6. Funktionen

6.1.Definition

Eine Funktion ist, vereinfacht ausgedrückt, eine Zusammenfassung von mehreren Anweisungen unter einem bestimmten Namen. Sie können sich eine Funktion als eine Art Teilprogramm (Unterprogramm) innerhalb eines beliebig komplexen Gesamtprogramms vorstellen.

Funktionen werden hauptsächlich aus zwei Gründen eingesetzt:

- Öfters benötigte Sequenzen von Anweisungen müssen nur einmal geschrieben werden und können dann auf relativ einfache Art von verschiedenen Stellen im Programm aus ausgeführt werden.
- Erst mithilfe von Funktionen ist es möglich, ein größeres Programm in kleine, logische Teile zu unterteilen. Diese Unterteilung erleichtert den Überblick über das Gesamtprogramm und damit (auch ganz wichtig!) die Wartbarkeit. Außerdem lassen sich einzelne Funktionen leichter testen als ein großes Gesamtprogramm.

Eine Funktion hat immer denselben Aufbau:

Beispiel 1

```
returnTyp funktionsName([parameter]) {
   lokale Variablen;
   Anweisungen;
   return([returnWert]);
}
```

Die Eingaben in den eckigen Klammern sind optional.

6.1.1. Funktionsdeklaration

Es gibt Unterschied zwischen Funktionsdeklaration und Funktionsdefinition. Die Definition einer Funktion umfasst ihren Kopf und Körper – das eigentliche Programmcode. Die Deklaration einer Funktion ist nur ihr Kopf:

Beispiel 2

```
returnTyp funktionsName([parameter]);
```

Damit kann man eine noch nicht definierte Funktion deklarieren und auf diese Weise dem Compiler sagen, dass es sie gibt und welche Parameter und Ergebnistyp sie hat. Wenn das man das nicht macht, gibt der Compiler einen Fehler heraus – er kennt die Funktion nicht, obwohl sie vielleicht später im Code definiert wird.

6.1.2. Funktionsdefinition

Bis jetzt haben wir alle unsere Anweisungen innerhalb der main-Funktion geschrieben. Für die Anweisungen in allen anderen Funktionen gelten daher dieselben Regeln.

- Man kann Variablen definieren (siehe auch unten bei den Gültigkeitsbereichen)
- Man kann Anweisungen schreiben
- Man kann andere Funktionen aufrufen
- aber: ***Achtung!*** Man darf innerhalb einer Funktion keine andere Funktionen schreiben (entweder deklarieren, noch definieren!)

Hier ein Beispiel für eine Funktion:

Beispiel 3

```
#include <iostream.h>

float berechneKegelvolumen(float r,float h)
{
   const float PI=3.14f;
   float g=PI*r*r;
   float v=(g*h)/3;
   return v;
};

void main()
{
   float radius;
   float hoehe;
   cout<<"Bitte geben Sie den Radius ein: ";
   cin>>radius;
   cout<<"Bitte geben Sie die Hoehe ein: ";
   cin>>hoehe;
   cout<<"Das Volumen betraegt:
"<<berechneKegelvolumen(radius,hoehe)<<endl;
};
```

6.2.Aufruf

Der Aufruf einer Funktion ist sehr einfach: man schreibt an allen Stellen im Code, wo die Funktion aufgerufen werden sollte, den Kopf der Funktion, allerdings mit folgenden Veränderungen:

Kopf der Funktion

Beispiel 4

```
float berechneKegelvolumen(float r,float h)
```

Aufruf der Funktion:

Beispiel 5

```
float ergebnis = berechneKegelvolumen(ein_r,ein_h)
```

Also das Ergebnis der Funktion wird einer Variable zugewiesen, die denselben Typ hat, wie das Ergebnistyp der Funktion. Ausserdem werden als Parameter schon initialisierte Variablen oder Konstanten weitergegeben.

6.3.Parameter

Bei der Deklaration der Funktion muss man mindestens den Typ der Parameter angeben:

Beispiel 6

```
void PrintHeader(int);
```

In der Definition dagegen muss man für jeden Parameter auch einen Parameternamen angeben. Dieser Name dient dann später im Körper der Funktion, um den Parameterwert zu erreichen. Die Definition sieht dann also so aus:

Beispiel 7

```
void PrintHeader(int pageNum);
```

6.3.1. called-by-value

Bei einem Parameter vom Typ *called-by-value* erhält die aufgerufene Funktion nur eine ***Kopie*** des übergebenen Wertes. Daraus folgt, dass Änderung des Parameters zwar innerhalb der Funktion erlaubt sind, diese aber auch nur auf die Kopie des Wertes wirken. Wird die Funktion wieder verlassen, so hat sich am Wert des übergebenen Parameters nichts verändert. Die Definition eines *called-by-value* Parameters erfolgt in der Art, dass innerhalb der Parameterklammer der Funktion lediglich der Datentyp und Name des entsprechenden Parameters angegeben wird. Im Beispiel wird am Ende der Funktion der Wert des Parameters *pageNum* zwar immer auf 99 gesetzt, jedoch wirkt sich dies nicht weiter in *main(...)* aus. Beim Aufruf der Funktion können Sie für einen *called-by-value* Parameter entweder eine Variable (erster Aufruf rechts) oder aber eine Konstante (zweiter Aufruf rechts) an die Funktion übergeben.

Beispiel 8

```
...
// Funktionsdeklaration
void PrintHeader (int pageNum);
....
int main ( )
{
    int pNum = 1;   // pNum initialisieren
    PrintHeader(pNum); // Funktion aufrufen
    ...             // pNum hier immer noch 1
    PrintHeader(10);   // fixe Seitennummer
```

```
}
// Funktionsdefinition
void PrintHeader(int pageNum)
{
   cout << "Seiten-Nummer: " << pNum << endl;
   pageNum = 99;    // Parameter verändern
}
```

6.3.2. Referenzparameter

Referenz-Funktionsparameter ermöglichen Funktionen, an sie übergebene Parameter dauerhaft zu verändern, so dass die innerhalb der Funktion durchgeführte Änderung an den Parametern auch nach dem Verlassen der Funktion noch gültig ist.

Die Referenz ist letztendlich im Prinzip nichts anderes als ein anderer Bezeichner für ein bereits bestehendes Datum. Um einen Funktionsparameter als Referenzparameter zu kennzeichnen, wird bei der Deklaration und der Definition der Funktion nach dem Datentyp des Parameters der Operator & angefügt. Beim Aufruf der Funktion wird ein Referenzparameter wie ein Parameter vom Typ *called-by-value* übergeben.

Beispiel 9

```
// Deklaration
void Swap (short& val1, short& val2);
// Hauptprogramm
int main ( )
{
    ....
    Swap (var1, var2);
    ....
}
// Funktionsdefinition
// Vertauscht die Inhalte zweier Variablen
void Swap (short& val1, short& val2)
{
    short temp = val1;
    val1 = val2;
    val2 = temp;
}
```

6.3.3. Konstante Referenzparameter

In manchen Fällen kann es durchaus sinnvoll sein, dass Parameter die per Referenz übergeben wurden (keine Kopiervorgang der Daten) innerhalb einer Funktion nicht veränderbar sind. Denken Sie an die vorherige Funktion *PrintHeader(...)*, die die aktuelle Seitennummer als Parameter erhalten hat. Niemand würde hier vermuten, dass die Seitennummer innerhalb der *PrintHeader(...)* Funktion verändert wird. Um nun Referenzparameter als nicht-veränderbar innerhalb einer Funktion zu kennzeichnen, wird dem Datentyp des Referenzparameters das Schlüsselwort *const*

vorangestellt. Jeder Versuch, einen als *const* definierten Referenzparameter innerhalb der Funktion zu verändern, führt zu einem Übersetzungsfehler.

Beispiel 10

```
...
// Funktionsdeklaration
void PrintHeader (const int& pageNum);
....
int main ( )
{
    int pNum = 1;   // pNum initialisieren
    PrintHeader(pNum); // Funktion aufrufen
    ...             // pNum hier immer noch 1
    PrintHeader(10);   // fixe Seitennummer
}
// Funktionsdefinition
void PrintHeader(const int& pageNum)
{
   cout << "Seiten-Nummer: " << pNum << endl;
}
```

6.3.4. Übergabe von Feldern

Wie man eindimensionale Felder übergibt, sieht man am besten von dem folgenden Beispiel:

Beispiel 11

```
// Funktionsdeklaration
void DoSomething(short*);
// Felddefinition und Initialisierung
short array[] = {10,20,30,40};
// Hauptprogramm
int main ( )
{
    ....
    DoSomething(array);
    ....
}
// Funktionsdefinition, Feld kann hier in
// der Funktion veraendert werden
void DoSomething (short* ptr)
{
    // Zugriff auf 1. Element
    short var = ptr[0];
    // 3. Element veraendern
    ptr[2] = 10;
}
```

Ein klein wenig komplizierter sieht die Sache bei mehrdimensionalen Feldern aus. Hier müssen Sie dem Compiler etwas unter die Arme greifen damit er die Feldelemente im Speicher auch findet. Bei der Deklaration und der Definition der Funktion müssen Sie die Feldgröße des übergebenen Feldes mit angegeben, damit der Compiler innerhalb der Funktion die Position der einzelnen Elemente korrekt

berechnen kann. Lediglich die Angabe der 'höchsten Dimension' ist optional (siehe Beispiel rechts). Die Übergabe des Feldes beim Aufruf der Funktion bleibt gegenüber eindimensionalen Feldern unverändert, d.h. in der Parameterklammer der Funktion steht auch hier lediglich der Name des zu übergebenden Feldes.

Beispiel 12

```
// Feld definieren
const int ROWS=4;
const int COLUMS=3;
short array[ROWS][COLUMNS];
// Funktionsdeklaration
void PrintVal(short arr[][COLUMNS]);

int main()
{
    // Funktion aufrufen
    PrintVal(array);
    ....
}
// Funktionsdefinition
void PrintVal(short arr[][COLUMNS])
{
    // Zugriff auf Feldelement
    short var = arr[0][2];
}
```

6.4.Rekursive Funktionen

Sehen wir uns jetzt noch einen 'Spezialfall' von Funktionen an. Da innerhalb einer Funktion (bis auf eine Ausnahme) alle Anweisungen erlaubt sind, können Funktionen selbstverständlich wiederum Funktionen aufrufen. Einen Sonderfall stellen hierbei solche Funktionen dar, die sich wieder selbst aufrufen. Solche Funktionen werden auch als rekursive Funktionen bezeichnet. Diese Funktionen benötigen aber immer ein Abbruchkriterium in etwa der folgenden Form um eine Endlos-Schleife zu vermeiden.

Beispiel 13

```
#include <iostream>
using namespace std;
// Funktions-Prototyping
void PrintLine(const short);
// Hauptprogramm
int main ( )
{
    // Funktionsaufruf
    PrintLine(4);
}
// Funktion PrintLine ruft sich selbst auf!
void PrintLine(const short count)
{
    // Sternchen und Zeilenvorschub ausgeben
    for (int index=0; index < count; index++)
```

```
        cout << " *";
    cout << endl;
    // Falls mehr als 1 Sternchen ausgegeben,
    if (count != 1)
        // Funktion erneut aufrufen, jetzt
        // jedoch mit einem Sternchen weniger
        PrintLine(count-1);
    // Nach dem letzten Sternchen fertig!
    return;
}
```

6.4.1. Default-Parameter

Man kann für einen oder mehrere Parameter auch default-Werte eingeben. Dies geschieht in der Definition der Funktion, mit Hilfe von Anfangswerten:

Beispiel 14

```
void myFunction ( int index, char name, bool done = true) ...
```

Hier muss man aber ein paar Regeln beachten:

- Die default-Parameter stehen immer am Ende der Parameter-Liste.
- Beim Aufruf der Funktion kann man ganz normal für alle Parameter Werte oder Variablen weitergeben; oder aber die default-Parameter auslassen – dabei gilt, dass man die letzten n Parameter auslassen kann. Also von insgesamt z.B. 4 default-Parameter darf man alle 4, die letzten 3, 2, 1 oder keine auslassen, aber nicht nur den zweiten oder nur den dritten.

Beispiel 15

```
...
void myFunction ( int index = 0, char name = ´a´, bool done = true) {
      hier passiert etwas...
}
void main () {
      ...
      myFunction(2, ´b´, true);
      myFunction(2, ´b´);
      myFunction(2);
      myFunction();
}
```

6.5.Gültigkeitsbereiche

Variablen, die innerhalb einer Funktion definiert werden, heißen lokale Variablen. Auf lokale Variablen kann nur innerhalb der Funktion zugegriffen werden, die die Variablen definiert. Andere Funktionen können auf diese Variablen nicht zugreifen. Beispiel:

Beispiel 16

```
void a()
{
```

```
  int x;
}

void b()
{
  int x;
}
```

Nachdem lokale Variablen nur innerhalb einer Funktion sichtbar sind, können Sie denselben Variablennamen in unterschiedlichen Funktionen verwenden. Im obigen Beispiel wird eine Variable x in der Funktion *a()* und in der Funktion *b()* definiert. Nachdem es sich hierbei um lokale Variablen handelt und die Variablen nur in den jeweiligen Funktionen gelten, würde obiger Code einwandfrei kompilieren. Würden beide Variablendefinitionen in derselben Funktion stehen, würde dies zu einem Fehler führen, weil Variablennamen in einem Gültigkeitsbereich nur ein einziges Mal vergeben werden dürfen.

Globale Variablen werden außerhalb einer Funktion definiert.

Beispiel 17

```
int x;

int main()
{
}
```

Im obigen Code-Beispiel wird eine Variable *x* als globale Variable definiert - die Definition steht außerhalb jeder Funktion. Während lokale Variablen nur einen begrenzten Gültigkeitsbereich besitzen, der sich nur über eine Funktion erstreckt, existieren globale Variablen im gesamten Programm. Das heißt, jede Funktion kann auf die globale Variable zugreifen, und zwar jederzeit. Globale Variablen existieren vom Programmstart an bis zum Ende, während lokale Variablen immer nur während der Ausführung der Funktion existieren.

Lokale und globale Variablen lassen sich leicht unterscheiden: Lokale Variablen werden immer innerhalb geschweifter Klammern definiert. Ob diese Klammern einer Funktionsdefinition angehören oder nicht spielt hierbei keine Rolle. Sie können geschweifte Klammern beliebig setzen, um Gültigkeitsbereiche für Variablen festzulegen.

Beispiel 18

```
int main()
{
  {
    int y = 10;
  }
  {
```

```
    int y = 20;
  }
}
```

Im obigen Beispiel werden innerhalb der Funktion *main()* zwei Variablen mit dem Namen *y* definiert. Dies ist nur deswegen möglich, weil die Variablen in unterschiedlichen Gültigkeitsbereichen definiert sind. Diese Gültigkeitsbereiche sind durch willkürliche Verwendung der geschweiften Klammern festgelegt worden, die wie Sie sehen nicht unbedingt einen Anweisungsblock einer Funktion oder einer Kontrollstruktur festlegen müssen.

6.6.inline – Funktionen

Häufig braucht man auch teilweise sehr kleine Funktionen. Diese lassen sich zwar problemlos programmieren, haben aber den Nachteil, daß das Programm für den Aufruf der Funktion schon Zeit verbraucht - gerade bei häufig benutzten Funktion sicherlich nicht wünschenswert, da spürbare Performanzverluste auftreten können.

Um dieses Problem zu umgehen, kann man verschiedene Ansätze wählen. Zum einen könnte man darauf verzichten, die Funktion zu schreiben und doch jedesmal den gleichen Programmteil hinschreiben - doch das ist doch sehr aufwendig und zeitverschwenderisch.

Eine wesentlich elegantere Möglichkeit ist es, dem Compiler anzugeben, daß er versuchen soll, die Funktion nicht wie eine normale Funktion zu behandeln, sondern direkt in den laufenden Code einzubauen. Damit spart man sich die Schreibarbeit, das Programm ist lesbarer, aber das Ergebnis ist dasselbe, als ob man die entsprechenden Befehle jedesmal hingeschrieben hätte. Die zugehörige Anweisung geht über das Schlüsselwort inline, am Beispiel mit dem Maximum (wo es sehr sinnvoll ist):

Beispiel 19

```
inline int maximum(int a, int b) {
    return (a>b)?a:b;
}
```

Dabei sollte man allerdings ein paar Dinge beachten:

- Das Programm wird größer, falls die Funktion mehr Code erzeugt, als durch den Aufruf gespart wird (und das kann sehr schnell der Fall sein)
- Der Geschwindigkeitsgewinn macht sich nur bei sehr kleinen und häufig genutzten Funktionen bemerkbar
- Sowohl die Tatsache, daß das Schlüsselwort genutzt wird, als auch die Tatsache, daß es fehlt, geben keinen Aufschluß darüber, was der Compiler wirklich macht,

da inline nur eine Empfehlung darstellt und umgekehrt einige Compiler im Rahmen einer Optimierung Funktionen selbständig inline compilieren.

6.7.Aufgaben

6.7.1. Schreiben Sie eine Taschenrechner-Erweiterung: der Benutzer kann zwischen verschiedenen Operationen (sin, cos, tan, log, quadratwurzel, potenzen usw.) auswählen und diese mit verschiedenen Parametern aufrufen. Implementieren Sie jede Operation mit Hilfe von einer Funktion mit den nötigen Parametern (achten Sie auf called-by-value und Referenzparametern). Jede kurze Funktion soll als ***inline*** definiert werden. Eine Beispiel-Interaktion zwischen Benutzer und Programm:

```
Bitte geben Sie die gewünsche Operation ein:
1: sin
2: cos
3: ln
4: potenz
5: quadratwurzel
5
Bitte geben Sie die Zahl ein, für die Quadratwurzel berechnet werden soll:
9
Quadratwurzel von 9 ist: 3.
Möchten Sie noch eine Operation durchführen (J für Ja, N für Nein): N
Danke für das Benutzen des Programms!
```

Sie können die Operationen selbst frei auswählen: Sie können es für verschiedene Operationen und/oder Formeln aus Mathematik, Physik, Informatik usw. implementieren.

6.7.2. Schreiben Sie eine rekursive Funktion, die rückwärts von 100 bis 1 zählt.

6.7.3. Schreiben Sie eine rekursive Funktion zur Berechnung von den Fibonacci-Zahlen.

Projekt. Schreiben Sie wieder das Tic-Tac-Toe Programm. Die Einzelheiten werden in der Übung besprochen.