sich nun wie eine normale Funktionsdeklaration (siehe 22). Der Aufruf kann nun entweder mit

```
v.operator*(w)
```

oder eben auch elegant mit

```
v*w
```

erfolgen.

Es können alle in der Tabelle im Anhang aufgelisteten Operatoren umdefiniert werden, ausser die Geltungsbereichoperatoren :: und der Konditionaloperator ?:. Die Prioritäten und die Zahl der Argumente (unär resp. binär) der Operatoren bleiben jedoch erhalten, und können nicht verändert werden.

7. Variablen

7.1. Variablenamen

Die Wahl der Variablennamen ist grundsätzlich frei, mit folgenden Einschränkungen:

1. Klein- und Grossbuchstaben werden unterschieden.
2. Man darf auch Ziffern verwenden, aber nicht am Anfang des Variablennamens.
3. Das Zeichen _ darf überall im Variablennamen vorkommen, alle anderen Spezialzeichen haben eine andere Bedeutung und dürfen deshalb nicht in Variablennamen vokommen.
4. Schlüsselwörter dürfen nicht verwendet werden, das sind solche die eine andere Bedeutung haben, z.B. if, double etc. Die vollständige Liste von Schlüsselwörtern findet man im Anhang.
5. Die maximale Länge von Namen ist beschränkt, sie hängt von der Implementation ab, ist aber in Praxis beliebig gross.

7.2. Deklarationen

Eine Deklaration ist ein statement der Form:

type variablename [initialisation];

Jede Variable muss vor ihrem ersten Gebrauch deklariert werden, dies kann an jeder beliebigen Stelle im Programm erfolgen (nur C++). Die Initialisierung ist fakultativ, ist aber sehr zu empfehlen, da nicht initialisierte Variablen eine der häufigsten Fehlerquellen darstellen. Die Deklaration kann mit einem assignment verbunden sein. Beispiel: Programm u2.cc:

```
#include <iostream.h>
int main()
{
  double x,y=1;                  // y wird initial., x nicht
  cout << " x = "; cin >> x;
  cout << " y = "; cin >> y;
  double sum_2 = x + y;          // Deklaration mit assignment
  cout << " Summe = ";
  cout << sum_2;
  cout << endl;
```

```
}
```

7.3.Sichtbarkeit (Scope)

Die Sichtbarkeit (Scope) einer Variablen hängt von der Stelle ab, wo sie definiert wird. Grundsätzlich kann eine in einem Block zwischen { ... } definierte Variable nur in diesem Block, und in allfälligen Unterblöcken davon verwendet werden, ausserhalb ist sie unbekannt. Man nennt diese Variable lokal in diesem Block. Ein ausserhalb der Funktionen und ausserhalb von main() definierte Variable heisst global. Das folgende Fragment eines Programmes möge das illustrieren:

```
int x = 10;      // globale Deklaration, im ganzen file sichtbar
int main()
{
      int a=5;   // lokale Deklaration in der function main
      int x=20
      {
            int y = a;      // y = 5, nur in dem Block sichtbar
            int z = x;      // z = 20;
            int x = 30;     // ab jetzt ist x = 30, lokal!
            ...
            b = x;          // b=30
            c = ::x;        // c=10
      }
}
```

In diesem Beispiel wird die Variable x dreimal mit verschiedener Sichtbarkeit definiert. Das ist erlaubt, und führt zu drei verschiedenen Variablen x mit verschiedenen Werten, was nicht besonders übersichtlich ist. Im Inneren des Blockes wird mit x die Variabale angesprochen, deren Deklaration im innersten Block stattfindet. Mit Hilfe des Geltungsbereichoperator (scope operator) :: kann man diejenige Variable x ansprechen, die am weitesten aussen deklariert wurde.

Eine Grundregel guten Programmierens lautet, dass man Variablen immer so lokal definieren soll, wie möglich und nur so global wie nötig. Damit vermeidet man viele Programmierfehler.

7.4.Lebensdauer

Lokale Variablen, die innerhalb eines Blockes definiert werden, existieren nur so lange, wie der Block gerade ausgeführt wird, am Ende des Blockes werden sie gelöscht. Möchte man nun in einem späteren Zeitpunkt den Block nochmals aufrufen, und soll z.B. die lokale Variable `xcall` ihren alten Wert behalten, muss man sie static deklarieren:

```
int bla()
      {
            static int xcall = 0;      // Initialisierung, nur 1. Mal
            xcall++;                   // Anzahl Aufrufe von bla()
            ...
      }
```

Das Gegenteil von static, alse der Nomalfall, heisst auto. Das muss man aber nicht explizit

angeben, auto ist default.

8. Input - Output

In C++ werden in der Bibliothek iostream.h die Operatoren << und >> definiert, sowie verschiedene Flags für das Formattieren. In fstream.h gibt es die zusätzlichen Funktionen für das Schreiben und Lesen von Files.

8.1. Terminal IO

Iostream.h kennt die Standardein- und ausgabe streams von UNIX, die mit cin, cout und cerr bezeichnet werden.

```
# include <iostream.h>
...
cout << "hallo hier bin ich, bitte x!=0 eingebn"// standard out
cin >> x;                                      // standard in
if (x==0) {cerr << "Fehler: x=0" << endl;} // standard error
```

Anstelle von Ausdrücken, die ausgegeben werden sollen, kann nach << auch folgendes stehen:

```
<< endl          // gibt ein linefeed aus.
<< ends          // gibt ein end of string (ASCII NULL) aus
<< hex           // hexadecimal Darstellung der folg. Zahlen
<< oct           // Oktal Darstellung
<< dec           // Dezimaldarstellung (default)
```

Die folgenden Funktions-Aufrufe spezifizieren die Formattierung des Outputs.

```
cout.width(4);  // Feldbreite, in das die Zahl geschrieben wird
cout.setf( ios::fixed, ios::floatfield) ;  // Fixkomma
cout.precision(2); // Anzahl Stellen nach dem Komma
cout.setf( ios::scientific, ios::floatfield); // Exp. Darst.
```

Statt diesem C++ I/O processing werden auch noch die Standard C Routinen printf und scanf aus stdio.h verwendet, denen man einen Formatierstring und die auszugebende Zahl übergibt:

```
# include <stdio.h>
...
printf("Die Zahl x ist %d \n",x);
```

%d ist ein Beispiel für einen Platzhalter, an dessen Stelle bei der Ausgabe die Zahl im angegebenen Format gesetzt wird. Als Platzhalter kann man z.B. angeben:

```
%7d        // Integerzahl in Dezimalnotation, Breite 7 Zeichen
%15.12f    // float or double als Fixkommazahl, Breite 15,
           // 12 Stellen nach dem Komma.
%15.2e     // Exponential-Darst., Breite 15 Zeichen,
           // 2 Stellen nach dem Komma
%c         // character
%s         // string
```

Es koennen auch mehrere Platzhalter und entsprechend viele Variablen verwendet werden:

```
printf("x = %d, y = %d \n",x,y);
```

8.2.File Input und Output

Fstream.h enthält die Objekte, mit denen man file I/O machen kann.Ein file wird als Objekt vom Typ fstream deklariertt. Dann kann mit den Zeichen << und >> von diesem file gelesen und geschrieben werden, ganz analog zu cin und cout. Beispiel:

```
# include <fstream.h>
...
fstream bla;     // interne Bezeichnung fuer das file Objekt
bla.open(extFilename,mode);
                 //extFilename ist der Filename im Betriebsystem
                 // mode kann sein: ios::in   lesen
                 //                 ios::out  neu beschreiben
                 //                 ios::app  append
if (bla.good()) // true, wenn open erfolgreich war.
   { bla >> x;} // lese vom file und speichere den Wert in x.
if (bla.eof()){}// true, wenn das Ende des files erreicht ist.
char c[80];      // string definiert
while (bla)      // true, wenn noch etwas zum Lesen da ist.
  {bla >> c;}    // liest strings, begrenzt durch Leerzeichen
bla.close()      // file schliessen.
```

9. Operatoren, Assignments, Vergleiche

9.1.Operatoren

Die grundlegenden arithmetischen Operatoren sind:

*	Multiplikation
/	Division (bei ganzen Zahlen abgerundet)
%	modulo = Rest der Division (nur ganze Zahlen)
+	Addition
-	Subtraktion

Beispiel: 10/3 = 3; 10%3 = 1.

Die logischen Operatoren

!	Negation
&&	UND
||	ODER

sind auf den Typ bool anwendbar, aber auch auf alle Integer. Dabei wird 0 als false, jeder andere Wert als true interpretiert.

Die Operatoren ~, &, | und ^ bedeuten bitweise Negation, UND, ODER und exclusiveOR. Sie

spielen in hardware nahen Programmen eine Rolle. Weiter gibt es die Operatoren << (shift left) und >> (shift right), die die bits innerhalb einer Variablen schieben, Beispiel: int a=5; a<<2 bedeutet "schiebe a zweimal nach links", das Resultat ist 20 (!) Nicht zu verwechseln mit den Input / Output Operatoren << und >>.

Eine vollständige Liste aller Operatoren findet man im Anhang.

Viele weitere Funktionen sind in der Bibliothek math.h vorhanden (siehe Zusammenfassung am Ende dieses Skripts und im Anhang).

9.2.Assignment

Ein assignment ist die Zuweisung des Resultates einer Berechnung an eine neuen Variable, zum Beispiel:

```
int i=5;
double y, pi=3.14;
y = pi * i;
```

Im letzten statement wird pi*i ausgerechnet, und das Resultat in der Variablen y abgespeichert. Auf der linken Seite des Gleichheitszeichens in einem assignment steht deshalb immer eine Variable, eine sogenannter lvalue, und niemals Operatoren. (Genauer: Ein Objekt k ist genau dann ein lvalue, wenn die Adresse &k existiert und beschrieben werden kann.) Werte von Ausdrücken, die nur auf der rechten Seite eines assignments stehen können (die nur Werte herausgeben können und nicht adressierbar sind), heissen rvalue.

Runde Klammern werden wie in der Mathematik verwendet um die Reihenfolge der Auswertung festzulegen. Beispiele:

```
y = 3 * (5 + 1)       // 18
y = (3 * 5) + 1       // 16
y = 3*5+1             // 16
y = 2+4*6             // 26
```

Es gelten also die gleichen Regeln wie in der normalen Algebra. Die Reihenfolge der Auswertung von Ausdrücken mit mehreren Operatoren wird duch die Priorität der Operatoren festgelegt, man findet die Prioritäten ebenfalls in der Operatortabelle im Anhang. z.B. hat * höhere Priorität als +,

Die Abkürzungen

```
i++   für   i=i+1
i--   für   i=1-1
i+=5  für   i=i+5
i/=x  für   i=i/x
```

werden for allem in Repetitionsstrukturen verwendet. Man kann diese Ausdrücke aber auch weiterverwenden, wie im folgenden Fragment:

```
int i=1,x,y;
x = i++;        // nach dieser Zeile ist i=2 und x=1
y = ++i;        // nach dieser Zeile ist i=3 und y=3
```

Bei der Postfix-Form, i++, wird i zuerst verwendet und dann erhöht, während in der Prefix-Form, ++i, i zuerst erhöht wird, und das Resultat nachher verwendet wird.

9.3.Vergleichsoperationen

Das Resultat eines Vergleichs ist entweder true (=1) oder false (=0) Es gibt folgende Vergleichsoperatoren:

```
<     <=    >     >=    ==    !=
```

mit offensichtlicher Bedeutung. Beachte den Unterschied zwischen dem assignment (=) und dem Vergleichsoperator auf Gleichheit (==). Beispiel:

```
bool b;
int x=5, y=5, z=6;
b = (x==y);           // true
b = (x!=y);           // false
b = (z>x);            // true
```

Man kann die Resultate der Vergleichsoperationen auch in Integer - Variablen speichern, sie erhalten dann die Werte 0 bzw. 1 für false bzw. true.

ACHTUNG: Das Testen auf Gleichheit zweier Fliesskommazahlen mit Hilfe des == Operators sollte vermieden werden, da das wegen der endlichen Rechengenauigkeit zu Fehlern führen kann.

10.Selektion

10.1. if statement

Die Syntax des if statements lautet wie folgt:

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

Falls die Bedinung erfüllt ist, wird Anweisung1 ausgeführt, sonst Anweisung2. Der else Teil ist fakultativ. Die Anweisungen können auch statement-sequences sein, in diesem Fall muss man {...} verwenden, um diese in einem Block zusammenzufassen. Beispiel:

```
if (z>=0)
      {
            x=-z;
            cout << "z ist positiv oder null";
      }
else
      {
            x=z;
            cout << "z ist negativ";
      }
```

(Beachte, dass hier nach einer } kein ; stehen muss.)

Abkürzungen: (sind im Interesse der Klarheit wie immer nicht zu empfehlen)

Das logische Resultat der Bedingung wird mit 0 (für false) und 1 (für true) dargestellt. Das if statement interpretiert eine Bedingung mit Wert 0 entsprechend als false, alle anderen Werte als true. Das statement `if (n) Anweisung ;` bedeutet deshalb, dass die Anweisung genau dann ausgeführt wird, wenn n ungleich 0 ist.

Im weiteren gibt es den Konditionaloperator ?:, es handelt sich um einen ternären Operator, der

durch die Aequivalenz der letzten beiden Zeilen des folgenden Fragmentes definiert ist:

```
int x,y,z;
bool b;
...
if (b) z=x; else z=y;      // ausführlich
z=b?x:y;        // Abkürzungstrick mit Konditionaloperator
```

10.2.switch statement

Will man Fallunterscheidungen mit meheren Fällen machen, kann man das switch statement verwenden:

```
switch ( Ausdruck )
{
     case Konstante1 : Anweisung1 break;
     ...
     case KonstanteN : AnweisungN break;
     default : AnweisungN+1 ;
}
```

Wenn das Resultat der Evaluation des *Ausdruckes* dem Wert einer der Konstanten entspricht, wird die entsprechende Anweisung (oder statement-sequence) ausgeführt. Passt der Ausdruck zu keiner Konstanten, wird die AnweisungN+1 ausgeführt. Das Resultat des Ausdruckes muss eine ganze Zahl sein, die Konstanten müssen ebenfalls Integer (oder char) sein. default ist fakultativ. Die Ausführung des switch statements wird beim ersten Auftreten eines break beendet. Lässt man also ein break weg, so wird auch die darauf folgende Anweisung ausgeführt, man kann also mehrere Fälle zusammenfassen (siehe das folgende Beispiel).

Typisches Beispiel: Das Programmieren von Menuoptionen:

```
char ch;
cout << "Waehle unter (a,b,c): ";
cin >> ch;
switch ( ch )
{
     case 'a' : case 'A' : /* Programm fuer Fall a */ break;
     case 'b' : case 'B' : /* Programm fuer Fall b */ break;
     case 'c' : case 'C' : /* Programm fuer Fall c */ break;
     default : cout << "Eingabe ungueltig! << endl ;
}
```

Häufig werden Aufzähltypen (enum) als switch Variable verwendet.

11.Repetition

Repetitionen werden verwendet für mathematische Iterationen, aber auch für das Abarbeiten von mehreren, ähnlich gelagerten Fallunterscheidungen. Für alle Repetitionsstrukturen gilt, dass man aufpassen muss, dass die Schleife auch mal endet, sonst läuft das Programm bis in alle Unendlichkeit im Kreis herum.

11.1.while statement

Die Syntax lautet:

```
while ( Bedingung  )  Anweisung ;
```

Am Anfang (auch vor der ersten Ausführung) wird die Bedingung evaluiert. Ist sie true wird die Anweisung ausgeführt. Der loop läuft solange bis die Bedingung nicht mehr erfüllt ist. Die Anweisung wird im allgemeinen eine statement-sequence sein, sie muss in diesem Fall in {...} stehen.

11.2.do while statement

Es handelt sich um dasselbe wie das while, aber die Bedingung wird nach statt vor der Anweisung evaluiert:

```
do Anweisung  while ( Bedingung  )  ;
```

Die Anweisungen werden in diesem Fall also immer mindestens einmal ausgeführt, auch wenn die Bedingung nie erfüllt ist. Beim while statement muss die Bedinungung mindestens am Anfang erfüllt sein, damit die Anweisungen einmal ausgführt werden.

11.3.for loops

Die Syntax für den for loop lautet:

```
for (  Initialisierung ; Bedingung  ; Update  ) Anweisungen ;
```

Beim ersten Mal wird die *Initialisierung* ausgeführt. Bei jedem Durchgang wird vorerst die *Bedingung* evaluiert. Ist sie falsch, wird der loop abgeborchen. Ist sie wahr, wird die *Anweisung* (oder der durch {...} definierte Block) ausgeführt, anschliessend wird der *update* ausgeführt. Beispiel:

```
for ( int i=0, p=2 ; i<4 ; i++ )
{
      p=p*p;
      cout << p << endl;
}
```

Die Initialisierung und der update können mehere, durch Kommas separierte statements enthalten. Alle Teile der for Schlaufe sind fakultativ, das statement `for(;;);` stellt eine Endlosschlaufe dar.

11.4.continue und break

Das statement `continue;` unterbricht den aktuellen loop Durchlauf an dieser Stelle und startet unmittelbar den nächsten Durchlauf. Das statement `break;` bricht den gesamten loop sofort ab. Beispiel:

```
int b=1;
for (;;)
{
      b++;
      if (b>10) break;
      if (b>5) continue;
      cout << b << endl;
}
```

```
cout << "Ende, b=" << b << endl;
```

liefert folgenden output:

2

3

4

5

Ende, b=11

Continue und break können in allen Repetitionsstrukturen eingesetzt werden. Sie sollten allerdings sparsam verwendet werden, da sonst bei komplexen Programmen gerne die Uebersicht verloren geht.

12.Unterprogramme

Unterprogramme dienen vor allem dazu, grosse Quelltexte besser zu strukturieren. Damit können komplizierte Programme in logisch zusammenhängende Komponenten aufgespalten werden. Unterprogramme bilden eine funktionelle Einheit, deren lokale Variablen von aussen nicht sichtbar sind. Unterprogramme oder Prozeduren werden in C++ *function* genannt, da sie sich genau wie eine mathematische Funktion verhalten.

Vor ihrem Gebrauch muss jede function definiert werden (Deklaration). Dabei muss man festlegen, welche Parameter oder Variablen die Funktion als Input braucht, und welche Variablen sie zurückgeben soll. Zum Gebrauch muss die Funktion aufgerufen werden (Call).

Unterprogramme sollten - wie das Hauptprogramm - nicht zu lange sein. Bei einem guten Programmierstil passt jedes Unterprogramm auf eine Bildschirmseite. Längere Algorithmen sollten in weitere Unterprogramme aufgespalten werden.

12.1.Deklaration einer function

Auch jede Funktion muss VOR ihrer Verwendung deklariert werden. Die Syntax einer function Deklaration lautet:

```
returntype funcname (argumentlist) { body };
```

Der *returntype* definiert den Typ der Variable, die als Resultat der Funktionsauswertung zurückgegeben werden soll. Es sind alle Typen als Returntyp erlaubt (auch struct) mit Ausnahme von *array*. Eine Funktion, die gar nichts zurückgibt, hat den Typ `void`, eine solche function heisst auch Prozedur.

funcname ist ein beliebig wählbarer Name, es gelten die gleichen Einschränkungen wie im Abschnitt 7.1 beschrieben.

argumentlist enthält die Deklarationen der Eingangsparameter für die Funktion. Falls die Funktion keine Eingangsparameter benötigt, muss hier ein leeres Klammerpaar stehen: (). Dadurch wird die Funktion als solche erkannt. Im Gegensatz zur gewöhnlichen Variablendeklaration können nicht Parameter vom gleichen Typ zusammengefasst werden:

```
int f (double x,y) { ... }            // FALSCH
int f (double x, double y) { ... }    // RICHTIG
```

Man spricht hier von formalen Parametern, bei der Deklaration der Funktion haben sie noch keinen effektiven Wert. Die hier verwendeten Namen sind jedoch diejenigen, die dann im body verwendet werden können. Beim Aufruf der Funktion werden hier effektive Werte eingefüllt, entweder als Konstanten, oder als Variablen mit den im aufrufenden Programm deklarierten Namen, man spricht dann von aktuellen Parametern, die im allgemeinen einen anderen Namen

als die formalen Parameter in der Deklaration haben.

body enthält die Folge der statements, die die Funktion ausführen soll. Das `return` statement definiert dabei, welche Grösse die Funktion als Resultat zurückgeben soll.

Das folgende Trivial - Beispiel deklariert eine Funktion, die das doppelte des Eingangsparameters zurückgibt:

```
int twice (int i)
{
     int t = 2 * i;
     return t;
}
```

Im übergeordneten Programm wird die Funktion wie folgt aufgerufen:

```
int k = 3, s;
....
s = twice(k);
```

Hier ist also k der aktuelle Parameter und i der formale Parameter.

Der Compiler überprüft, ob Zahl und Typen der formalen Parameter mit den aktuellen Parametern übereinstimmt.

12.2.Funktionsprototyp oder Signatur

Wenn man eine Funktion vor ihrer vollständigen Deklaration verwenden will, oder falls die Funktion Teil einer Bibliothek ist und ausserhalb des Files verwendet werden soll, in dem sie deklariert ist, kann man an jeder Stelle vor der eigentlichen Deklaration ein Prototyp formulieren. Für das obige Beispiel lautet der Prototyp:

```
int twice (int);
```

Der Prototyp ist also die Deklaration ohne Funktionsbody, die nur die Typen, aber nicht die lokalen Namen der formalen Parameter enthält. Natürlich muss der Funktionsprototyp bez. Name, sowie Zahl und Typ der Variablen mit der später erfolgenden vollständigen Deklaration übereinstimmen.

12.3.Inline function

Normalerweise geschieht das Aufrufen einer function dadurch, dass vorerst der vollständige Zustand des Rechenwerkes mit allen Registern zwischengespeichert wird. Dann wird die Funktion ausgeführt, und anschliessend der ursprüngliche Zustand des Rechenwerkes wiederhergestellt. Mit dem der Deklaration vorangestellten Schlüsselwort inline, wird stattdessen der Code der function an die Stelle des Aufrufes hineinkopiert:

```
inline int twice(int i) { return 2 * i;}
```

Damit kann man also die Ausführung des function Aufrufes optimieren.

12.4.Passing by value, passing by reference

Normalerweise werden die Werteparameter beim Aufruf als aktuelle Parameter in die function kopiert (passing by value). Das bedeutet, dass ein Parameter in der function verändert werden kann, ohne dass dies die entsprechende Variable im aufrufenden Progamm verändert. Das folgende Beispiel berechnet die Quadratwurzel einer Zahl A nach dem Newton'schen Algorithmus. x ist ein Schätzwert für das Resultat, mit dem der Algorithmus startet:

```
double sqroot(double A, double x)
```

```
{
      while (fabs(x*x - A) > 1E-10)
            x= (x+A/x)/2;
      return x;
}
```

x ist also ein Werteparameter, der in der Prozedur verändert wird. Das aufrufende Programm merkt davon aber nichts: Auch nach dem Aufruf

```
y = sqroot ( A, x);
```

hat x noch immer denselben Wert.

Möchte man ausnahmsweise die Veränderung eines Parameters auch im Hauptprogramm zur Verfügung haben, muss man den Parameter by reference übergeben, man spricht auch von Variablenparameter. Dies geschieht durch Einfügen des Zeichens & (die sogenannte Referenz) zwichen Typ und Name des formalen Parameters. Dabei wird der Prozedur statt dem Wert nur der Pointer zur Variable übergeben, eine Veränderung des Wertes der Variablen in der function hat dann offensichtlich globale Konsequenzen: Beispiel:

```
void vertausche ( double& a, double& b)
{
      double h;
      h = a; a = b; b = h;
}
```

Hier werden also die Werte der beiden Variablen vertauscht.

ACHTUNG1: Arrays werden immer als Variablenparameter übergeben, auch man das & nicht angibt. Mit dem Ausdruck `const` vor der Deklaration der formalen Parameter verhindert der Compiler aber eine Veränderung der Werte innerhalb der function.

ACHTUNG2: Diese Referenz gab es beim normalen C noch nicht, sodass (leider immer noch) häufig explizit pointers als Parameter von Funktionen programmiert werden, wenn man den Parameter ändern möchte.

Zusammenfassend gibt es also 3 Möglichkeiten, wie man das Resultat einer Funktion ins aufrufende Programm zurückgegeben kann: Als Beispiel soll das Doppelte von k in s gespeichert werden:

1. Als return parameter:

```
int twice1(int i) {return 2*i;}     // Deklaration
s = twice1(k);                      // Aufruf
```

2. Als Variablenparameter

```
void twice2(int& i) {i=2*i;}        // Deklaration
s=k; twice2(s);                     // Aufruf
```

3. Als pointer

```
void twice3(int* p) {*p=2*(*p);}    // Deklaration
s=k; twice3(&s);                    // Aufruf
```

Version 1 soll normalerweise verwendet werden. Version 2 muss man nur verwenden, wenn die Variable vom Typ array ist. Version 3 soll man vermeiden, sie ist hier nur zum Verständnis angegeben.

12.5.Rekursion

Da die Werteparameter für die Ausführung einer function in diese hineinkopiert werden, können functions auch sich selber aufrufen. Dieses Programmierkonzept heisst Rekursion. Sie lehnt sich an die mathematische Definitionsmethode der Induktion ab. Siehe die Uebungsbeispiele.

13.Standard - Bibliotheken

Die folgenden Bibliotheken sind in allen Implementationen vorhanden. Eine detaillierte Liste aller Funktionen dieser Bibliotheken findet man im Anhang. Für die genauen Definitionen der Funktionen dieser Bibliotheken siehe z.B. unter

http://www.cplusplus.com/ref/ (enthält zu jeder Funktion die Definition und ein Beispiel)

13.1.stdio.h

enthält die klassischen C Routinen, um files zu schreiben und zu lesen. Ebenso die klassischen IO functions scanf und printf, resp. fscanf und fprintf.

Verwende aber besser iostream.h bzw. fstream.h

13.2.stdlib.h

enthält 5 verschiedene Gruppen von functions:

conversion: (zwischen binären Zahlen und ASCII Darstellungen)

atof, atoi, atol, ecvt, fcvt, itoa, ltoa, strtod, strtol, strtoul, ultoa

dynamic memory allocation/deallocation:

calloc, free, malloc, realloc

process control and environment variables:

abort, atexit, exit, getenv, putenv, system

sorting and searching:

bsearch, lfind, lsearch, qsort, swab

mathematical operations:

abs, div, labs, ldiv

13.3.string.h

functions um strings (arrays of char) zu manipulieren: (z.B. kopieren, einfügen, suchen, vergleichen etc.)

13.4.time.h

Zeit und Datums Funktionen.

13.5.math.h

Alle Winkelfunktionen arbeiten mit Bogenmass (Radians).

abs	Return absolute value of integer parameter
acos	Calculate arccosine
asin	Calculate arcsine

atan	Calculate arctangent
atan2	Calculate arctangent, 2 parameters
atof	Convert string to double
ceil	Return the smallest integer that is greater or equal to x
cos	Calculate cosine
cosh	Calculate hyperbolic cosine
exp	Calculate exponential
fabs	Return absolute value of floating-point
floor	Round down value
fmod	Return remainder of floating point division
frexp	Get mantissa and exponent of floating-point value
labs	Return absolute value of long integer parameter
ldexp	Get floating-point value from mantissa and exponent
log	Calculate natural logarithm
log10	Calculate logarithm base 10
modf	Spli floating-point value into fractional and integer parts
z=pow(x,y)	Calculate numeric power: $z = x^y$
sin	Calculate sine
sinh	Calculate hyperbolic sine
sqrt	Calculate square root
tan	Calculate tangent
tanh	Calculate hyperbolic tangent

13.6.iostream.h und fstream.h

enthält die cin und cout routinen, samt Ergänzungen für file IO und Formattierung etc.

Stichwortverzeichnis