C++
Minnnni-eBook

Version 2.01
Autor-Email: dafrewa@lycos.de
Webseite: http://www.freengfoad.de.vu

Dieses Tutorial ist eher für Anfänger geeignet und beschäftigt sich mit den Grundlagen der Programmiersprache C++. Wer schon einmal ein paar Zeilen in C oder C++ geschrieben und kompiliert hat, kann das erste Kapitel überspringen. Dieses Tutorial erhebt kein Anspruch auf Vollständigkeit.

Inhaltsverzeichnis

1. Was brauche ich um anzufangen?

1a. Einen Compiler

Zunächst einmal wird ein Compiler benötigt. Dies ist ein Programm mit den man C++ Code in eine ausführbare Datei (also als .exe für Executive = Ausführbar) umwandelt. Diesen Vorgang nennt man compilieren oder erstellen und geht nur in eine Richtung, das heißt vom Code zum Programm. Wenn man also ein Programm ändern möchte, muss man dies in seinen Codezeilen tun und nochmals compilieren.

Beides, der C++ Code und das ausführbare Programm ist unabhängig voneinander. Die Codedateien sind also das was der Programmierer eingibt und lesen kann, aber der Computer nicht. Und das fertige Programm kann nur der Computer verstehen aber macht genau das was der Programmierer als C++ Code eingegeben hat.

Der Code befindet sich in reiner Textform und dieser kann auch jederzeit geändert werden. Die ausführbare fertig compilierte Datei kann nicht geändert werden und kann zwar gelesen werden, allerdings dann nur als sinnloses Zeichenchaos wie etwa ^12r#df+_<l2tgg

Die Bedienung des Compilers ist recht unterschiedlich je nach Ausstattung, in älteren meist kostenlosen muss man dies per Hand eingeben:

    [Schlüsselwort fürs kompilieren] <hauptdatei.cpp>

Das jeweilige Schlüsselwort hängt vom Compiler ab und müsste in der Beschreibung des jeweiligen Compilers stehen. Bei neueren bedienerfreundlicheren und besseren Compilern gibts dafür einen einfaches Menüpunkt und der Code ist noch grafisch etwas belegt.

Man kann dieses Tutorial aber auch ohne Compiler lesen.

Zur Windows-Programmierung braucht man einen etwas besseren Compiler und die Hauptfunktion sieht dort etwas anders aus. Windows-Programmierung kommt jedoch hier nicht vor.

Um beispielsweise ein Spiel zu entwickeln, reicht C++ allein nicht aus. Dazu fehlen einem dann noch Kenntnisse über die Windows-Programmierung (Fenster, Menüs, Kontextmenüs, Dialogboxen usw.) und Kenntnisse über die Grafikprogrammierung, entweder mit DirectX oder OpenGL.

Falls du nun einen Compiler hast (gibts auch im Web, für dieses Tutorial vollkommen ausreichend, wenn du nach “C++ Compiler” suchst), installiere ihn nun. Dann lege ein Verzeichnis an, welches dann die Arbeitsverzeichnisse enthält. Also zum Beispiel C:\Cplusplus\ mit einzelnen Verzeichnissen für die Quelltexte.
Für die ersten Programme reicht dann eine Quelltextdatei aus, also main.cpp oder auch test.cpp oder irgendwas anderes deiner Wahl. Für größere Programme teilt man den Code in mehrere .cpp-Dateien auf.

1b. Bibliotheken

Bibliotheken haben meist die Endung .LIB oder .DLL. Diese sind beim Compiler mit dabei und meist im Bibliotheken-Verzeichnis. Bibliotheken enthalten eine Reihe Funktionen, die man in sein Programm mit einbinden und nutzen kann, so brauchtman nicht neu schreiben, was schon da ist. Man kann auch selbst Bibliotheken erstellen wenn man eine Reihe von ähnlichen Funktionen auch bei anderen Programmen/Spielen brauchen könnte. Bibliotheken werden in diesen Tutorial nicht verwendet.

Wenn ein Programm beispielsweise ausgibt: “Fehler: Audio.DLL wurde nicht gefunden!” heisst dies, dass dem Programm ein paar Funktionen fehlen und somit nur eingeschränkt laufen oder garnicht.

1c. Schreibweisen und Syntax beachten

Syntax ist der Aufbau des Codes. Nur wenn dieser richtig aufgebaut ist, kann man ihn compilieren. Das heißt unter anderen, dass jede Anweisung mit einen Simikolon zu beenden ist. Danach fängt man am besten eine neue Zeile an aber dies gehört nicht zum Syntax sondern zur besseren Übersicht. Man kann auch alles hintereinander schreiben ohne Leerzeichen und Tab´s, aber das ist unübersichtlich.

Der Compiler ist case-sensitive, das heißt er unterscheidet Groß- und Kleinschreibung. Versucht man also etwas wie "energie" zu ändern und hat diese an anderer Stelle "Energie" genannt, so kann das nicht gehen, da "energie" nicht "Energie" ist. “A” und “a” ist also nicht das gleiche und bei jeden anderen Buchstaben genauso.

Wenn hier etwas in Anführungszeichen steht, ist nur der Inhalt gemeint. Die Anführungszeichen haben also außer bei der Textausgabe mit “cout” nichts zu bedeuten.

Im diesen eBook kommen auch Wörter vor, die vielleicht einige nicht verstehen. Darum hier die 3 wichtigsten erklärt:

Definition = Erklärung/Erläuterung von etwas
Deklaration = Etwas neu einrichten
Initialisierung = Etwas mit einen Startwert belegen

In den C++ Codedateien (meist mit der Endung .cpp) wird man Kommentare machen müssen. Kommentare sind dazu da, irgendwas in der Quelltextdatei reinzuschreiben, um etwas zu erklären damit man es auch noch in ein paar Wochen versteht, was eine Funktion macht. Kommentare werden nicht mitcompiliert. Diese Stellen werden einfach übersprungen.

Wie kann man nun kommentieren? Dazu gibt es 2 Möglichkeiten. Davor noch der Hinweis, dass in diesen Tutorial Code generell eingerückt ist, um diesen vom übrigen Schriftbild zu trennen.

Möglichkeit 1 eines Kommentares:

```
Code....
Code...     // Kommentar, alles ab // bis Ende der Zeile ist ein Kommentar
Code....
// auch ohne Code vorne, ab den beiden Strichen ists ein Kommentar und
// gehört nicht zum Code
```

Möglichkeit 2 eines Kommentares:

```
Code...
Code...
/* Alles Zwischen /* und */ ist ein Kommentar, auch über ganze
Zeilen können auskommentiert werden. Es lässt sich so
genausogut auch Code auskommentieren.  */
Code...
```

1d. Geschichte von C und C++

C++ ist eine Weiterentwicklung von C und eine der meist verbreitestens und benutzen Programmier-sprachen. C wurde in den 70er Jahren entwickelt.

Und was nimmt man?
Ob man zuerst C lernt oder gleich mit C++ anfängt ist egal. Da C++ aber auch C aufbaut, fängt man am besten mit C++ an, da man dieses wohl auch eher braucht. Außerdem ist C manchmal nicht ganz so einfach zu nutzen, da es dort keine Klassen gibt, man kann den Speicher nicht allokieren und es gibt keine Funktionsprototypen und somit ist auch kein Überladen von Funktionen möglich.

Um es anders zu sagen, man lernt nicht erst wie man eine Schallplatte auflegt, wenn es CD´s gibt, ausser man möchte dies unbedingt.

1e. Objektorientierte Sprache

C++ ist eine objektorientierte Sprache. Das heißt, dass es besser strukturiert ist. Man nehme also als Beispiel ein C++ Code, der über mehrere 1000 Zeilen geht. Davon sind dann 240 für die eine Sache, 120 für eine andere, 300 Zeilen nur für Audio usw. Nun kann man einzelne Objekte auch in anderen Programmen verwenden, indem man das Audio-Objekt einfach dem neuen Projekt hinzufügt. Schon kann man dessen Funktionen nutzen, ohne den Code des Objektes genau kennen zu müssen.

Was sind nun Objekte im Klartext? Das können sein: Laderoutinen für bestimmte Formate, ein Spieler und dessen Bewegungen, Audioroutinen zum laden und abspielen von Sounds, Objekte für den Input, also für Maus/Tastatur, Objekte für die 3D-Kamera und andere.

Und Codetechnisch? Codetechnisch heißt dies, dass man seine Objekte in Klassen unterbringen kann, und der Inhalt dieser Klasse eigentlich nicht interessiert. Hauptsache man kennt die öffentlichen Funktionen. Denn dann kann man diese nutzen und auch wieder in neueren Projekten verwenden. Denn wie oben schon erwähnt braucht man nicht das Rad neu zu erfinden (sonst könnte man gleich mit einen Betriebssystem anfangen, was etwas dauern könnte).

Und Codetechnisch die Zweite: Um es mal auch verständlich für die auszudrücken, die nicht wissen, was eine Klasse ist:

Man muss nicht wissen wie ein Auto oder Fernseher oder gar ein Flugzeug funktioniert oder von innen aussieht. Man muss nur die öffentlichen Funktionen “An”, “Aus”, “Schneller”, “Bremsen” und diese Sachen wissen. Bei einen neuen Flugzeug wird man sicherlich nicht alles neu entwickeln, denn das würde Jahrzehnte dauern, sondern bestimmte Objekte wiederverwenden. Oder anders gesagt: Wenn man nur den Benzinverbrauch eines Autos verringern möchte, bastelt man nur am Motor, und nicht an der Lenkung oder Scheibenwischer. Klassen werden später noch erklärt.

1f. Wozu man C++ braucht

Es gibt noch andere Programmiersprachen, davon ist C++ jedoch einer der verbreitesten und schnellsten. Die meisten Anwendungen sind mit C++ geschrieben. Dazu zählen Grafikprogramme, Spiele, Browser, Textverarbeitung und andere. Es gibt heute auch schon viele Module und Bibliotheken, die einen viel Arbeit abnehmen, um nicht ganz von vorne anzufangen. Allerdings muss man darauf teilweise Lizenzgebühren zahlen, so auch für bestimmte Datei-Formate, falls man dafür Laderoutinen zur Verfügung stellt (nicht für das bloße Abspielen). Demzufolge werden kostenlose Super-Mega-All-Media-Player nicht unterstützt, auch wenns dafür alle möglichen Laderoutinen schon gibt.

2. Funktionen

2a. Eine allgemeine Beschreibung

Funktionen erfüllen irgendeine Aufgabe, Beispiele:

Produkt ausrechnen, Sprite Darstellen, Menü zeigen, Mauszeiger verstecken, Programm beenden, Datei öffnen, Datei speichern, Song abspielen, Bild verkleinern, Bild löschen, Figur springen lassen, Auto starten, Tür öffnen, Fenster öffnen, Licht anmachen, CD brennen, PC runterfahren, Anwednung starten, Online sich einwählen, Ball fangen, Essen, Würfel malen, Farben ändern, Text verschlüsseln und viele viele mehr.

Funktionen beginnen mit einen Rückgabetyp, es folgt ein Name für die Funktion und schließlich folgt der Funktionsrumpf. Die geschweiften Klammern { und } öffnen und schließen den Funktionsrumpf.. Das ganz sieht dann etwa so aus:

```
Rückgabetyp Funktionsname()
{
Inhalt der Funktion oder
        was die Funktion so macht
}
```

2b. Die Funktion Main

Es gibt eine Hauptfunktion. Diese muss “main” (kleingeschrieben) heissen. Nur diese startet von alleine. Alle anderen Funktionen müssen aufgerufen werden.

Erstmal folgendes Beispiel, daran wird dann Zeile für Zeile erklärt.

```
1:      #include <iostream.h>
2:
3:      void main()
4:      {
5:      int aepfel = 4;
6:      int eingabe;
7:      cout << "Wieviel Äpfel essen?\n";
8:      cin >> eingabe;
9:      aepfel = aepfel - eingabe;
10:     cout << aepfel << endl;
11:     }
```

Die Nummern mit den Doppelpukten sind nur da, um hier die Zeilen zu erklären. Falls du einen Compiler installiert hast, übernehme jetzt diese Zeilen in eine neue .cpp-Datei (kann mit jeden Texteditor erstellt werden, falls im Compiler keiner vorhanden ist). Dann compiliere die .cpp-Datei und du hast dein erstes selbstgemachtes Programm.

Nun zur Erklärung der einzelnen Zeilen:

Zeile 1:
“#include” heisst dass man eine weitere Codedatei einbindet. Wenn dies hier also die Datei “main.cpp” ist, dann wird noch “iostream.h” und “iostream.cpp” eingebunden (man braucht nur immer die Headerdatei, also mit der Endung ".h" einbinden, welche dann die dazugehörige .cpp-Datei einbindet). In dieser Datei sind nun weitere Funktionen, hier im Programm für Eingabe und Ausgabe.

Zeile 2 ist zu besseren Übersicht völlig leer.

Zeile 3:
Hier beginnt eine Funktion, dieses Programm hier hat nur diese eine. Es ist die Funktion “main”. Diese muss so heissen, für alle anderen kann man den Namen frei wählen und alle anderen werden dann aufgerufen von einer Stelle in “main” oder von einer anderen Funktion heraus. Das Wort "void" davor steht für den Rückgabetyp. Erstmal wird hier nur “void” verwendet und damit weiß der Compiler, dass dies eine Funktion ohne Rückgabewert ist, also eine Funktion, die keine Werte zurückgibt.

Nach dem Funktionsnamen “main” stehen 2 runde Klammern, die können hier leer bleiben und sind später für Parameter gedacht - das heißt wenn man der Funktion irgendwas übergeben will, zum Beispiel 2 Zahlen, einen Text oder ähnliches was diese Funktion dann benutzt.

Zeile 4 und Zeile 11:
Dies sind 2 geschwungene Klammern für den Funktionsrumpf. Den Inhalt der Funktion also. Dieser kann beliebig lang sein.

Zeile 5 bis 10:
Dies ist der Inhalt der Funktion. Dass, was das Programm macht.

Zeile 5 und Zeile 6: In der Funktion selbst werden hier zuerst 2 Variablen deklariert mit den Namen "eingabe" und "aepfel". Variablen bestehen zuerst aus den Typ der Variable, dann den Namen. Variablen sind dazu da, Werte zu speichern, das können Texte sein oder Zahlen, die genutzt werden. Den Typ der Variablen (für was ist die Variable bestimmt) braucht man nur einmal ganz zum Anfang davorschreiben nämlich bei der Deklaration.

Danach kann man die Variable einach ändern indem man

```
Variablenname = Neuer Wert;
Aepfelzahl = 5;
Seiten = 52;
```

schreibt. Bei der Deklaration kann man entweder einen Wert zuweisen oder später dies irgendwann nachholen. Mehr dazu im Variablenkapitel.

Zeile 7: Dann wird die Frage ausgegeben. “cout” und “cin” sind in “iostream.cpp” definiert, deswegen musste auch oben diese Datei eingebunden werden. Dort sind die Funktionen zu “cin” und “cout”, die man braucht um Strings/Variablen einzulesen und auszugeben. Variablen mit Zahlen werden ohne Anführungszeichen geschrieben. Strings, also nur Text, werden mit Anführungszeichen geschrieben (wichtig).

Nachdem man eingegeben hat, wieviel man essen möchte, wird der Rest ausgerechnet und das dann wieder mit cout ausgegeben.

Zeile 10 am Ende: \n (im Textstring) und << endl sind Möglichkeiten für einen Zeilenumbruch (Sprung in nächste Zeile).

Weiterer solcher In-Text-Zeichen sind:

\t = Tabulator
\\ = Backslash
\a = einfaches Beep

2c. “cout” und “cin”

Dazu ein paar Beispiele die irgendwo stehen können innerhalb einer Funktion. Die Erklärungen folgen in den Kommentaren dahinter:

```
cout << aepfel << endl;              // Ausgabe des Wertes in Variable namens "aepfel"
cout << "Äpfel: " <<  apefel << endl;
                                     // Ausgabe des Wortes "Äpfel" und folgend den Wert in Variable "aepfel"
cout << "Test";                     // Ausgabe des Wortes "Test"
cout << aepfel << "und" << birnen << endl;
                                     // Ausgabe zweier Variablen - dazwischen das Wort “und”
cin >> aepfel;                       // Wartet auf Eingabe und ENTER, speichert dies dann in genannter Variable
```

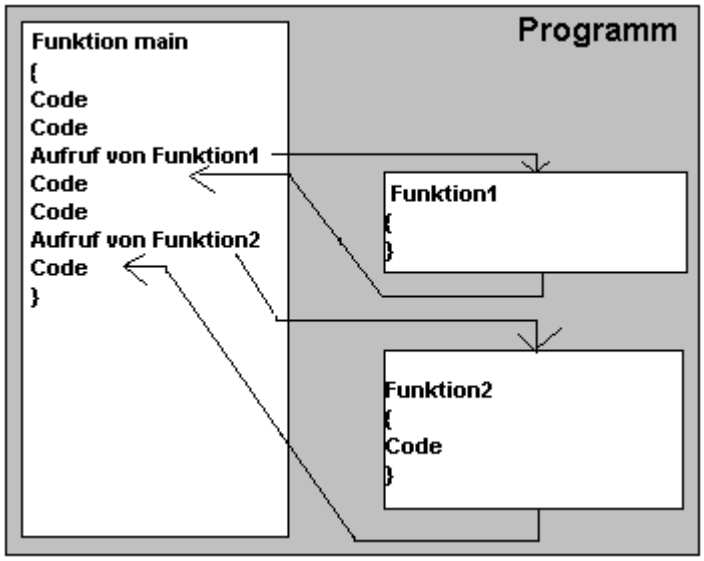
Nun kurz eine Darstellung von der Ausgabe im DOS-Fenster:
Es sei die Variable "aepfel" = 10 und "birnen" = 5

Code	Ausgabe
cout << aepfel << endl;	10
cout << "Äpfel: " << apefel << endl;	Äpfel: 10
cout << "Test\n";	Test
cout << aepfel << "und" << birnen << endl;	10 und 5

“cout” und “cin” braucht man nur beim jetzigen Wissensstand. Später bei Windowsprogramm gibt es da schon weitaus bessere Möglichkeiten.

2d. Die Wege des Programms bei mehreren Funktionen

Nachdem eine Funktion fertig abgearbeitet wurde, geht das Programm wieder dahin zurück wo es herkam. Dazu ein kleines Schaubild:



Die Pfeile zeigen den Weg des Programmablaufs, dass immer nur an einer Stelle ist.

Wie groß eine einzelne Funktion ist, hängt davon ab wie man es am besten ordnet. Man kann auch das ganze Programm in nur 10 Funktionen schreiben. Das wär aber unübersichtlich. Das Programm muss auch logisch geordnet sein. Im oberen Schaubild könnte das Programm zum Beispiel folgendes machen:

1. Frage nach 2 Zahlen (in main)
2. Übergebe diese beiden Zahlen an Funktion1, die dafür da ist, das Produkt auszurechnen.
3. Rechne (in Funktion1)
4. Übernehme das Produkt von Funktion1
5. Übergebe die beiden Zahlen an Funktion2, die dafür da ist, die Summe auszurechnen.
6. Rechne (in Funktion2)
7. Übernehme die Summe von Funktion2
8. Gebe beides aus (in main)

Funktion1 und Funktion2 machen hierbei was eigenständiges, nämlich übernehmen 2 Zahlen, rechnen etwas aus und übergeben das Ergebnis.

2e. Ein Beispiel aus der Praxis: Ein Flugzeug

Was macht ein Flugzeug? Welches sind die Grundfunktionen? ---> Funktionen schreiben
Wann macht es was? --> Verbindungen zwischen den Funktionen erschaffen, Aufrufen von weiteren Funktionen

Programmdatei in “flugzeug.cpp” - die Hauptdatei und die auch die einzigste:

// Es folgen die Funktionsprototypen, damit dass Programm weiß, dass es diese Funktionen gibt

```
void main();
void starten();
void landen();
void parken();
void fliegen_bis_zum_ziel();
```

// Es folgt die Hauptfunktion

```
void main()
{
starten();                  // zuerst starten
fliegen_bis_zum_ziel();  // nachdem Flugzeug nun gestartet ist, fliegt es
landen();                   // und landet dann wenn die “fliege-Funktion” beendet wurde
}
```

// Es folgen nun die einzelnen Funktionen

```
void starten()      // nachdem diese Funktion ausgeführt wurde,
                    // geht der Programmverlauf wieder in main zurück
{
....
}

void landen()
{
....
parken();
...
}

void fliegen_bis_zum_ziel()
{
....
}

void parken()
{
....
}
```

Das Programm kann immer nur an einer Stelle durchlaufen werden. Wie ist nun der Programmverlauf?

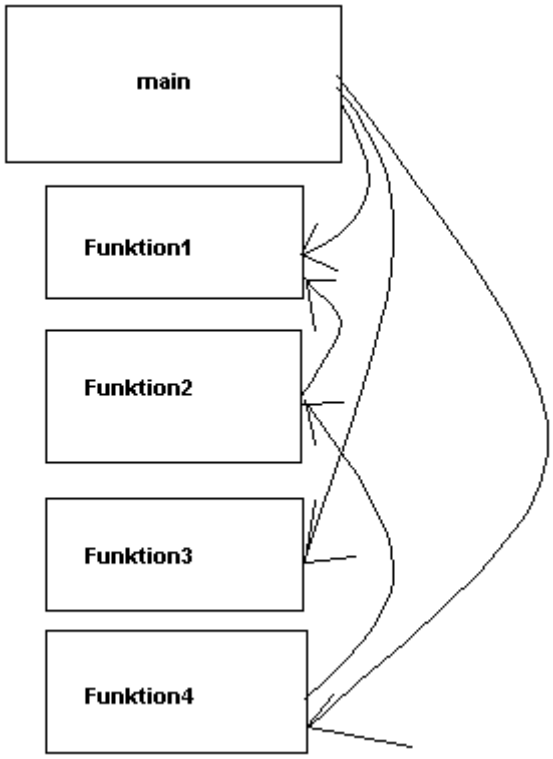
1. "main" (bis "starten" aufgerufen wird)
2. "starten" (bis Funktionsende)
3. wieder zurück nach "main", dort steht "fliegen_bis_zum_ziel" als nächstes
4. nach "fligen_bis_zum_ziel" wieder zurück in "main", dann "landen"
5. In "landen" gehts zwischendurch zur Funktion "parken",
6, nachdem "parken" abgearbeitet wurde gehts wieder weiter in "landen" an der Stelle zurück
da die Funktion noch nicht fertig war

2f. Der Funktionsprototyp

Warum sollte man immer Funktionsprototypen benutzen?
Wie in diesen Flugzeugdemo oben zu sehn ist, steht ein sogenannter Prototyp der Funktion schon ganz oben im Programm. Zu jeder Funktion gibt es einen Funktionsprototyp. Dieser ist nicht zwingend erforderlich aber höchst praktisch. Man schreibt diesen am besten ganz oben in der Datei oder im Header. Sinn dieses Funktionsprototypen ist der, dass wenn im Programm an irgendeiner Stelle diese Funktion aufgerufen wird (also gebraucht wird), bereits vorher entweder die Funktion oder der Funktionsprototyp schon aufgetaucht sein muss, sonst kennt er diese Funktion nicht und bricht ab. Der Compiler liest den Code von oben nach unten.

Aufbau dieses Funktionsprototypes:

```
Rückgabetyp Funktionsname(mögliche Parameter);
// gegenüber der Funktion der selbe Kopf aber
// kein Funktionsrumpf und oben zusätzlich ein Simikolon
```



Und warum schreibt man dann die Funktionen die main braucht, nicht zuerst?
Ganz einfach: Das geht hier zwar aber normalerweise geht dies nicht, da manche Funktionen andere aufrufen, die dann wieder andere aufrufen usw.. Und die müssen dann wieder an Funktion X ran, und irgendwann kennt eine Funktion eine andere nicht, da es nicht möglich ist, alle so zu ordnen.

Nebenstehenes Schaubild hilft vielleicht etwas mehr, das zu verdeutlichen:

Die Hauptfunktion will nun im Laufe des Programms an den Funktionen 1, 3 und 4 ran. Funktion 4 will jedoch an 2 ran und diese widerum an 1.

Als Ergebnis kann man sagen:
Ohne Prototypen gehts hier nicht.

Neben diesen 1. Grund für Funktionsprototypen gibt es noch 3 andere Gründe für Funktionsprototypen:

Nummer 2: Vorbelegung von Variablen in Funktionsparametern (Standardparameter)
und
Nummer 3: Überladen von Funktionen durch mehrfaches Benennen
und
schließlich
Nummer 4: Inline-Funktionen

Ok, da dies jetzt kein Mensch versteht, der nicht schon C++ -Ahnung hat, wird dies bei geeigneter Zeit Schritt für Schritt und einzeln erklärt (nämlich in 2k., 2l. und 2m.), so dass man nach dem 2. Kapitel fast alles über Funktionen weiß. Aber schon jetzt sei gesagt: Wenn ein Fehler kommt: “Funktion blablasowieso nicht gefunden” und sie ist trotzdem da, dann gibt es mit hoher Warscheinlichkeit keinen Prototyp.

2g. Ein Passwort-Programm:

Nun folgt wieder ein Beispiel, was man auch selbst kompilieren kann. Ein Passwort-Programm.
Hier gibt es 2 Funktionen, einmal die Funktion “main”, die von sich aus startet und einmal eine Funktion namens "Geheimfunktion", die nur aufgerufen wird wenn das Eingegebene dem Passwort entspricht.

```
#include <iostream.h>                          // für cout und cin

void Geheimfunktion();                         // Funktionsprototyp

void main()
{
cout << "Dieses Programm führt eine Geheimfunktion aus.
          Dazu wird ein Passwort benötigt.\n";
int pw = 12345;                                // Das Passwort in einer Variable gespeichert
int was_eingegeben;                            // Variable für Nutzereingabe
cout << "Bitte Passwort eingeben: ";
cin >> was_eingegeben;
if (was_eingegeben == pw)
          {
          Geheimfunktion();                    // Aufruf der Funktion falls Eingabe == Passwort
          }
}

void Geheimfunktion()
{
cout << “\nDas Passwort war richtig.”;
}
```

Zur Erklärung:

_ ist das einzigste Sonderzeichen dass man in Variablennamen und Funktionsnamen verwenden darf. Der Mittelstrich und das Leerzeichen ist nicht erlaubt.

Die Variable beginnt hier mit int, dies steht für integer und heisst, dass dort eine Ganzzahl gespeichert wird. Eine Ganzzahl ist 1, 2, 3 usw. - keine Brüche und keine Kommazahlen.

Ganz oben steht ein Prototyp der Funktion namens.

Im Programm selbst wird erst ein Text ausgegeben. Dann folgt eine Abfrage. Die Zeile mit if kann man in etwa so lesen

```
if (wenn das so ist wie hier steht)
          {
          dann das ausführen, sonst nicht;
          }
```

Wenn das nicht so ist, dann wird der Inhalt von “if” einfach übersprungen. In { und } kann auch eine Variable verändert werden oder man kann wie oben eine Funktion aufrufen. Es können dort auch mehrere Anweisungen stehen oder weitere If-Abfragen oder Schleifen. Dazu mehr in den nächsten Kapiteln.

Bei einen Aufruf einer Funktion/Variablen schreibt man einfach den Funktionsnamen, jedoch nicht den Rückgabetyp, der Rückgabetyp “void” (gibt nichts zurück, drum ist dies hier auch nicht weiter erklärt) wird nur im Prototyp und vor den Funktionsnamen in der Funktion geschrieben, nicht beim Aufrufen.

Nach dem Aufruf der Funktion namens "Geheimfunktion" kehrt der Compiler wieder zu "main" zurück.
Die Funktion könnte auch nur “Geheim” heissen, “Login” oder ähnliches. Das Benennen bleibt den Programmierer überlassen.

Falls du einen Compiler installiert hast, kompiliere dies und seh selbst was das Programm macht.

Der letzte Text wird nur sehr kurz angezeigt. Das liegt daran, dass das Programm danach wieder gleich endet, da nach dieser Anweisung nichts mehr steht. Dem könnte man entgegen wirken, indem man zum Schluss noch zu einer Enter-Eingabe auffordert.

Falls ein Fehler auftritt, woran könnte es liegen?
- Prototyp vergessen?
- irgendein Simikolon vergessen?
- alle Klammern gesetzt?
- “iostream.h” includiert (mit einbezogen)?
- cout oder cin oder die Pfeile hinter cout/cin verwechselt?
- den Typ der Variable/Funktion vergessen vergessen?
- Groß- und Kleinschreibung beachtet?

Alternativ zum Funktionsaufruf in “if” geht allerdings auch folgendes:

```
if (was_eingegeben == pw)
        {
        cout << "\nDas Passwort war richtig.";
        }
```

So spart man die Extrafunktion. Da die Funktion hier nur sehr klein ist, kann man dies machen. Diese sollte hier nur zeigen, wie man sich das in Funktionen aufteilt.

Zur Erinnerung: Das “\n” steht für einen Zeilenumbruch. Falls man probiert, das Programm ohne Zeilenumbruch zu kompilieren, steht der ganze Text hintereinander weg.

2h. Logischer Aufbau von Programmen

Wie schon erwähnt, sollten Programme logisch aufgebaut sein. Hierzu noch ein Beispiel eines Autos. Dies ist nicht zum kompilieren gedacht. Es soll nur zeigen, wie man generell Programme aufbaut.

```
void starten();                 // Prototypen
void fahren();
void bremsen();
void anhalten();
void beschleunigen();

bool bremse = false;            // Variablen vom BOOL-Typ = richtig (true) oder falsch (false)
bool am_ziel = false;           // zu Beginn alles auf false gesetzt
bool fahr_schneller = false;
int tempo = 0;                  // Geschwindigkeit, am Anfang auf 0 gesetzt

void main()
{
starten();
while(1)                        // Endlosschleife von das, was in den Klammern steht
    {
    Auto_fahren();
    if (bremse == true) { bremsen(); }
    if (fahr_schneller == true) { beschleunigen(); }
    if (am_ziel == true) { anhalten(); break }      // mit "break" raus aus der Endlosschleife
    }
}

void starten()
{
...
}

void fahren()
{
...
}

void bremsen()
{
...
}

void anhalten()
{
...
}

void beschleunigen()
{
....
}
```

Statt “if (am_ziel == true)” könnte man auch schreiben “if (am_ziel)”. In solch einen Fall wird true angenommen.
Auf false wird dann bei “if (!am_ziel)” getestet, wobei das Ausrufezeichen für “ist nicht” steht.

2i. Übergabeparameter:

Nochmal eine Sache zum Funktionsaufbau. Eine Funktion sieht in etwa so aus:

```
Rückgabetyp Funktionsname(Übergabeparameter)
{
...
Anweisungen wie die Änderung von Variablen, Schleifen, Aufrufe anderer Funktionen ...
...
}
```

Übergabeparameter sind die Werte in den runden Klammern der Funktion. Dies heisst, dass dort etwas der Funktion übergeben wird, und zwar an die die aufgerufen wird.

Dies kann ein Text/String sein oder ein Wert, den die Funktion braucht. Dazu gleich ein Beispiel mit lokalen Variablen.

```
#include <iostream.h>   // hier hinter ein Simikolon zu setzen ist übrigens ein beliebter Fehler

void Text_ausgeben(int textnummer);              // Prototyp

void main()
{
int textnummer;                                  // Ganzzahl-Variable
char frage = "Welchen Text ausgeben?\n";         // char-Datentyp (Strings)
cout << frage;                                   // Frage-String wird ausgegeben
cin >> textnummer;
Text_ausgeben(textnummer);                       // Funktionsaufruf mit Übergabe von textnummer
}

void Text_ausgeben(int textnummer)               // Variable der main-Funktion hier unbekannt,
{                                                // da lokale Variablen nur in der Funktion selbst
                                                 // gelten. Deshalb muss hier neu deklariert werden.
if (textnummer == 1)
        {
        cout << "Dies ist Text1";
        }
if (textnummer == 2)
       {
       cout << "Dies ist Text2";
       }
 }                                               // Ende der Funktion, also zurück nach der Funktion die
                                                 // Text_ausgeben aufgerufen hat, hier also main
```

Hier muss in "Text_ausgeben" in den Parametern auch der Typ der Variable stehen, da er dort neu angelegt wird. Die Funktion kennt die Variable in “main” nicht (siehe Definiton von lokaler Variable).

Im Aufruf der Funktion (in main letzte Zeile) darf kein Typ davor, da dieser nur einmal vor einer Variablen stehen darf (man kann sie nur einmal deklarieren, dass heisst ihren Typ festlegen und dafür Speicher reservieren).

Der Text wird ausgegeben mit einer Abfrage. Je nach Wert in der Variablen wird entweder Text1 oder Text2 ausgegeben. Danach kehrt die Funktion wieder in “main” zurück. Eine Funktion kehrt immer dahin zurück von wo sie aufgerufen wurde. Da dort nichts mehr steht, ist das Programm beendet.

2j. Rückgabewerte

Funktionen können etwas übernehmen, was dann oben in den Parametern steht. Aber sie können auch etwas zurückgeben, z.B. ob eine Funktion erfolgreich war oder einen ausgerechneten Wert.

Eine Funktion kann zwar beliebig viele Parameter übernehmen, aber kann nur einen Wert zurückgeben.

Dazu ein kleines Beispiel in der eine Funktion zu einer Eingabe auffordert, diese dann mit einer bereits vorhandenen Zahl vergleicht und ein bool´schen Wert zurückgibt. Das heisst sie gibt entweder true für wahr oder false für unwahr zurück.

```
#include <iostream.h>

bool FindeZahl();     // Prototyp der unteren Funktion, Rückgabetyp diesmal nicht void sondern bool

void main()
{
...
...
if (FindeZahl() == false))                    // hier wird der Rückgabewert von der Funktion mit true
verglichen
        {
        cout << "Funktion war nicht erfolgreich";
        }
...
...
}
```

Statt hier einen Text auszugeben, je nachdem ob Funktion wahr oder unwahr, kann man auch andere Funktionen aufrufen, die dann z.b. bei wahr weitere Abfragen machen und bei unwahr das Programm beenden. Einfachhalber wird hier nur ein Text ausgegeben.

```
bool FindeZahl()                    // Funktion, wo die eingegeb. Zahl mit der Zielzahl verglichen wird
{
const int ziel = 1234;              // Zielzahl
int eingabe;
cout << "Bitte Zahl eingeben: ";
cin >> eingabe;
if (eingabe != ziel)                    // WENN (eingabe NICHT ziel)
        {
        return false;                   // Fehlgeschlagen, es wird false zurückgeliefert
        }
return true;                  // hier landet das Programm nur, wenn es oben nicht schon
                              // bei return false zurückgekehrt ist
}
```

Hinweis zu "return":
Schreibt man “return”, dann ist die Funktion damit zu Ende, dass heisst auch wenn nach “return” noch weitere Anweisungen stehen, wird dies nie ausgeführt, wie folgendes verdeutlicht

```
int irgendeineFunktion()
{
..                          // wird alles ausgeführt
return ergebnis;       // zurückgeliefert wird die Variable namens "ergebnis"
...
cout << "Dieser Text wird nie ausgegeben";
}
```

"const":

“const” heisst, dass man diese Zahl nicht ändern kann. Dies ist zwar eh nicht möglich, da man garnicht dazu kommt (keine Eingabemöglichkeit dafür, "ziel" wird auch vom Programm selbst nie geändert), aber “const” dient auch dafür, dass man selbst sieht, dass diese Zahl immer gleich ist. Bei ein paar Hundert Zeilen erleichtert dies die Fehlersuche, wenn man von vornherein schon vor den Variablen “const” schreibt, die immer gleich bleiben. Das können auch Texte sein.

Ein weiteres Beispiel soll nun eine Rechenfunktion zeigen und diemsla einen int-Typ verwenden für die Rückgabe, also eine Ganzzahl. Hier werden der Rechen-Funktion 2 Zahlen übergeben und die Rechen-Funktion liefert als Rückgabetyp das Ergebnis zurück.

```
#include <iostream.h>                    // Einbinden für Ausgabe und Eingabe

int Rechnen(int, int);                   // Prototyp der unteren Funktion

void main()
{
int zahl1, zahl2;                        // keine Initialisierungen, da man Zahlen noch nicht weiß
cout << "Bitte 2 Zahlen eingeben: \n";
cin >> zahl1;
cin >> zahl2;
cout << Rechnen(zahl1, zahl2);
}

int Rechnen(int zahl1, int zahl2)      // Rückgabetyp Name der Funktion und (Übergabe-Parameter)
{
int ergebnis = zahl1 * zahl2;
return ergebnis;                         // Rückgabewert abhängig vom Typ oben, also eine Ganzzahl
}
```

Ausgabe ist dann diese:
(Wer einen Zeilenumbruch möchte, setzt einfach einen dazwischen)

```
Bitte 2 Zahlen eingeben:
2 3 6
```

2k. Vorbelegung von Variablen in Funktionsparametern

Irgendwo ganz weit oben steht was von Funktionsprototypen. Diese haben noch einen weiteren Zweck. Hier erstmal ein paar normale Funktionsprototypen von Funktionen, die was übernehmen und dies dann ausgeben.

```
void GibTextAus(char text);
void GibNummerAus(int nummer);
```

Nun zur möglichen Vorbelegung. Vorbelegen heisst, man belegt die Variablen schon vor, und wenn man nichts üergibt, nimmt der Compiler diese Werte. Ansonsten werden sie überschrieben. Wenn man die Prototypen (und nur die Prototypen) etwas ändert, dann hat man diese Variablen vorbelegt. In der Funktion selbst bleibt alles beim alten, drum habe ich diese hier auch weggelassen.

```
void GibTextAus(char text = “Kein Text übergeben”);
void GibNummerAus(int nummer = 0);
```

Und wie nutzt man dies?
Werden die Funktionen jetzt aufgerufen und man übergibt ihnen einen Text/Zahl, dann arbeitet die Funktion damit. Allerdings kann man jetzt auch neu überhaupt nichts übergeben (wäre sonst ein Fehler).
Dann nimmt der Compiler die Vorbelegungen, was in diesen Fall heissen würde, er würde “0” bzw “Kein Text übergeben” ausgeben.

2l. Überladen von Funktionen

Eine sehr gute Neuerung gegenüber C ist die Überladungsmöglichkeit. Was heisst das nun? Normalerweise kann nur eine Funktion einen bestimmten Namen haben. Also kann es nur eine Funktion namens “datei_schreiben” usw. geben. Überladen heisst, es können mehrere Funktionen mit den gleichen Namen aber mit unterschiedlichen Parametern auftauchen. Beim dateischreiben fällt mir dafür folgendes ein:

Prototypen:

```
datei_schreiben(char dateiname);
datei_schreiben();
```

Funktionen:

```
datei_schreiben(char dateiname)
{
...
}

datei_schreiben()
{
...
}
```

Die Nutzung des Ganzes ist denkbar einfach. Übergibt man der Funktion einen Dateinamen, wird die erste Funktion aufgerufen, die mit den Parameter. Übergibt man nichts, wird die zweite Funktion die ohne Parameter aufgerufen. Dann könnte hier eine neue Datei angelegt werden.

Dies geht auch mit anderen Parametern und es können zum Beispiel auch 5 Funktionen mit den selben Namen auftauchen, wenn verschiedene Parameter vorhanden sind, sieht der Compiler das auch andee Funktionen. Anstatt dieses folgende:

```
Zeichne_Dreieck(int, int, int);          // 3 Koordinaten
Zeichne_Fünfeck(int, int, int, int, int);  // 5 Koordinaten
```

... könnte man auch das so schreiben:

```
Zeichne(int, int, int);
Zeichne(int, int, int, int, int);
```

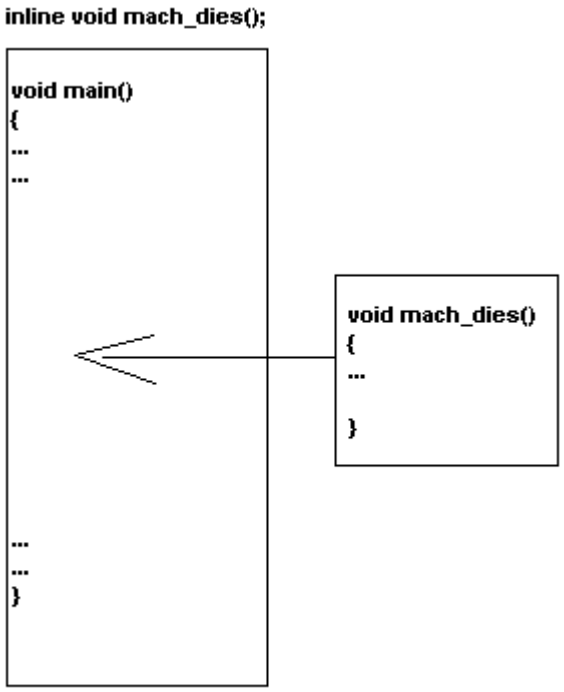
Und natürlich ist die Groß- und Kleinschreibung auch noch zu beachten. Das folgende sind für den Compiler alles verschiedene Funktionen, da keine so ist, wie die andere:

```
Zeichne(int, int, int);
ZeichNe(int, int, int);
zeichne(int, char, int);
zeiChne(int, char, int);
zeichne(float, char, int);
zeichnen(float, char, int);
```

In den Funktionsprototypen ist der Name nicht vorgeschrieben. In den Funktionen selbst oben in den Parametern muss er jedoch dabeistehen.Wer sich nicht sicher ist, kann ihn immer hinschreiben, hier also ecke1, ecke2, ecke3, usw... Sobald man was größeres macht, wird man ihn aber aus Schnelligkeitsgründen weglassen.

2m. Inline-Funktionen

Dieses kann man relativ kurz fassen. Schreibt man vor dem Prototyp der Funktion das Wort “inline” so ist dies eine inline-Funktion (nur in den Prototyp das Wort “inline” davorschreiben). Das heisst, dass der gesamte Inhalt der Funktion beim kompilieren dort reinkopiert wird, wo die Funktion normalerweise andernfalls aufgerufen würde. Dann spart man einen Aufruf. Dazu ein Schaubild.



Und Sinn des ganzen?

Bei den Rechnern heutzutage, die mehrere Millionen Anweisungen pro Sekunde nur wenig. Früher aus Schnelligkeitsgründen, da jeder Aufruf Zeit kostete bei den mickrigen Speichern. Trotzdem kostet jeder Aufruf ein paar Millisekunden Zeit und Funktionen, die sehr sehr oft gebraucht werden, sollte man als “inline” angeben oder die Programmstruktur nochmal überdenken, denn der Nachteil von inline-Funktionen ist, dass es die ausführbare Datei größer macht. Logisch wenn eine Funktion beispielsweise ein paar hundert Mal kopiert wird, anstatt sie aufzurufen.

3. Variablen

3a. Was sind Variablen?

Variablen sind Bereiche im Speicher, die einen Wert/Text speichern. Es gibt bestimmte Datentypen, damit der Compiler weiß, was für eine Art diese Variable ist und wieviel Speicher zu reservieren ist. Für einen Text braucht man eine andere Menge an Speicher als eine Ganzzahl und für eine Kommazahl wiederrum eine andere Menge. Ums anders zu sagen, für ein Buch braucht man anderen Platz als für eine CD oder Auto. Und für Ein Flugzeug braucht man nicht ganz soviel Platz wie für einen ganzen Planeten. Wie Variablen genutzt werden, ist ja oben schon zu sehen, hier noch ein paar Erklärungen dazu.

3b. Deklarationen und Zuweisungen

Variablen sind solange gültig bis das Programm beendet wird. Ändert man eine Variable während eines Programmdurchlaufs, schließt man das Programm dann und startet es neu, beginnt wieder alles von neuem. Alle Variablen werden wieder zurückgesetzt. Hat sich das Programm irgendwo mitten im Ablauf befunden, egal, alles beginnt wieder neu - Zum Glück, man stelle sich nur vor was bei einen Programmabsturz wäre, man würde Neustarten und der Fehler wäre immer noch da, drum hilft Neustarten auch bei sowas immer so gut.

```
int y = 5;          // Deklaration und Zuweisung
int z = 5;
int x;              // nur Deklaration weil man noch keinen Wert hat

x = y + z;          // Ergebnis
```

Dies ist eine Deklaration von 3 Variablen namens x, y und z. Dies heisst dass man den Compiler sagt, dass dort Speicher hierfür zu reservieren ist. Diesen Vorgang braucht und darf man nur einmal für jede Variable machen. Für Ganze Zahlen wird “int” verwendet.

Man schreibt also beim ersten Benutzen einer Variablen oder schon vorher:

```
Variablentyp Variablenname;
```

oder

```
Variablentyp Variablenname = Wert;          // wenn man den Wert am Anfang schon weiß
```

In der letzten Zeile oben wird die Summe von y und z an x zugewiesen, so dass X gleich Y plus Z ist. Man könnte x auch gleichzeitig deklarieren und einen Wert zuweisen.

```
int y = 5;
int z = 5;
int x = y + z;
```

Diese Zeile heisst auch, dass x = y + z ist (nebenbei: dies nennt man Zuweisung). Ohne Deklaration kann man eine Variable nicht benutzen, da der Compiler nicht weiß, was dies dann für ein Datentyp ist.

Das geht also nicht, da man “x” nicht 2 Mal anlegen kann. Das wäre so, als ob man 2 Fächer mit gleichen Namen für eine Sache anlegt.

```
int x;
int x = y + z;
```

Und folgendes geht auch nicht, da der Datentyp verschieden ist:

```
char text = “test”;          // Text-Variable
int x = 5;                   // Ganzzahl-Variable
int y = x +  text;           // "text" plus "zahl" kann man nicht rechnen
```

Ausnahme wäre eine Typenumwandlung, die hier aber erstmal nicht erklärt wird.

Das nächste sind alles mögliche Zuweisungen (man weist also der Variable einen Wert zu):

```
x = 5;                  // x ist jetzt 5, irgendwann vorher muss “x” schon deklariert worden sein

x = 10;                 // x ist jetzt 10

int d, z = 5, u;        // So deklariert man mehrere Variablen, nur “z” wird hier vorbelegt

d = 2;                  // und hier wird “d” mit dem Wert 2 belegt

d + z;                  // das geht aber ergibt keinen Sinn, geht aber, doch die beiden Werte werden
                        // zusammenaddiert aber nirgends gespeichert

u = d + z;              // das klingt schon besser

u = d + d;              // das hier ergibt 2 mal den Wert von “d”
u = 2 * d;              // das hier aber auch
```

Mehr zum Rechnen mit Variablen später.

3c. Präfix und Postfix (Übersetzt = davor und danach)

Man nehme eine Anzahl von Äpfeln.

```
int aepfel = 20;
```

Nun zählt man einen hinzu oder zieht einen ab. Wie drückt man dies Codetechnisch aus?

Möglichkeiten gibt es da verschiedene:

```
aepfel = aepfel + 1;        // sich selbst plus 1 rechnen
aepfel = 1 + aepfel;        // das gleiche
aepfel = apefel++;          // und mal was neues
aepfel = ++aepfel;          // auch um eins erhöht
aepfel = aepfel--;          // hier wird eins abgezogen
```

Warum heisst C++ so wie es heisst? - Weil es eins über C ist. Man kann eine Ganzzahlvariable einfach um eins erhöhen (inkrementieren) oder um eins verringern (dekrementieren) indem man “++” oder “--” davor oder auch danach hinschreibt. Dies ist eine gute Methode, da man sehr oft eine Variable nur um 1 erhöht oder verringert. Beispiele wären da Zähler, Energiepunkte, Stufe, Bytes, Passwort-Eingabe-Versuche, Levelnummer, Sekunden oder Minuten bei einer Uhr und viele viele andere.

Was ist nun Präfix und Postfix?
Ganz einfach, es gibt einen kleinen Unterschied ob man

```
aepfel++;
```

oder

```
++aepfel;
```

schreibt.

Befinden sich eine Erhöhung/Verringerung einer Variablen und dazu noch Beispielsweise "cout" in der selben Zeile, nur dann ist dies wichtig:

```
tomaten++;      // Wert in Variable "tomaten" wird erhöht, nachdem mit Variablen gearbeitet wurde
++tomaten;      // Wert in Variable "tomaten" wird erhöht, dann wird mit der Variablen gearbeitet

tomaten--;      // Wert in Variable "tomaten" wird verringert nachdem mit Variablen gearbeitet wurde
--tomaten;      // Wert in Variable "tomaten" wird verringert, dann wird mit der Variablen gearbeitet
```

```
tomaten = 5;
cout << tomaten++;          // Ausgabe: 5, erst Ausgabe, dann Erhöhung um 1

tomaten = 5;
cout << ++tomaten;          // Ausgabe: 6, erst Erhöhung um 1, dann Ausgabe
```

Das gleiche gilt für die Verringerung um 1. Dies ist wie gesagt nur wichtig wenn in einer Zeile mehrere Anweisungen stehn, da snst ja Zeile für Zeile abgearbeitet wird.

Bei den folgenden Zeilen ist es egal ob man das als Präfix oder Postfix schreibt, da hier eine klare Reihenfolge gegeben ist.

```
tomaten = 5;
tomaten++;              // Präfix
cout << tomaten;        // tomaten = 6
```

oder

```
tomaten = 5;
++tomaten;              // Postfix
cout << tomaten;        // tomaten = 6
```

---> Ist das selbe.

Für ein weiteres Beispiel, wie man um eins runter oder rauf zählt erstmal etwas Code:

```
void main()
{                                   // Anfang
 int gesamt;                        // Deklaration von einer Variable namens gesamt
 int aepfel = 4;                    // Man hat 4 Äpfel
 int gegessen;
 gegessen = 1;                      // Man isst einen und schreibt den Wert in die dafür
                                    // vorgesehene Variable
 gesamt = aepfel - gegessen;        // Ausrechnng: Jetzt sinds nur noch 3
 }                                  // Ende
```

Dieses Beispiel wird nun etwas umgeschrieben. Das Ergebnis ändert sich jedoch nicht.

```
void main()                 // Funktion, mehr dazu im Funktionskapitel
{                           // Anfang
int aepfel = 4;             // Man hat 4 Äpfel
aepfel -= 1;                // Man isst einen
}                           // Ende
```

Dies war nun eine weitere Möglichkeit etwas runterzuzählen. Wenn man "-=" schreibt heisst dies, man zieht die rechte Zahl ab und weist das Ergebnis der linken Zahl zu. Umgekehrt gilt dies auch für "+=".

Statt "-=" kann man auch inkrementieren (1 dazuzählen) und dekrementieren (1 abziehen) wie oben schon erklärt. Der Code sehe dann folgendermaßen aus:

```
void main()
{                           // Anfang
int gesamt;                 // Deklaration von einer Variable namens gesamt
int aepfel = 4;             // Man hat 4 Äpfel
aepfel--;                   // Man isst einen, Variable aepfel ist jetzt 3
aepfel++;                   // Man bekommt einen, Variable aepfel ist jetzt wieder 4
gesamt = aepfel;            // Einfache Zuweisung von den Wert in aepfel an gesamt
}                           // Ende
```

Hinweis:
Die Programme sind bisher zwar ausführbar, man sieht aber nichts, da dort nur gerechnet wird, und nichts ausgegeben. Versuche dies zu ändern mit Hilfe von "cin" und "cout", falls du einen Compiler hast.

3d. Leerzeichen und Tab´s

Leerzeichen und Tab´s nennt man “Whitespaces”. Diese werden nicht beachtet vom Compiler. Sie dienen nur zur Trennung von Wörtern und zur Übersicht. Statt des obigen Codes könnte man also genauso gut hinschreiben:

```
        x=    y           + z          ;
oder
        x=y+z;

oder   x
        =y          +z       ;
```

Der Nachteil dieses grad gezeigten ist die schwere Lesbarkeit. Wenn man codet sollte man diese und weitere Regeln einhalten, die wichtigsten:

- Jede Anweisung in eine extra Zeile
- Zwischen den Funktionen, Strukturen und Klassen etwas Platz lassen
- gut Kommentieren
- auf gleiche Rechtschreibung achten, zum Beispiel alles klein ausser konstante Variablen
- sich Überlegen wie man Wörter trennt in Variablennamen und Funktionsnamen, 3 Möglichkeiten:

```
        Anzahl_Aepfel          // richtig --> Der Unterstrich ist als einzigstes Sonderzeichen erlaubt
        AnzahlAepfel           // richtig
        Anzahl-Apefel          // falsch --> Bindestrich ist nicht erlaubt
```

- den Inahlt von Funktionen etwas mehr einrücken:

```
        void eine_funktion()
        {
                Code
        }
```

Kurz gesagt ein einheitlicher Stil ist sehr wichtig, wenn man nicht dauernd suchen möchte, wie hat man nun das geschrieben und wie eine andere Variable/Funktion.

3e. Namen von Variablen

Variablen die x, y oder z heissen nützen eigentlich niemanden. Der Name sollte so sein, dass man an jeder Stelle des Quellcodes weiß, wofür diese Variable ist. Stell dir ein Programm mit 20 Funktionen und 30 Variablen mit Namen wie “g3gs4576ghdhdgroue58346” vor. Wäre superlustig, dass zu bearbeiten.

Schlechte Namen:

```
        int x;
        char text1;
        float wichtig;      // mitten im Programm verliert man bei mehren
                            // solchen Variablennamen den Überblick
        int egtrfsg;
        char blabla;
        int ein_int_kommt_selten_allein;
```

Gute Namen:

```
        int gesamt_aepfel;
        float gehalt;
        int auto_gang;
        int spieler_energie;
        char endtext;
        char fehler_beschreibung;
```

3f. Mehrfachzuweisungen

Wie der Name der Überschrift schon sagt, man weist mehrere Variablen gleichzeitig zu.

```
int neu = gesamt = aepfel;
```

Hier wird zuerst "aepfel" an "gesamt" zugewiesen, dann der Wert in "gesamt" an Variable "neu". Dies kann man beliebig in die Länge ziehen wenn man 2 oder mehrere Variablen mit den gleichen Wert nämlich der ganz hinten belegen möchte.

Hier folgend erstmal ein paar Variablen deklariert.

```
int a, b, c, d, e;
```

Als Namen werden einfach mal ein paar Buchstaben genommen. Nebenbei: Zahlen als Variablennamen oder am Anfang von Variablennamen mit Buchstaben sind nicht möglich. Bei Funktionsnamen gilt das gleiche. Und Zahlen sollte man in Namen wenig verwenden. Das nervt gewaltig bei großen Programmen, wenn hinten jeder Name eine Zahl zu stehen hat. Zahlen in der Mitte oder am Ende sind aber möglich.

Nun werden hier den einzelnen Variablen Werte zugewiesen:

```
a = 5;
b = 5;
c = 5;
d = 5;
e = 5;
```

Und wie geht nun eine Mehrfachanweisung? Dies würde dann so ausschauen:

```
a = b = c = d = e = 5;
```

Oder so:

```
c = b = e = d = a = 5;
```

Es kommt am Ende also besser gesagt am Anfang das gleiche raus.

3g. Konstante Variablen (Konstanten)

Wenn man vor einer Variablendeklaration das Wort “const” hinschreibt, so kann man diese Variable nicht mehr ändern. Dies nennt man dann Konstante.

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

Möchte man das wie im folgenden trotzdem ändern, gibt das einen Fehler:

```
pi = 4.32;        // nicht möglich
```

Weclhen Zweck deint das?
Man sieht besser durch und es erleichtert die Fehlersuche.

3h. Neue Typdefinition

Und wieder ein neues Wort. Man kann mit dem Wort “typedef” Typen neudefinieren für Variablen oder den Rückgabewerten von Funktionen.

Muster:

```
typedef signed short int kurzzahl;        // typedef ALT NEU Simikolon
```

Nach dieser Typenneudefinition kann man nun statt “signed short int” einfach “kurzzahl” schreiben.

```
kurzzahl produkt;       // Wird vom Compiler dann als signed short int gelesen
```

Die hier genannten Typen werden noch erklärt. Man könnte auch nur aus int einen neuen Typ machen und aus float und auch aus char, hier werden jetzt mal aus allen 3 zusammen ein neuer Typ gemacht:

```
typedef     int     ganzzahl;
typedef     float   kommazahl;
typedef     char    text;
```

Nun kann man zusätzlich statt

```
int tomaten;
```

auch

```
ganzzahl tomaten;
```

schreiben. Dies ist zwar nicht sehr empfehlenswert, funktioniert aber und zeigt was eine Typen-Neudefinition ist. Grundsätzlich sollte man dies nur einsetzen wenn der Typ wirklich sehr lang ist und man diesen Typ aber häufig benutzt, was höchstens bei der Windows- oder DirectX-Programmierung der Fall ist. Alterntiv geht auch “#define”, was später erklärt wird.

3i. Casting

Casten heisst Umwandeln einer Variablen eines Typen in einen anderen Typen. Also “int” in “float”, oder “float” in “int” oder “double” in “int” usw. Aber nur wenn dies auch möglich ist, eine Kommazahl ist ja in eine Ganzzahl umwandelbar aber ein Text nicht (geht trotzdem, kommt aber nichts Sinnvolles raus).

```
int aepfel = 200;       // Ganzzahl, Variable namens "aepfel" Wert = 200
float birnen = 25.5;    // Kommazahl, Variable namens "birnen" Wert = 25.5
```

Neu und Wichtig:
Kommas werden mit Punkten angegeben. Das rührt daher, dass im Englischen ein Komma ein Punkt ist und ein Punkt ein Komma.

```
int obst;                       // Ganzzahl, Name: obst, Wert: noch keiner
obst = aepfel + birnen;         // FEHLER! Verschiedene Typen
```

Der richtige Weg ist nun das Casten eines der beiden Typen.

```
obst = aepfel + (int)birnen;      // Variable "birnen" wird nun in eine Ganzzahl umgewandelt
cout << obst;                     // Ausgabe: 225, hinter den Komma das fällt weg
```

oder

```
obst = (float)aepfel + birnen;      // Variable "aepfel" wird nun in eine Kommazahl umgewandelt
cout << obst;                       // Ausgabe: 225.5
```

Aufbau beim Casten ist also folgender:

```
(neuer Typ)Variablenname      // Variable wird in den Klammern angegebenen Typ umgewandelt
oder
neuer Typ(Variablenname)      // Variable wird in den davor angegebenen Typ umgewandelt
```

Man kann also entweder den Variablennamen oder den Typ in Klammern setzen. Für die, die bisher nur in "php" programmiert haben, sicherlich etwas unverständlich, warum der Compiler nicht selbst castet, aber "php" ist halt nicht C++, wie du bei Variablen schon bemerkt haben solltest.

3j. Zusammen deklarieren

Variablen können auch zusammen deklaraiert werden. Statt

```
int x;
int y;
```

kann man auch

```
int x, y, z;
```

schreiben. Also nur mit Komma getrennt und vorne der Typ. Der Typ muss gleich sein.
Man kann den Variablen auch gleich einen Wert zuweisen (oder nur teilweise wie folgt)

```
int x = 3, y = 4, z;
```

Diese können dann auch jederzeit wieder geändert werden.
Irgendwo im Programm:

```
x = 4;
y = 200000;
z = 0;
```

3k. Rechnen mit Variablen

Gerechnet wird mit den 4 Grundrechenarten (auf Nummerblock), auch hier gilt "Punktrechnung geht vor Strichrechung" oder man setzt Klammern, dazu ein paar Beispiele

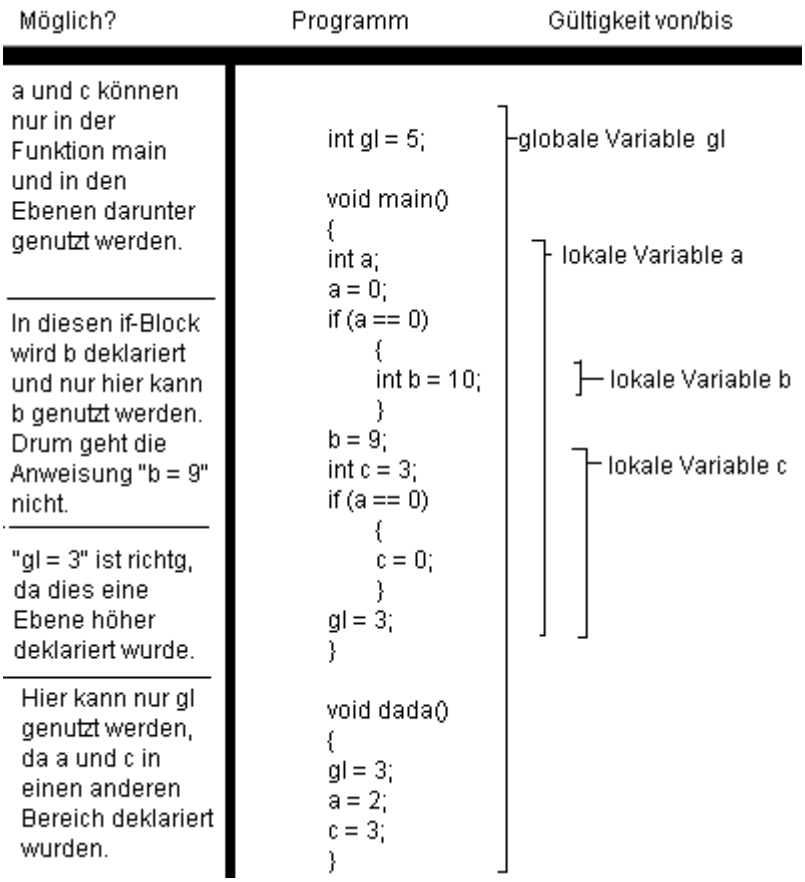
```
gesamt = x + y;                 // Addition
gesamt = x - y;                 // Subtraktion
gesamt = x * y;                 // Multiplikation
gesamt = x / y;                 // Division

gesamt = (x + y) * 5;           // x plus y rechnen, dann mal 5
gesamt = 1000++;                // ergibt 1001
gesamt = 1000--;                // ergibt 999
gesamt += 1000;                 // auch 1001
gesamt = gesamt + 1000 - x;

if (x < 100) { x++; }           // wenn x unter 100 dann 1 dazu
else { x = 20; }                // sonst x gleich 20
```

Es ist also wie in der Mathematik.

3l. Lokale und Globale Variablen

Die Variablen werden in Blöcken (z.B. Ein Funktionsblock) deklariert und sind deshalb auch nur da gültig. Ein Block ist immer das zwischen “{” und “}”. Prinzipiell kann man Variablen überall deklarieren. Hauptsache wenn sie genutzt wird, wurde sie schon deklariert und die Variable wurde im gleichen Block oder im Block darüber deklariert. Ein Block ist also der Inhalt von Funktionen oder der Inhalt von if-Anweisungen und weiteren Schleifen.

Das Schaubild soll nun darstellen wo Variablen gültig sind. Die Variable “gl” ist die einzig globale hier. Globale Variablen werden in der höchsten Ebene also außerhalb jeder Funktion deklariert und sind deshalb überall im gesamten Programm gültig. Die 3. Spalte “Gültigkeit von/bis” gibt den Bereich an, von und bis wann die Variable gültig ist, nämlich von der Deklaration bis zum Ende des jeweiligen Blockes.



Zur Übung gibt es jetzt ein Programm mit 5 Fehlern:

```
#include <iostream.h>;

void Produkt_rechnen                    / Prototyp

void main()
{
int produkt;
int zahl1 = 2;
int zahl2 = 3,
produkt_rechnen();
}

void Produkt_rechnen()
{
Produkt = zahl1 * zahl2;
cout << produkt;
}
```

Fehler dieser Funktion oben sind nun:

1. Hier wurde mal Groß mal Klein geschrieben, der Aufruf wird so nicht gehen, und womit das Programm rechnet, gibts teilweise garnicht. Wenn man einmal Produkt schreibt und einmal produkt dann ist das für den Compiler wie Äpfel und Birnen.

2. Das Wort Prototyp wurde nicht auskommentiert und wird als Code angesehen werden, dies gibt einen Fehler. Auskommentiert wird mit // und nicht mit /.

3. Bei "include <datei.h>;" Hier ist das Simikolon falsch. Includiert wird ohne Simikolon.

4. Der Prototyp der Funktion "Produkt_rechnen" hat 1. kein Simikolon dran und 2. fehlen die runden Klammern.

5. Selbst wenn man dies jetzt alles verbessert hat, kann nichts ausgerechnet werden. Die Funktion "Produkt_rechnen" kennt "zahl1" und "zahl2" nicht. Es wurde auch nicht deklariert, da in der main alle Variablen lokal sind, das heisst nur die main kennt die Deklarationen und die Werte.

Sinn dieses Fehlerprogramms?

Nun ich wollte zeigen, was beliebte Fehler sind. Wer seine ersten Programme bastelt, der sollte sich sowas wie eine mögliche Fehlerliste machen, nach der man gehen kann.
Und dann jeden einzelnen durchgehen. Die Fehlersuche nimmt bei großen Spielen und Programmen unglaublich viel Zeit in Anspruch und es gibt dann (verständlicherweise) immer neuere bei sehr großen Programmen (ich sag nur W.....s). Und man sollte nichts 2 Mal kontrollieren.

Hier eine etwas überarbeitete Version von der Fehlerfunktion, diesmal alles richtig:

```
#include <iostream.h>

void produkt_rechnen(int zahl 1, int zahl2)

void main()
{
int zahl1;
int zahl2 = 0,
cout << "Bitte 2 Zahlen eingeben: "
cin >> zahl1;
cin >> zahl2;
produkt_rechnen(zahl1, zahl2);          // Übergibt zahl1 und zahl2 an untere Funktion
}

void produkt_rechnen(int zahl1, int zahl2)
{
int produkt = zahl1 * zahl2;                                  // rechnen
cout << "Das Ergebnis ist " << produkt << "...";       // Ausgabe
}
```

Die Ausgabe sieht dann so aus:

```
Bitte 2 Zahlen eingeben: 5 3
Das Ergebnis ist 15 ...
```

Zwischen “cin” und “cin” könnte man jetzt auch noch einen Zeilenumbruch ausgeben. Leerzeichen/ Sonderzeichen in Variablennamen sind nicht möglich. Nur der Unterstrich _ ist als einzigstes Sonderzeichen möglich. Man kann mit “cout” auch mehrere Variablen und Texte gleichzeitig ausgeben. Hier im Beispiel wird einmal ein String und danach eine Variable ausgegeben. Bei Variablennamen kommen keine Anführungsstriche dran. Bei Strings kommen Anführungsstriche dorthin, da der Compiler das erkennen muss, dass er damit nicht irgendwas tun soll. Hier wurde ausserdem alles kleingeschrieben und man kann die Zahlen diesmal selbst eingeben.

Dies war mit lokalen Variablen geschrieben. Da die obige lokale Variable in der “Text_ausgeben”-Funktion unbekannt ist und nur der Wert übergeben wird, kann diese lokale Variable auch anders heissen wie gleich im Beispiel. Es wird nur der Wert übergeben. Dieser wird dann der neudeklarierten Variable übergeben und dort gespeichert.

```
#include <iostream.h>

void Text_ausgeben(int textnr);                    // Prototyp

void main()
{
int textnummer;                                      // Ganzzahl-Variable
char frage = "Welchen Text ausgeben?\n";     // char-Datentyp (Strings)
cout << frage;                                       // Frage-String wird ausgegeben
cin >> textnummer;
Text_ausgeben(textnummer);                    // Funktionsaufruf mit Übergabe nur vom Wert

void Text_ausgeben(int textnr)
{
if (textnr= 1)  { cout << "Dies ist Text1"; }
if (textnr= 2) { cout << "Dies ist Text2"; }
}
```

Dies waren jetzt Beispiele mit lokalen Variablen. Würde man das jetzt umschreiben mit globalen Variablen, würde dies so aussehen:

Nochmal zur Erinnerung:
Global heisst, dass die Variablen irgendwo ausserhalb von Funktionen stehen und darauf dann jede Funktion zugreifen kann. Da diese Methode jedoch veraltet ist und man auch leichter durcheinander kommt, wenn da jede Variable irgendwo in der Luft steht, macht man dies nur wenn dies wirklich nicht anders geht.

```
#include <iostream.h>

void Text_ausgeben();              // Prototyp

int text;                          // Ganzzahl-Variable diesmal global, also nicht in einer Funktion
                                   // auf diese Weise können alle Funktionen textnummer nutzen

void main()
{
char frage = "Welchen Text ausgeben?\n";       // char-Datentyp (Strings)
cout << frage;                                 // Frage-String wird ausgegeben
cin >> text;
Text_ausgeben();                               // Funktionsaufruf
}

void Text_ausgeben()                 // keine Parameter, trotzdem muss immer ( und ) dabeistehen
{
if (text == 1)  { cout << "Dies ist Text1"; }
if (text == 2) { cout << "Dies ist Text2"; }
}
```

Auf die globale Variable "text" können diesmal alle Funktionen zugreifen.

Noch eine andere Möglichkeit wäre diese, in der alles in main steht, wäre dies:

```
#include <iostream.h>

void main()
{
int text;                                      // lokale Ganzzahl-Variable
char frage = "Welchen Text ausgeben?\n";       // char-Datentyp (Strings)
cout << frage;                                 // Frage-String wird ausgegeben
cin >> text;
if (text == 1)  { cout << "Dies ist Text1"; }
if (text == 2) { cout << "Dies ist Text2"; }
}
```

Hier sind die Anweisungen fürs Textausgeben alle in "main" und es gibt keine Extrafunktion.

Das kann man zwar machen, wenn man 4 oder 5 Zeilen hat, aber wenn man z.B. 50 Zeilen hat, gehts nicht mehr. Man versucht dies deshalb in Funktionen einzuordnen, also Funktion x macht dies dann und Funktion y dann das und z widerrum was ganz anderes.

3m. Lokale und Globale Variablen mit gleichen Namen

Was passiert wenn eine Lokale und Globale Variable den gleichen Namen hat? Die Frage ist schnell zu beantworten: Die lokale Variable hat Vorrang und würde zuerst genutzt:

```
int variable;              // Globale Variable

void funktion();           // Prototyp

void main()
{
int variable;              // Lokale Variable mit gleichen Namen
variable = 5;
}
```

So, nun die Kontrollfrage: Welche der beiden Variablen wird hier auf 5 gesetzt? Wer oben richtig gelesen hat, wird dies beantworten können: Die lokale Variable, denn diese hat Vorrang.

Wie kommt man trotzdem an die globale Variable? Wieder eine kurze Frage und eine kurze Antwort:
Mit Doppelpunkten. Im folgendes Codeteil wird zuerst die lokale Variable auf 5 gesetzt und dann die globale Variable auf 3.

```
int variable;              // Globale Variable

void funktion();           // Prototyp

void main()
{
int variable;              // Lokale Variable mit gleichen Namen
variable = 5;              // Lokale Variable wird auf 5 gesetzt
::variable = 3;            // Globale Variable wird auf 3 gesetzt
}
```

Schlussfolgerung: Existieren also 2 Variablen, einmal lokal und einmal global mit gleichen Namen, so kann man im Bereich der lokalen Variablen, wo die lokale dann auch Vorrang hat, die globale nutzen indem man 2 Doppelpunkte vor den Namen setzt.

Die Doppelpunkte gehören nicht zum Namen, sie weisen nur den Compiler an, dass jetzt die globale Variable gemeint ist.

4. Datentypen

Bei der Deklaration von Funktionen und Variablen, also das Reservieren des Speichers, dass es diese Funktion/Variable gibt, anders gesagt, das Bauen eines Faches, indem was reingepackt werden kann, muss vor dem Funktionsnamen und Variablennamen der jeweilige Datentyp geschrieben werden um die Größe des Speichers hierfür zu bestimmen.

Bisher wurde in den obigen Beispielen meist “int” für eine Ganzzahl oder “void” für Nichts genommen, es gibt aber noch mehr Datentypen. Hier nun die reinen C++ Datentypen (es gibt noch mehr in der Windowsprogrammierung und noch mehr in DirectX).

4a. Datentypen bei Variablen

Muster:

```
Datentyp Variablenname ;        // bei Deklaration/Initialisierung
Variablenname = Wert ;          // Zuweisung eines Wertes
```

Datentypen (Größe des Speicher nach je System leicht abweichend)

int	- Ganzzahl = von -2147483648 bis +2147483647
long int	- Ganzzahl = genauso da wenn nur “int” dabeisteht “long” angenommen wird
short int	- Ganzzahl = von -32768 bis + 32767

Zur Verwendung: Man nimmt “int” standardmäßig für Ganzzahlen. Wenn “int” nicht ausreicht, nimmt man “long int”. Wenn auch “short” ausreicht nimmt man “short int” um Speicher zu sparen. Die Technik ist allerdings so weit fortgeschritten, dass es soviel Speicher gibt, dass man auch immer einfach nur “int” nehmen kann und sich “short” und “long” sparen kann.

Nun weitere Datentypen:

char	- Buchstaben und Ziffern bis 128 Zeichen
bool	- steht für boolean und heisst entweder richtig oder falsch (true/false)
float	- Kommazahlen (+/- 3.4e), 7 Ziffern
double	- Kommazahlen mit doppelter Genauigkeit, also 14 Ziffern

Vor “int” kann man dann noch “unsigned” oder “signed” schreiben. Wenn man nichts davor schreibt wird “signed” angenommen. “signed” heisst mit Minus-Vorzeichen, damit kann man obige Zahlen verwenden. “unsigned” heisst ohne Vorzeichen, es gibt also keine Negativwerte, dafür kann die Zahl größer sein.

Bytegröße:
Jeder dieser Datentypen belegt im Speicher des Computers mal 2 Byte, mal 4 Byte, mal 8 je nach Rechner.

Der Datentyp “void” hat vor Variablennamen nichts zu suchen. Man kann ja nicht einen Wert in “Nichts” stecken. Dieser ist nur für Funktionsrückgabewerte bestimmt, wenn diese nichts zurückgeben wie bei “main” meist der Fall.

4b. Datentypen bei Funktionen (heisst hier dann Rückgabetyp)

Muster:

```
Rückgabetyp Funktionsname (Übergabeparameter)
{
Anweisungen (also Zuweisungen, Funktionsaufrufe u. ä. );
return Rückgabewert;
}
```

Hier steht der Datentyp für den Rückgabewert (ein Wert welche die Funktion zurückgibt). In den ersten Beispielen wird “void” verwendet (nur bei Funktionen möglich). Dies heisst das nichts zurückgegeben wird (kein return in letzter Zeile). Zurückgeben heisst, dass eine Funktion beispielsweise das Produkt ausrechnet und dies dann zurückgibt und dafür muss Speicher reserviert werden und den Compiler mitgeteilt werden, was dort genau zurückgegeben wird. Wenn dies eine Ganzzahl ist schreibt man also

```
int produkt_ausrechnen(Übernahme zweier  Zahlen)
{
produkt = zahl1 * zahl2;
return produkt;                    // Ganzzahl, deswegen int als Typ vor Funktionsname
}
```

Nimmt man nun als Rückgabetyp “bool” kann die Funktion eine Entscheidung fällen und dann “true” oder “false” zurückliefern und bei einen “char”-Rückgabetyp, gibt die Funktion einen Text oder Buchstaben zurück.

5. Schleifen und Abfragen

5a. Vorbereitung

Im nachfolgenden Kapitel kommt “#define” vor, dies hat folgenden Syntax (Aufbau)

```
#define   DasWasAusgetauschtWerdenSoll   DasWasDaDannHinSoll   Simikolon
```

“#define” wird in der .cpp-Datei ganz oben hingesetzt, damit man schnell mal was austauschen kann. Da gibt es jede Menge auszutauschen, bestimmte Wörter gegen Werte zum Beispiel
(also etwa die Wörter “PI” gegen 3.141592 oder einen selbstdefinierten (selbstgemachten) Variablentyp gegen den richtigen Variablentyp-Namen).

Und noch ein wichtiger Satz den man für dieses Kapitel unbedingt braucht:
Schleifen und Abfragen können nur innerhalb von Funktionen stehen, nicht irgendwo im Freien.

5b. If-Abfrage

Man kann hier einen Wert überprüfen, und je nach Wert entweder was ausführen oder nicht. Muster:

```
if (wenn das so ist, wie hier steht)
{
dann das ausführen, wenn nicht dann halt nicht und hier runter weiter, also dies hier ignorieren;
}
```

Alternativ geht auch

```
if (wenn das so ist wie hier steht)
dann das ausführen, wenn nicht dann weiter;
```

Hier wurden die Klammern weggelassen. Dies geht aber nur bei einer Anweisung unter “if”. Danach geht das normale Programm weiter. Ein besserer Stil ist wie oben mit Klammern, da man sonst leicht denkt, dass die gemeinten Zeilen zum normalen Programm gehört und auf jeden Fall ausgeführt wird.

5c. If-Else-Abfrage

Man kann hier einen Wert überprüfen, und je nach Wert entweder was ausführen oder nicht.
Zusätzlich gibt es hier ein else, das heisst wenn das nicht so ist wie in if steht.
Else heisst übersetzt “sonst”.

```
if (wenn das so ist wie hier steht)
{
dann das ausführen;
}
else
{
wird nur ausgeführt wenn if nicht ausgeführt wurde;
}
```

Und nochmal: “if”, “else”, “while”, “for” usw. können nur innerhalb von Funktionen stehen.

```
if ((schönes Wetter== ja) && (genug Zeit== ja))  { Baden gehen; }
else (zuhause bleiben;)
```

Anmerkung: “&&” heisst hier UND.

Weitere solcher Operatoren wie “&&” (und) sind:

&&	- UND	- beides muss wahr sein
II	- ODER	- eins davon muss so sein

Zum ODER-Zeichen noch etwas:
Dies ist auf deutscher Tastatur die Taste zwischen Y und ShiftLinks 1 unter A das 3. Zeichen.
Diese muss 2 Mal gedrückt werden, für 2 Striche.

!=	- NICHT	- wenn das nicht so ist
==	- GLEICH	- wenn das so ist, wird nur in if oder else verwendet,

Das Doppelte Gleichheitszeichen (==), dass nur bei Schleifen und Abfragen verwendet wird) ist nicht mit den normalen Zuweisung-Gleichheitszeichen (=) zu vergleichen. Ein sehr häufiger Fehler ist dieser hier:

```
if (wetter = regen)
{
schirm_rausholen();
}
```

Und manche Compiler kompilieren dies auch noch ohne die geringste Fehlermeldung. Das ganze wirkt sich auf dem Programmablauf dann in den Maße aus, dass es je nach Compiler entweder richtig oder falsch gelesen wird, je nach Laune des Compilers. Compiler fangen also nicht jeden Fehler ab und wenn sich ein Programm seltsam verhält, dann ist irgendein Fehler im Code, der nicht abgefangen wurde und in irgendeiner Weise zu berichtigt versucht wurde. Viele Programme mit Fehlern laufen gut bis man eines Tages ihn dann doch entdeckt.

Auch mehrere if´s sind möglich:

```
if (heute == montag)   {das machen;}
if (heute == dienstag)  {dies machen;}
else                    {das da machen;}      // weder Montag noch Dienstag
```

Wenn man dieses hier nun etwas umschreibt:

```
if (heute == montag)   {das machen;}
else if (heute == dienstag)  {dies machen;}
else                    {das da machen;}      // weder Montag noch Dienstag
```

kommt das gleiche dabei raus. Dies ist aber die bessere Variante, da man die Hierarchien (Stufen/Ebenen) der if´s und else´s leicht durcheinander bringen kann, wenn diese mehrfach verschachtelt sind.

5d. while

Auch “while” ist eine Schleife. Sie sagt in etwas aus:

```
while (solange das WAHR ist, was hier oben drin steht)
{
das hier ausführen;
}
dann weiter;
```

Im Gegensatz zum “if” führt “while” also so lange den Inhalt aus der in den Klammern steht, bis obige Bedingung nicht mehr wahr ist. “if” hat dies nur einmal kontrolliert. Wenn die Bedingung bei “if” wahr war, wurde der Inhalt ausgeführt und es ging gleich weiter.

Bei der Passwort-Abfrage-Funktion etwas weiter oben, kann man so mehrmals testen ob die Eingabe richtig ist und auch mehrmals dann abfragen.

Die 1 in der Bedingung von “while” heisst, dass dies eine Endlosschleife ist, sie läuft ewig bis in der Schleife selbst gestoppt wird. Stoppen kann man eine Endlosschleife und auch andere Schleifen durch “break”. Dann wird auch der Rest nicht mehr ausgeführt und das Programm läuft am Ende der Schleife weiter. Bei “continue” stoppt das Programm genauso, diesmal kehrt das Programm jedoch wieder zum Schleifenanfang zurück.

Generell sollte man mit Endlosschleifen vorsichtig umgehen, sonst hängt das Programm fest, wie folgendes:

```
while(1)          // Endlos-Schleife
{
int i;             // Variable Ganzzahltyp namens “i”
i++;               // Rechne eins dazu
}
```

Falls man dieses kompiliert und ausführt, kann man erstmal Neustarten, da der PC immer nur eine Anweisung ausführen kann und wenn er dies erstmal ausführt, dann führt er es endlos aus und nichts anderes geht mehr.

Folgendes richtiges Beispiel nun soll “break” und “continue” verdeutlichen:
(irgendwo in einer Funktion)

```
while(1)
{
weg suchen;
if  (verlaufen == ja) {  continue; }          // wieder am Anfang beginnen
if (ziel gefunden == ja ) {  break; }        // Ausführung geht dann weiter nach der Schleife
}
```

Und hier die vorerst letzten neuen Operatoren:

<=	kleiner oder gleich
>=	größer oder gleich
>	größer
<	kleiner

Nun folgend eine Schleife, die sich solange ausführt wie x unter, also < 5 ist.

```
int x = 0;                                  //  Variable

while(x < 5)                                // läuft solange bis x nicht mehr unter 5 ist und
{                                           // beginnt immer wieder von vorne
x++;                                        // x um eins erhöhen
}
cout << "x hat nun den Wert: " + x;         // Ausgabe --->    x hat nun den Wert: 5
```

Und jetzt noch eine verbesserte Passwort-Abfrage:

```
void main()
{
int passwort = 12345;
int zaehler = 0;                            // Variable die speichert, wie oft man eingegeben hat
cout << "Passwort eingeben: ";              // Ausgabe
while ((eingabe != passwort) && (zaehler < 3))          // 2 Kontrollen
          {
           zaehler++;                                   // zaehler erhöhen
           cin >> eingabe;                              // Eingabe abfragen
           }
// hier gehts weiter wenn die Bedingung in while nicht mehr wahr ist
if  (eingabe != passwort)
          {
           cout << "Passwort 3 mal falsch eingegeben, Programm wird beendet.";
           }
else       // hier könnte auch "if (eingabe == passwort)" stehen
          {
          cout << "Passwort richtig eingegeben, jetzt gehts weiter.";
           Passwort_geschützte_Funktion();
          }
}
```

Lese nun das Programm so wie der PC es lesen würde, Zeile für Zeile oder compiliere es und teste selbst. Hier wurde "if", "else" und "while" eingebaut. Sicherlich kann man dies auch anders aufbauen. Hauptsache es läuft und es ist leicht zu verstehen.

Was passiert hier im Programm?
Zuerst wird ein Eingabe-Zähler und ein Passwort deklariert. Nun beginnt die while-Schleife, jedoch maximal 3 mal zu fragen. Wenn das Passwort richtig war, dann ist die while-Schleife zu Ende, da die Abfrage oben ja nicht mehr wahr ist. Nach der while-Schleife gehts weiter mit einer if-else-Abfrage, die dann entweder ausgibt, dass das Passwort falsch oder richtig war und nun wird die geschütze Funktion dafür aufgerufen.

5e. do...while

Ähnlich wie die Schleife while. "do...while" diese läuft jedoch mindestens einmal durch, auch wenn die Bedingung nicht wahr ist:

```
do                      // Schleife erstmal einmal durchlaufen
{
Anweisungen;
Anweisungen;
}
while (x < 100)         // wenn wahr, dann zurück an den Anfang, also zu "do", sonst weiter runter
```

5f. for - Schleife

Die for-Scleife ist sehr ähnlich der while-Schleife. Um sie besser zu vergleichen hier erstmal eine for-Schleife die sich genau wie die while-Schleife verhält.

```
for ( ; Abfrage ; )
{
alle Anweisungen hierhinein und oben leer hinter 2. Simikolon und vor 1. Simikolon;
}
```

Die Deklaration von Variablen und Initialisierung dieser erfolgt weiter oben wie bei while auch.
Hier ist jedoch ein Platz freigelassen. Anders gesagt, hier passen gleich 3 Dinge in den Klammern oben.

Die volle for-Schleife sieht dann so aus, wobei man jedes Element freilassen kann:

```
for (Deklaration/Initialisierung/Wert zuweisen; Abfrage; Anweisung )
{
Weitere Anweisungen;
}
```

Der Vorteil ist: Man kann hier Variablen deklarieren und initialiserien (mit Werten belegen) die man nur für die Schleife braucht. Sowas kommt öfters vor. Das beste Beispiel dafür ist, wenn man die Durchläufe der Schleife genau festlegen will wie oben.

Zusätzlich kann in der for-Schleife auch noch eine Anweisung oben stehen, diese kann man entweder freilassen wie hier:

```
for (Deklaration/Initialisierung/Wert zuweisen; Abfrage; )
{
alle Anweisungen hierhinein und oben leer hinter 2. Simikolon;
}
```

Oder man schreibt dort nur die Anweisung rein, wenn man die Durchläufe begrenzen will wie hier:

```
for (int Durchlauf = 0 ; Durchlauf != 5 ; Durchlauf++)
{
alle Anweisungen hierhinein und oben leer hinter 2. Simikolon;
}
```

Übersetzt man dies, könnte das so lauten:

```
Schleife (Ganzzahl namens Durchlauf = 0 ; solange Durchlauf nicht 5 ist ; Addiere 1 zu Durchlauf)
{
und führe dies und das hier aus als weitere Anwesiungen;
}
```

Dies ist dann das selbe wie:

```
for (int Durchlauf = 0 ; Durchlauf != 5 ; )
{
Durchlauf++
alle Anweisungen hierhinein und oben leer hinter 2. Simikolon;
}
```

Um eine Endlosschleuife aus der for-Schleife zu machen, lässt man oben einfach alle frei.
Aber die Simikolons dürfen trotzdem nie fehlen.

```
for (;;)    // wie while(1) auich eine Endlosschleife, Ausweg dann nur mit break noch möglich
{
Anweisungen;
}
```

Eine letzte Möglichkeit von for-Schleifen ist eine mit leeren Rumpf, wenn die Schleife nur eine Anwesiung hat.

```
for (Deklaration/Initialisierung/Wert zuweisen; Abfrage; Anweisung )
{
        ;           // Nur Semikolon, da die eine Anweisung oben ausreichte
}
```

Das Simikolon in Blöcken, also das zwischen { und }, ist notwendig, da kein Block ganz leer bleiben darf.

Welche Schleife man wann nimmt und wofür bleibt jeden selbst überlassen, was man wann besser findet.

Es gibt noch eine ältere Möglichkeit mit “goto”. Da diese aber nicht mehr gebräuchlich ist und Fehler so sehr leicht entstehen, nenn ich diese hier erst garnicht. Für die diese Methode kennen sei bloß gesagt, sie funktioniert noch.

5g. Switch und case

Ähnlich wie Schleifen funktioniert auch “switch” und “case”. “switch” übernimmt dabei eine Variable und mit “case” kann man jede Einzelne testen ohne großes Geteste mit “if”.

Wie funktioniert ein Fahrstuhl? Nun das würde in etwa so laufen:

```
#include <iostream.h>

void Fahre_zu_Stock1();
void Fahre_zu_Stock2();
void Fahre_zu_Stock3();
void losfahren();
void Bleib_stehen();

void main()
{
int etage;
cout << “Welche Etage?”;
cin >> etage;
switch(etage)                                   // übernimmt die Variable “etage”
        {
        case 1:  Fahre_zu_Stock1(); break;     // break = raus aus der Schleife
        case 2:  Fahre_zu_Stock2(); break;
        case 3:  Fahre_zu_Stock3(); break;
        default:  Bleib_stehen();               // standard, falls nichts stimmt, falls man z.B. 4 eingibt
        }
}

void Fahre_zu_Stock1()
{
losfahren();
if (angekommen == true) {anhalten;}
}

void Fahre_zu_Stock2()
{
 ...
}

void Fahre_zu_Stock3()
{
...
}

void Bleib_stehen()
{
...
}
```

Muster von “switch” und “case”:

```
switch (Wertübernahme)
        {
        case Möglichkeit: Anweisungen dafür; break;
        case Möglichkeit: Anweisungen dafür; break;
        case Default: Anweisungen wenn nichts von oben stimmt; break;
        }
```

Wichtig:
Break ist notwendig, da das Programm das andere sonst auch noch ausführt. So wird dann abgebrochen.

6. Ungarische Notation

Dies ist zwar Standard, aber nicht Pflicht und man braucht diese nicht zu beachten. Aber wenn man sich große Quelltexte ansieht, hilft es, sie besser zu verstehen. Gemeint ist die Ungarische Notation, die wie folgt funktioniert:

Man setzt am Anfang eines jeden Namens für eine Variable einen Buchstaben, der den Typ der Variable angibt, damit man auch später noch genau weiß, was dies eigentlich ist.

Typ	Beispiel
b für bool	bool bEingabeOK;
c für char	char cText;
i für int	int iZahl;
n für Number (also int oder long oder short)	int nZahl;
f für float	float fTemperatur;
p für einen Zeiger	int* pZeiger;
C vor Klassen	class CGeometrie;

Die U.N. beschreibt aber nicht nur Variablen genauer, sondern es sollen zum Beispiel alle Wörter in einen Variablennamen/Funktionsnamen mit einen Großbuchstaben beginnen:

```
fTempVonHeute;
iZahlNeu;
cTextDerBeiFehlernErscheint;
```

Desweiteren werden nach dieser Empfehlung Konstanten durchgängig mit Großbuchstaben geschrieben und mit Unterstrich getrennt.:

```
int ANZAHL_GEGNER;
char FEHLER_TEXT;
```

Wie gesagt ist dies kein Muss, aber es sorgt für ein einheitliches Schriftbild unter allen Programmierern. Ich persönlich nutze sie leider hier nur recht selten.

7. Dateien einbinden

7a. Anführungszeichen oder Pfeile

Dateien einbinden kann man auf 2 verschiedene Arten machen:

```
#include “EingebundeneDatei.h”
#include <EingebundeneDatei.h>
```

Bisher wurde immer alles mit den Pfeilen < und > eingebunden, wenn es benötigt wurde. Anführungsstriche statt Pfeile vo und nach dem Dateinnamen bedeuten nun, dass die genannte Datei im gleichen Verzeichnis ist, also im Arbeitsverzeichnis statt im Include-Verzeichnis vom Compiler. Schaue selbst am besten mal nach, wieviel Dateien schon im include-Verzeichnis stehen, sicher eine ganze Menge. Und diese enthalten auch eine ganze Menge an mitgelieferten Code. Schreibt man nun selbst zusätzliche Dateien, so legt man die sicher nicht ins Include-Verzeichnis sondern ins Arbeitsverzeichnis (das wo auch der Rest ist). Und in diesen Fall setzt man schließlich Anführungsstriche.

Warum nun den Code in mehrere Dateien schreiben? - Dafür gibt es 2 Gründe:
1. Wenn man den ganzen Quelltext eines Programmes in ein .cpp-Datei schreibt, sieht man ganz schnell ab ein paar Hundert Zeilen nicht mehr durch. Deswegen teilt man die Dateien so gut wie möglich auf. Die Hauptdatei (main.cpp oder ähnlich) bindet dann andere mit ein.

2. Teilt man diese noch logisch auf je nach dem was die Funktionen in einer Datei machen, kann man diese später wiederverwenden. <iostream.h> zum Beispiel ist für Eingabe und Ausgabe bestimmt. <string> ist für die Nutzung von Strings und <math.h> für die Nutzung von höheren Mathematischen Funktionen.Und ähnlich sollte man sein Programm auch in gleichartige Teile aufteilen.

Und warum haben Codedateien entweder die Endung .h oder .cpp?
---> Die .h-Dateien sind die Headerdateien
---> die .cpp-Dateien sind die reinen Codedateien

Was nun wo steht:

In den Headerdateien:	Kommentare: Was machen die Klassen? Was machen die Funktionen im einzelnen?, Klassengerüste, Strukturen, globale Variablendeklarationen, Funktionsprototypen
In den Codedateien:	Funktionen selbst, die Klassenfunktionen, Möglich hier natürlich auch das andere.

Beispiel einer Aufteilung für ein Spiel:

In Datei	Eingebunden wird
Main.cpp	spieler.h, iostream.h, level.h

Dafür macht man folgendes in der main.cpp

```
#include "spieler.h"       // eingebunden wird spieler.h und die dazugehörige spieler.cpp
#include "level.h"         // " und " weil Datei im aktuellen Verzeichnis
#include <iostream.h>      // < und > weil Datei im Compilerverzeichnis
```

.h steht also für die Headerdateien, aber warum werden nur die immer eingebunden? Leicht zu erklären: Die .cpp-Dateien die genauso heissen, wie die Headerdateien werden automatisch eingebunden.

Ein Beispiel:

```
// main.cpp

        #include "spieler.h"
        #include <iostream.h>

        void main()       // hierfür braucht man keinen Prototyp,  der Compiler sucht auf jeden Fall danach
        {
        ....
        }
```

```
// spieler.h: Prototypen

        #ifndef   _spieler__h_       // wenn schonmal eingebunden
        #define  _spieler__h_       // dann alte gegen neue austauschen
                                     // Erklärung: schützt vor Doppeleinbindung falls mehrere .cpp -Dateien
                                     // die gleiche Datei einbinden wollen

        void Rennen();               // Figur rennt
        void Springen();             // Figur springt

        #endif                       // Ende
```

```
// spieler.cpp: Funktionsinhalt

        void Rennen
        {
        ...
        }

        void Springen
        {
        ...
        }
```

In diesen Beispiel ist nun zu sehen dass in der Headerdatei etwas von #ifndef, #define und #endif steht. Dies muss nicht dort stehen, wenn eine Datei nur einmal eingebunden wird. Aber wenn sie mehrmals eingebunden wird: zum Beispiel eine “musik.h” für Soundfunktionen wird in 5 verschiedenen Dateien eingebunden.

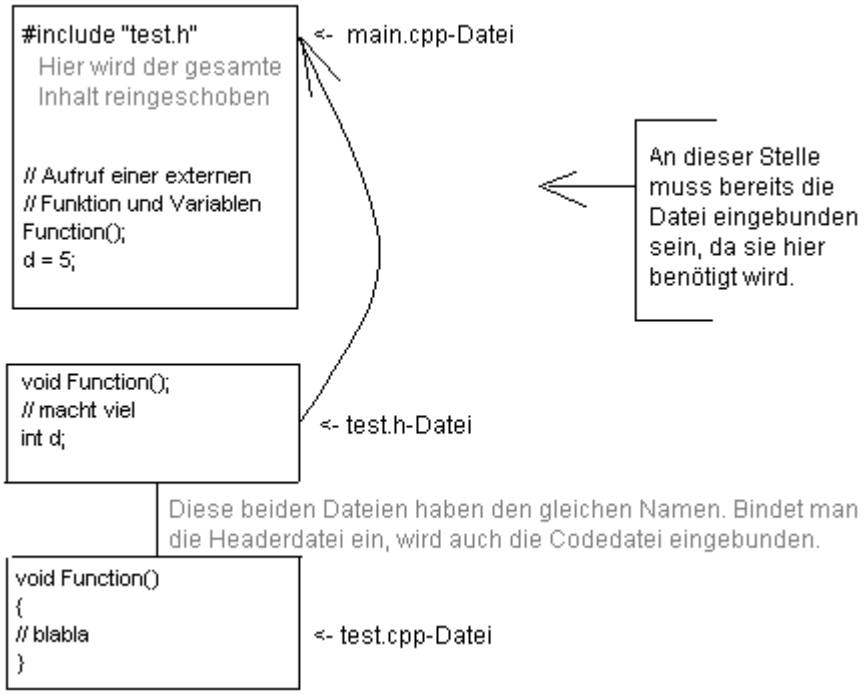
Man kann dies endlos weiterführen, etwas komplexer wäre dies hier, ein Modell für ein Grafikprogramm:

“main.cpp” -> bindet ein eine “audio.h”, “windows.h”, “grafik.h”
“grafik.h” -> bindet ein die “windows.h”, “bmpFormat.h”, “tgfFormat.h”, “zeichen.h”, “FormateAllgemein”
“zeichnen.h” -> bindet ein eine ”FormateAllgemein.h”

Hier würden für den Fall, dass man diese Schlüsselwörter nicht benutzt, manche Dateien mehrfach eingebunden werden.

“main.cpp” kann natülich auch eine “main.h” haben, falls man die Hauptdatei irgendwo anders benötigt. Es können natürlich auch die Funktionsprototypen und Deklarationen in der .cpp-Datei stehen.

Und noch 2 Schlusssätze dazu:
Bindet man andere Dateien ein, so wird deren Inhalt vor dem Inhalt der Datei eingerückt, wo sich der Aufruf befindet. Hierfür sei kurz ein Schaubild dargestellt. Die großen Kästchen stellen Dateien dar.



Bemerkung:
Die Hauptdatei heisst hier immer “main.cpp”, sie kann aber auch beliebig anders heissen.
Die Hauptfunktion “main” muss jedoch “main” heissen.

7b. Codedateien, welche dem Compiler beiliegen:
(es gibt noch viele viele weitere)

windows.h	= Für Menüs, Fenster, Steuerelemente, Messageboxen ... (diese Datei wird hier erstmal nicht benötigt)
iostream.h	= Für Ausgabe und Eingabe (wird hier für “cout” und “cin” gebraucht)
fstream.h	= Für das Schreiben und Lesen in Textdateien

8. Aufzählungen mit “enum”

8a. Aufbau von “enum”

“enum” ist ein Typ um mehrere int-Konstanten gleichzeitig zu erzeugen, also in etwa eine Liste.

```
enum eMonate
{
Januar,
Februar,
Maerz,
April,
Mai,
Juni,
};
```

Eine andere Schreibweise wäre diese, dies wirkt sich jedoch nur optisch aus.

```
enum eMonate { Januar, Februar, Maerz, April, Mai, Juni };
```

Syntax (Aufbau):

Um “enum” zu benutzen, schreibt man also das Wort “enum” gefolgt von einen Namen der enum-Liste, welcher frei wählbar ist. Dann folgt ein Block, der mit { anfängt und mit } endet. Und ganz zum Schluss muss ein Simikolon stehen - Hier ist dann die Liste der int-Variablen zu Ende.

Der Block in enum:

Im Block zwischen { und } stehen nun die Konstanten, welche man deklarieren will, mit einen Komma voneinander getrennt. Diese sind vom Typ “int”. Dabei erhält jede Variable einen aufsteigen Wert angefangen von 0 ganz oben: Für den Compiler sieht dann dann also wie folgt aus:

```
enum eMonate { Januar = 0, Februar = 1, Maerz = 2, April = 3, Mai = 4, Juni = 5 };
```

Man kann dies auch selbst so hinschreiben oder aber man belegt selbst die Werte je nach Belieben:

```
enum eMonate { Januar = 0, Februar, Maerz = 5, April, Mai = 10, Juni = 11};
```

Nun sieht das als normale “const int” wie folgt aus für den Compiler:

```
Januar = 0;
Februar = 1;
Maerz = 5;
April = 6;
Mai = 10;
Juni = 11;
```

Wurde ein Wert nicht belegt, so belegt diese auch hier der Compiler mit den nächsten Wert, der folgt. Nach 0 für Januar folgt also 1 und nach 5 für den März folgt 6.

Nun kann man diese Werte auch abfragen. Dazu wird jetzt zur besseren Veranschaulichung ein neuer int namens “w” deklariert und mit den Wert von “April” initialisiert:

```
int w = April;          // heisst dass “int w = 6” ist
```

8b. Nutzung von “enum”

Das folgende Codeteil zeigt eine Nutzungsmöglichkeit:

```
enum eMonate { Januar, Februar, Maerz, April, Mai, Juni };

enum eMonate MONAT;                    // Monat vom Typ eMonate deklarieren

MONAT = Januar;                        // Nun ist MONAT gleich Januar

int zaehler = 0;

while (MONAT != 12)              // solange "MONAT" nicht 12 ist
{
Zaehler++;                       // addiere eins zum Zähler und fange Schleife von neuem an
}
```

“enum” kann den Code vergleichen, wenn man es richtig einsetzt.

9. Arrays

9a. Variante mit Variablen

Hier zuerst ein Beispiel mit Variablen. Danach in 9c wird genau das gleiche Programm umgeschrieben und nutzt dann Arrays. In den Variablen wird der Preis eines Tisches gespeichert von einem Möbelhaus. Dieser Preis soll bei Eingabe des Tisches ausgegeben werden.

```
#include <iostream.h>

void main()
{
int tisch1 = 20;
int tisch2 = 30;  int tisch3 = 35;
int tisch4 = 90;  int nummer;

cout << "Für welchen Tisch den Preis anzeigen: \n";
cin >> nummer;
switch(nummer)
   {
   case 1: cout << "Betrag: \t" <<  tisch1;       // “Betrag:” Tabulator Variablenwert
   case 2: cout << "Betrag: \t" <<  tisch2;
   case 3: cout << "Betrag: \t" <<  tisch3;
   case 4: cout << "Betrag: \t" <<  tisch4;
   }
}
```

Erklärung:
Hier werden oben zuerst die Preise fest eingegeben, dann wird abgefragt, für welchen Tisch man den Preis haben möchte.

Problem:
Es gibt hier 4 Tische. Ein Möbelhaus wird nicht nur 4 Tische haben. Die Anzahl der Zeilen für ein so primitives Programm wäre also riesig. Kommen dann noch ein paar Stühle dazu und will man den Preis auch noch selbst eingeben, wäre das Programm für die paar Funktionen etwas zu groß.

Lösung:
Eine Lösung wären Arrays. Arrays sind ähnlich wie Variablen auch Speicherbereiche, in denen man einen Wert hinterlegen kann. Der Vorteil eines Arrays ist aber, dass man mehrere Felder hat. Es ist also eine Zusammenfassung von Variablen gleichen Datentyps und gleichen Namens.

9b. Deklaration und Zuweisungen von Arrays

Muster

```
Variablendatentyp  Name  [Felderanzahl];
```

Neu sind hier die eckigen Klammern, dort schreibt man die Anzahl der Felder hin oder anders, wie oft man diese Variable haben möchte. Will man 5 Mal die Variable “Tisch”, kommt dort die Zahl 5 hinein. Will man 1000 Mal die Variable Möbelstück, kommt dort die Zahl 1000 hinein.

Einzelzuweisung

```
Testarray[0] = Wertzuweisung;          // ohne Typ davor, da dies eine normale
                                       // Zuweisung wie bei Variablen ist
Testarray[1] = Wertzuweisung;

Testarray[354] = Wertzuweisung;
Testarray[23] = Wertzuweisung;
...
```

Die Anzahl der Felder, also das in den Eckigen Klammern, kann später nicht geändert werden.

Deklaration und Zuweisung mit allen Werten gleich bei Beginn:

```
int Testarray[4] = { 40, 20, 10, 35 };        // man spart 3 Zeilen gegenüber Variablen
```

oder

```
int Testarray[] = { 40, 20, 10, 35 };         // Compiler zählt selbst und ordnet diese ein,
                                              // bloß [ und ] sind notwendig
```

Beispiel einer Änderung vom 4. Feld:

```
Testarray[3] = 34;                            // zu zählen beginnt man bei Arrays mit 0
```

Der “Testarray” hat 4 Felder. Das heisst, man kann jeden einzelnen der 4 Felder mit Werten belegen.
Im folgendes werden alle mit 123 belegt:

```
Testarray[0] = 123;
Testarray[1] = 123;
Testarray[2] = 123;
Testarray[3] = 123;
```

Der “Testarray” hat 4 Felder aber warum geht es dann nur bis 3? Weil man mit 0 anfängt zu zählen.
0 ist die erste Zahl für den Compiler.Es folgt dann 1, 2, 3, 4, 5, usw. Das verwirrt manchmal etwas.

Aber mal ein anderes Beispiel:
Wie würde nun ein Array mit 7 Wochentagen aussehn vom Datentyp “float”?
In jeden soll die Tagestemperatur gespeichert werden:

```
float fWochentage[7];             // 7 Mal die Variable “Wochentage” deklarieren als Array

fWochentage[0] = 32.6;            // Temperatur für den 1. Wochentag in Feld 0 speichern
fWochentage[1] = 22.1;            // Temperatur für den 2. Wochentag in Feld 1 speichern
fWochentage[2] = 27.6;            // Temperatur für den 3. Wochentag in Feld 2 speichern
fWochentage[3] = 26.7;            // Temperatur für den 4. Wochentag in Feld 3 speichern
fWochentage[4] = 29.9;            // Temperatur für den 5. Wochentag in Feld 4 speichern
fWochentage[5] = 32.2;            // Temperatur für den 6. Wochentag in Feld 5 speichern
fWochentage[6] = 36.3;            // Temperatur für den 7. Wochentag in Feld 6 speichern
```

Und was passiert wenn man aus Versehen Feld 7 benutzt, welches NICHT vorhanden ist?
Dann fängt entweder der Compiler den Fehler ab und gibt eine Fehlermeldung aus oder das Programm läuft und es passieren merkwürdige Sachen, da man irgendwo was in den Speicher geschrieben hat, was dort nicht hingehört.

9c. Tischbeispiel nun mit Arrays

Schreibt man das Tischbeispiel von 9a. oben um (Array statt Variablen), kommt dies heraus, was schon sehr viel kürzer und klarer ist:

```
#include <iostream.h>

void main()
{
int tisch[4] = { 20, 30, 35, 90 };
int nummer;
cout << "Für welchen Tisch den Preis anzeigen: \n";
cin >> nummer;
cout << "Betrag: \t" <<  tisch[nummer];     // der PC sieht statt "nummer" hier den Wert des Feldes
}
```

10. Zeiger

10a. Definition Speicheradressen

Ein Zeiger ist eine Variable. Diese Variable speichert jedoch keinen Wert (also Zahl oder Text) sondern eine Speicheradresse.

Vom Datentyp der Variablen hängt nicht nur die Größe der Zahl und was diese Variable beinhaltet (Text, Zahl, Bool) ab, sondern auch die Bytegröße.

Bei der Deklaration und Initialisierung von Variablen wird also ein Platz im Speicher reserviert. Das suchen von freien Stellen im Speicher und das Belegen macht dabei der PC selbst.

Bildliches Modell eines Speichers:

Speicheradresse	Inhalt
12001:	int-Variable aepfel
12002:	auch noch, da int 2 Bytes hat
12003:	bool richtig_oder_falsch
12004:	float Summe
12005:	auch noch, da Float 4 Bytes hat
12006:	auch noch, da Float 4 Bytes hat
12007:	auch noch, da Float 4 Bytes hat
12008:	noch frei
12009:	noch frei
12010:	int-Variable zaehler
12011:	auch noch, da int 2 Bytes hat
.....	.....
usw.	usw.

In diesen Speicheradressen stehen also Werte. Der Compiler braucht bloß den Datentyp kennen, damit er weiß wieviel Speicher er wofür genau reservieren soll.

Ein Zeiger ist nun eine Variable, die genau auf solch eine Adresse zeigt. Um von einer Variablen die Speicheradresse rauszubekommen, schreibt man vor der Variablen ein “&”.

```
int anzahl = 3;          // Ganzzahl-Variable namens “anzahl” und initialisiert mit 3
cout << anzahl;          // Ausgabe des Variableninhalts
cout << &anzahl;         // Ausgabe der Speicheradresse der Variable “anzahl”
```

Dieses sieht dann in etwa so aus bei der Ausgabe:

```
3                        // cout-Zeile1
0x4fd9:d111              // cout-Zeile2
```

Jetzt weiß man die Speicheradresse von der Variable "anzahl". Diese lässt sich auch speichern, aber wo? In einen int nicht da so eine Speicheradresse voller Sonderzeichen sein kann, in einen char auch nicht, und in einen bool erstrecht nicht.

Genau dafür braucht man einen neuen Datentyp, der oben einfachhalber nicht auftauchte.
--> Nämlich ein Zeiger
Der Zeiger speichert die Speicheradressen von Variablen. Es ist sozusagen das passende Schuhfach für den Schuh.

10b. Definition Zeiger

Zeiger werden ähnlich wie Variablen deklariert. Dabei muss der Datentyp der gleiche sein wie die Variable selbst:

```
Datentyp Sternchen Zeigername = Adresse oder 0 oder NULL;
```

Weiß man die Adresse noch nicht, so schreibt man 0 oder NULL.
0 ist in C++ das gleiche wie NULL, wenn man “Nichts” meint.

```
int anzahl = 3;                          // Initialisierung mit 3
cout << "Apfel: " << anzahl;             // Ausgabe der Variable (Wert)

cout << "\nSpeicheradresse von der Variable anzahl: " << &anzahl;

int* zeiger_für_anzahl = &anzahl;        // Speicheradresse wird in Zeiger gespeichert
```

Im obigen Beispiel steht im Zeiger die Adresse von der Variable "anzahl". Das heißt wenn man jetzt den Wert ausgibt, der dort steht, dann ist dieser der gleiche. Genauso gilt dies nach Veränderungen von einen der beiden (Wert auf Zeiger oder Variablenwert):

```
// Zeiger und Variable anlegen, Zeiger dabei mit NULL initialisieren.
int* z_temperatur = NULL;
int temperatur = 30;

// im Zeiger die Adresse von der Variablen speichern
z_temperatur = &temperatur;

// die aktuele Temperatur eingeben
cout << "Bitte Temperatur eingeben: "
cin >> temperatur;

// Zeilenumbruch Möglichkeit 1 im Text, also zwischen den Anführungszeichen.
cout << "\n";

// dann wieder ausgeben, diesmal aber mit Hilfe des Zeigers
// danach Zeilenumbruch Möglichkeit 2
cout << "Die aktuelle Temperatur ist " << *z_temperatur << endl;
```

Ausgegeben wird der geänderte Wert, da ja “z_temperatur” auf “temperatur” zeigt. Dies bleibt dann auch das ganze Programm über, außer man übergibt dem Zeiger wieder den Wert NULL.

Das Sternchen vor dem Zeiger heisst, das der Wert gemeint ist, der sich hinter der Adresse verbirgt. Nach einen Datentyp wie ganz oben bedeutet ein Sternchen, dass man einen Zeiger deklariert. Verwendet man das Sternchen ohne Datentyp irgendwo, steht dies für den Wert in dieser Adresse (also die Adresse von der Variablen).

```
*z_temperatur = 25;      // Wert wird verändert, und zwar auf der Adresse, also wird bei
cout << z_temperatur;    // der Ausgabe der ariablen jetzt auch 25 angezeigt
```

Bildliche Darstellung des Speichers

```
12001:          int-Variable “temperatur” mit Inhalt 30 bzw. 25 von oben
12002:             auch noch, da int 2 Bytes hat
12003:          .......
12004:          int-Zeiger “z_temperatur” mit Speicheradresse von der obigen Variable,
12005:             ändert man hier also den Wert, ändert man ihn eigentlich oben,
12006:             da der Zeiger ja auf obige Variable zeigt, das gleiche auch umgekehrt
.......         .........
usw.            usw.
```

10c. Wozu braucht man Zeiger

Zeiger haben den Vorteil dass sie global gelten, sie werden auch global deklariert. Zeiger werden bei der Windows-Programmierung sehr oft verwendet. Zur Wiederholung: Global heisst ausserhalb von Funktionen. Variablen sollten möglichst nicht global sein.

10d. "new" und "delete"

Und wieder 2 neue Wörter.Und wieder haben diese was mit den Speicher zu tun. Speicher ist wichtig für Variablen, Arrays und Klasseninsatnzen, denn dafür muss Speicher bereit gestellt werden. Schließlich müssen die Daten ja irgendwo stehen.

Wenn man Variablen oder Arrays deklariert, weiß man für gewähnlich schon die Anzahl der Felder bei Arrays oder aber die Größe der Variablen.Das nennt man statische Speicherbereitstellung. Der Speicher und dessen Größe lassen sich auch nicht mehr ändern. Und wenn man immer nur kurzzeitig ein paar Variablen braucht? Dann muss man gleich ganz zu Beginn diese deklarieren und kann diese auch nicht mehr löschen. Der Speicher wird unnötig belegt. Oder aber man benötigt ein Array und weiß aber die Felderzahl zum Beginn noch nicht. Es könnte auch sein, dass die Felderanzahl sich ständig ändert.
Soll man dann sicherheitshalber einen Array mit 10000 Felder deklarieren? Und wenn man nur jeweils maximal 100 Felder braucht? Und soll man wenn man die Größe von ein paar "chars" nicht weiß, erstmal die vollen 255 Zeichen deklarieren. Und wenn man nur 5 jeweils braucht? Nein, das wäre alles eine sehr große Verschwendung von Speicherplatz. Gut, wer 30 GB RAM hat, kann dies gern tuen, aber die Programme und Spiele sollen ja auch auf Rechnern mit normaleren 16 oder 32 MB Ram laufen.

Die Lösung:
Ein besserer Weg sind dynamische Speicherverwaltung. Dynamische Variablen und Arrays sind nicht gleich zu Beginn da, sondern erst wenn sie benötigt werden. Arrayfelder können einfach hinzugefügt werden. Neue Instanzen von Klassen können einfach hinzugefügt werden.

10e. "new"

Das Schlüsselwort "new" ist nun dafür da, eine neue Variable oder einen neuen Array oder eine neue Klasseninstanz anzulegen. Und jetzt noch eine Antwort warum das ganze unter Zeiger steht: Der Rückgabewert von "new" ist die Speicheradresse des grad reservierten Speichers. Und nun mal als Muster ein paar Zeilen Code:

```
new int;                    // falsch
```

Das geht zwar aber ergibt keinen Sinn, hier wird Speicher bereitgestellt für eine neue int-Variable, die man irgendwo irgendwann benötigt, aber man muss natürlich auch den Rückgabewert, also die Adresse des Speichers, speichern. Neuer Versuch aber davor sei nochmal ein wichtiger Satz wiederholt: Zeiger sind Variablen, die Adressen von Speicher speichern.

```
int* pZeiger = NULL;    // erstmal hat der neue Zeiger hier keine Adresse gespeichert
pZeiger = new int;
```

zu Zeile 2: Hier wird wie im ersten Code, einen Absatz weiter höher, eine neue int-Variable erzeugt, das kann irgendwo im Programm sein, wo man zum Beispiel noch ein int für eine Zahl braucht. Aber diesmal wird der Rückgabewert, nämlich die Speicheradresse in einen dafür vorgesehen Zeiger gespeichert. Dieser Zeiger wurde in der ersten Zeile deklariert und mit NULL initialisiert.

Bleibt vielleicht noch die Frage warum der Zeiger vom int-Typ ist. Nun, das steht oben bei Zeigern: Der Zeigertyp muss der gleiche sein, wie der der Variablen.

Jetzt werden zwei leere Zeiger für das was dann einen Absatz später kommt deklariert:

```
float* pZeiger2 = NULL;          // Leerer Zeiger für Adressen von Kommazahlen
double* pZeiger3 = 0;            // Leerer Zeiger für Adressen von Kommazahlen
```

Zur Wiederholung: “0” ist gleich “NULL” wenn es für “Nichts” oder “Leer” steht und “double” sind Kommazahlen mit mehr Stellen hinter dem Komma. Die Belegung von diesen Speicher könnte so aussehen:

```
pZeiger2 = new float;
pZeiger3 = new double;
```

Kann man Zeiger auch in und der selben Zeile deklarieren und gleich eine Adresse hineinschreiben? Ja, das geht und so lassen sich die 4 bisherigen Codezeilen auf dieser Seite auf 2 kürzen:

```
float* pZeiger2 = new float;          // beides zusammen erledigen: Zeigerdeklaration und ...
double* pZeiger3 = new double;        // ... Adresse reinschreiben vom neuangelegten Speicher
```

Ob man zuerst einen Zeiger deklariert und diesen erstmal leer lässt und später genau diesen Zeiger dann mit einer Adresse von den neu reservierten Speicher füllt, bleibt jeden selbst überlassen.

Auf diese Art und Weise werden also dynamisch neue Variablen angelegt und die Adresse des Speichers dort wird in einen Zeiger gespeichert. Bei Arrays sieht das ganz ähnlich aus:

```
int* pZeiger5 = new int[200];
```

Der Datentyp des Zeigers bleibt “int”, denn in diesen soll ja eine Adresse zu den folgenden Array oder der Variable vom Typ “int” stehen. Nach den “new” folgt hier also ein dynamisch erstellter Array. Ebendso könnte dort das stehen:

```
...
int wieviel = 20;                     // erfährt man erst im Programm an einer bestimmten Stelle
int* pZeiger5 = new int[wieviel];     // und genau so einen großen Array erstellt man dann auch
...
```

10f. “delete”

Das Schlüsselwort “delete” löscht den mit “new” erstellten Speicherbereich wieder. Wenn man den Speicherbereich nicht wieder löscht, entstehen sogenannte “Memory Leaks”. Da sind Stellen im Speicher die selbst angelegt wurden durch “new” aber nicht nicht wieder gelöscht wurden.Oder aber man fordert 2 Mal einen Speicher mit “new” für das selbe an. Dann kann man an den ersten Speicherbereich nicht mehr herankommen und auch nicht mehr löschen. Auch das gibt ein “Memory Leak”.

Was macht man, wenn man alle 4 Speicherbereich wieder löschen möchte, die oben erstellt worden sind? Bei statischen also auf herkömmliche Weise erstellte Arrays und Variablen ist dies nicht nötig, aber bei dynamisch erstellte dafür. Dies wird dann wie folgt gemacht:

```
delete pZeiger;                  // so löscht man dynamisch erstellte Variablen
delete pZeiger2;                 // so löscht man dynamisch erstellte Variablen
delete pZeiger3;                 // so löscht man dynamisch erstellte Variablen
delete pZeiger4;                 // Fehler: hier ist nichts zu löschen, “pZeiger4” gibt es nicht
delete pZeiger5[];               // und so löscht man dynamisch erstellte Arrays
```

Es muss nur “delete” gefolgt vom Speicherbereich hingeschrieben werden. Bei Arrays muss hinter den “delete” noch “[]” stehen, dies aber ohne Inhalt sondern einfach nur die beiden eckigen Klammern.

11. Strukturen

Strukturen sind Sammlungen von Variablen. Dies dient der Übersicht, Vereinfachung und man kann endlich auch mehr als einen Wert von einer Funktion zurückgeben (nämlich als Rückgabe gleich eine ganze Struktur).

Zur Erläuterung ersteinmal ein paar normale Variablen:

```
int energie;                // kein guter Name, was für eine Energie ist gemeint?
int spieler1_energie;       // besser
int feind1_energie;
int feind2_energie;

int treffer_des_spielers;
int treffer_von_gegner1;
// und so weiter
```

Wenn man das jetzt auch noch mit anderen Eigenschaften macht, hat man schnell mal ein paar hundert Variablen im Raum zu stehen. Gibt es eine Vereinfachung? Ja, man packt das ganze strukturiert in eine Struktur:

```
struct figur                    // Eine Figur und ihre Variablen
{
int energie;
int treffer;
int punkte;:
int position_x;
int position_y;
bool schluessel1_gefunden;      // true oder false
bool schluessel2_gefunden;      // true oder false
};

struct auto                     // Ein Auto und seine Variablen
{
int energie;
int geschwindigkeit;
int gang;:
float position_x;
float position_y;
char fahrername;
};
```

Es wird also “struct” geschrieben gefolgt von den Strukturnamen und ein Block voller Variablen.

Wenn man diese Struktur im Code hat, kann man neue Typen erstellen vom Typ "spieler" oder "auto". Erst wenn man diese neuen Typen erstellt hat, kann man auf die Variablen des Typen zugreifen. Beispiele:

```
figur spieler;
figur feind1;
figur feind2;
figur held;
auto opell;
auto auti;
auto fort;
auto merzedes;
```

Und die Strukturen sind dabei wie Typen, also im Vergleich dazu in etwa sowas hier:

```
int level;
float temperatur;
```

Und bei dieser Hauptfigur kann man nun die Variablen des Typen nutzen, die in der Struktur stehen. Dafür setzt man zwischen Typ und Variable einen Punkt. Die Variablen kann man dann nutzen wie andere Variablen auch.

```
spieler.punkte = 200;
spieler.schlüssel1_gefunden = false;                    // zu Beginn

if ((tueroeffnen == 1) && (schluessel1_gefunden == true))     // Wenn Tür geöffnet werden soll
{                                                              // und die Hauptfigur den Schlüssel hat
geh_in_raum1();                                                // diese Funktion aufrufen
}
else
{
cout << "Die Tür ist abgeschlossen";
}

if (feind1.treffer == true)
{
spieler.energie = spieler.energie - 50;
}
```

Nochmals sei gesagt: Eine Struktur kann wie ein Typ gesehn werden. Es muss immer erst ein Typ erstellt werden. Wenn man das folgende so schreiben würde

```
auto.gang = 4;
figur.punkte = 22222;
```

dann wäre das genauso falsch wie:

```
int = 5;
float = 204;
```

Auch Funktionen kann man in Strukturen schreiben. Dies macht man aber selten, denn dafür gibt es Klassen.

Für folgendes Text wird das Wissen benötigt was ein Vektor ist. Ein 3er-Vektor hat 3 Punkte: x, y und z. Und kann zum Beispiel eine Raumkoordinate sein im All.

Und jetzt wird mal ein höchst praktisches Beispiel aufgeführt. Stellen wir uns vor, man braucht einen 3er-Vektor und es gibt keinen Vektortyp. Dann bastelt man sich einfach einen:

```
struct vektor
{
float x;
float y;
float z;
};
```

Und nun kann “vektor” genau wie “int” oder “float” verwendet werden:

```
int groeße;
char name;
bool irgendwas;
vektor koordinaten;
```

Man könnte sich auch einen 2er-Vektor basteln nur mit x und y für die Bildschirmkoordinaten.

Und wenn eine Funktion mal einen 3-teiligen Wert wie einen 3er-Vektor zurückgeben soll, kann man dies mit Hilfe von Strukturen.

12. Klassen

12a. Definition

Klassen sind ähnlich wie Strukturen, sie kamen erst mit C++ neu hinzu und werden ähnlich verwendet wie Strukturen mit der Ausnahme, dass man statt "struct" dann "class" schreibt, und diese auch voneinander ableiten kann (Vererbung). Der Zweite Unterschied ist die Bereichseinteilung in “public”, “private” und “protected”. Funktionen werden in Strukturen äußerst selten verwendet obwohl dies möglich ist, also könnte man dies als 3. Unterschied nehmen.

Eine Musterklasse:

Headerdatei (.h):

```
class Auto
{
int  benzin;
bool tuerzu;                    // bool ist ein Typ, hier geht dann nur entweder richtig oder falsch
int anzahl_gaenge;
void fahren();
void bremsen();
};

// Typen von "Auto" erstellen:

Auto Volfo;
Auto BNW;
Auto Viat;
```

Codedatei (.cpp):

```
BNW :: fahren()          // Funktion der Klasse Auto
{
... Inhalt ...
}

... ... ...
```

Hier werden wie bei Strukturen auch zuerst Typen erstellt. Das nennt sich bei Klassen dann “Instanzen”. Mit Hilfe von “new” und “delete” kann man auch neue Instanzen dynamisch erstellen, die man zu Beginn noch nicht kennt. Es könnte zum Beispiel noch ein eigener Rennwagen in einen Rennspiel gebaut werden oder mehrere Profile in einem Rollenspiel.

Klassen stehen in der Headerdatei (*.h), und die Variablen und Funktionen dazu in der dazugehörigen Codedatei (*.cpp). Dies ist weit verbreiteter Standard aber nicht zwingend erforderlich. Man sollte sich am Standard halten auch wenn dieser manchmal nur empfohlen wird und nicht notwendig ist. Ansonsten versteht man zwar sein Quellcode aber wenn man sich andere ansieht, kann es etwas dauern bis man diesen dann auch versteht.

Hier ein Beispiel für eine Klasse, wobei die ganze Klasse in der Codedatei steht und die Funktionen diesmal in der Klasse selbst stehen. Auch dies ist möglich. Es sollte aber nur bei sehr kleinen Klassen gemacht werden und sehr kleinen Klassenfunktionen. “Funktionen in Klassen”, “Klassenfunktionen”, “Klassenmethoden”, “Memberfunktionen” oder “Methoden” meint ein und dasselbe. Häufig werden nur Methoden zu den Funktionen in Klassen gesagt.

```
class Auto
{
int  benzin;
bool tuerzu;                    // bool ist ein Typ, hier geht dann nur entweder richtig oder falsch
int anzahl_gaenge;
void fahren()
        {
        ...
        }
void bremsen()
        {
        ...
        }
}
```

12b. Konstruktor und Destruktor

Mit Konstruktur und Destruktur sind Funktionen der Klasse gemeint. Diese Funktionen heissen genauso wie die Klasse. Wofür man diese nutzt, steht hier in den Kommentaren im Konstruktor und Destruktor.

```
Auto :: Auto()
{
// ...für Initialisierungen
}

Auto :: ~Auto()       // mit Tilde-Zeichen davor
{
// für Aufräumarbeiten wie Speicherbereinigung u. ä.
 // Destruktor wird automatisch erstellt, wenn man keinen hinschreibt
}
```

Der Konstruktor ist für die Initialiserung zuständig, der Destruktor leistet die Aufräumarbeit beim Programende. Beides ist nicht notwendig. Diese haben aber auch den Vorteil, dass sie automatisch aufgerufen werden.Natürlich können auch noch, wie bei anderen Funktionen auch, Parameter übergeben werden und eine Überladung des Konstruktors wie bei Überladungen von Funktionen ist auch möglich.

Bastelt man sich einen eigenen Konstruktor und Destruktor, was man möglichst vernmeiden sollte, sehe das wie folgt aus:

```
Auto :: Init()
{
// ...für Initialisierungen
}
Auto :: Ende()
{
// für Aufräumarbeiten wie Speicherbereinigung u. ä.
 // Destruktor wird automatisch erstellt, wenn man keinen hinschreibt
}
```

12c. private, public, protected

Dies ist der eine große Unterschied zu Strukturen. Dazu nochmal das Beispiel von oben, diesmal nur mit diesen Schlüsselwörtern:

```
class Auto
{
public:                         // kein Simikolon sondern einen Doppelpunkt kommt dahinter
      int benzin;
      bool tuerzu;
      int anzahl_gaenge;
      void fahren()
      {
      ...
      }
      void bremsen()
      {
      ...
      }
private:
     bool ist_tuer_zu;
}
```

Hier tauchen die Wörter "public", “protected” und "private" auf. Diese Wörter stehen einmalig in der Klasse und in unbestimmter Reihenfolge vor dem Bereich von Funktionen und Variablen die das sein sollen. Hier soll also alles öffentlich sein und nur “ist_tuer_zu” soll privat sein.

“Public”, “protected” und “private” heißen übersetzt öffentlich, geschützt und privat und bedeutet folgendes:

Public:

Funktionen hier drunter können auch von anderen Funktionen, welche nicht in der Klasse sind und anderen Klassenmethoden aufgerufen werden. Sie sind also öffentlich zugänglich.

Private:

Nur Klassenmethoden aus der eigenen Klasse können diese Funktionen und Variablen nutzen. Steht also irgendwo im Quelltext das hier:

```
BNV.tuer_ist_zu = false;
```

Gibt dies einen Fehler, denn das können nur Funktionen aus der Klasse machen, und zwar so hier

```
tuer_ist_zu = false;                // irgendwo in einer Klassenfunktion
```

Ist das nicht auch ein Fehler weil da “BNV”, also die Instanz fehlt? NEIN, denn innerhalb der Klasse schreibt man dies ohne Instanz. Siehe dazu auch “12d”.

Protected:

Nur abgeleitete Klassen können diese Funktionen und Variablen nutzen.
Zur Ableitung sei später noch was gesagt.

12d. Nutzung von Variablen und Funktionen von Klassen

Variablen und Funktionen der Klassen können auch kurz gesagt einfach nur Member heissen oder Klassenmember. Im folgenden Codeteil werden von der obigen Autoklasse 3 Instanzen gebildet. Dann wird was am Benzinstand verändert einmal von innerhalb der Klasse und einmal von ausserhalb. Und am Ende wird die Funktion “bremsen” gebraucht, auch diese von innen und außen.

```
Auto Volfo;
Auto Merzedes;
Auto Porshe;

Porshe.benzin = 40;     // Nutzung von aussen irgendwo aus einer anderen nicht-Klassen-Funktion
Benzin = 40;            // Nutzung von innen von einer Klassenfunktion, z.b. von “fahren”

Merzedes.fahren();      // Nutzung von außerhalb
fahren();               // Nutzung von innerhalb der Klasse aus einer anderen Klassenmethode
```

12e. Ein Mitarbeiter

Man nehme ein Programm, dass Mitarbeiter verwaltet. Soll man nur Variablen in diesen Sinne hier entwerfen? Es wäre zwar möglich, aber nicht objektorientiert, schwer zu lesen und wenn man neue Mitarbeiter hinzufügen möchte? Und wenn man Mitarbeiter raus haben möchte? Geht so nicht.

```
float Mitarbeiter_Thomas_Gehalt = 100;
int Mitarbeiter_Thomas_Urlaub_Gesamt = 20;
int Mitarbeiter_Thomas_Urlaub_genommen = 10;
int Mitarbeiter_Thomas_Urlaub_rest;
int Mitarbeiter_Thomas_Fehlzeiten = 10;
int Mitarbeiter_Thomas_Arbeitsort = 2;
bool Mitarbeiter_Thomas_istVollzeit = false;

// und so weiter für weitere Daten

// und so weiter für 200 andere Mitarbeiter
```

Wenn man das so machen würde, wären das tausende Zeilen und kein Ende...
Man könnte sich auch ein Einwohnermeldeamt einer Großstadt vorstellen mit einen Programm, dass die Adressen verwaltet.

Also würde man hier bzw. müsste man hier (anders gehts garnicht mehr bei den Datenmengen) eine Klasse erstellen wie auf der folgendes Seite es auch gemacht wird.

Eine Mitarbeiter-Klasse:

```
class Mitarbeiter
{
public:
        float Gehalt;
        int Arbeitsort;
        char Straße;
        int Postleitzahl;
        datum eintritt;
        datum vertrag;
        void GehaltZahlen();
        void AdresseAendern();

private:                    // das folgende kann nur von Klassenfunktionen genutzt werden
        int Urlaub_Gesamt = 30;
        int Urlaub_Genommen;
        int Urlaub_Rest;
};
```

Natürlich gibts da auch noch mehr Daten aber dies kann ja jeder selbst hinzufügen dann.

Und was ist “datum” für ein Typ? Diesen muss man sich selbst basteln:

```
struct datum
{
int Tag;
int Monat;
int Jahr;
};
```

Dann kann mit diesen Typen gecodet werden:

```
eintritt.Tag = 1;
eintritt.Monat = 1;
eintritt.Jahr = 2;
```

Und Mitarbeiter können jetzt auch instanziert werden:

```
Mitarbeiter Thomas;
Mitarbeiter Hein;
Mitarbeiter Steffen;
Mitarbeiter Juergen;
Mitarbeiter Alex;
```

Und mit “new” und “delete” können diese auch dynamisch erstellt und wieder gelöscht werden. Das heisst man sich das Programm so aufbaut, dann braucht man nicht bei jeden neuen Mitarbeiter das Programm neu zu compilieren, damit dann der Mitarbeiter auch drin ist sondern sagt einfach “new Mitarbeiter Sowieso”.

12f. Friend-Klassen

```
class Azubi
{
friend Mitarbeiter;                    // Hat zum Freund die Mitarbeiter-Klasse
public:
        int Arbeitsort;
        char Straße;
        int Postleitzahl;
        datum eintritt;
        datum vertrag;
        void GehaltZahlen();
        void AdresseAendern();

private:                        // das folgende kann nur von Klassenfunktionen genutzt werden
        int Urlaub_Gesamt = 30;
        int Urlaub_Genommen;
        int Urlaub_Rest;
        float Gehalt;
};
```

Durch “friend <andere Klasse>” wird nun festgelegt, dass die genannte Klasse auf alle Daten der eigenen Klasse zugreifen darf. Die Mitarbeiterklasse darf also nun die Daten in der Azubi-Klasse nutzen
--> Zum Beispiel um nach einer Übernahme die Daten zu übernehmen durch einen einzigen Funktionsaufruf.

Aber darf die Azubiklasse auch die Mitarbeiterklasse einsehen? Nein. Nur wenn dort in der Mitarbeiterklasse auch die Azubiklase als Freund genannt ist:

```
class Mitarbeiter
{
friend Azubi;     // erst jetzt darf die genannte Klasse an meine Daten
...
...
```

Ansonstern dürfte die Mitarbeiterklasse zwar die Azubiklasse nutzen aber nicht umgekehrt.

12g. Abgeleitete Klassen

Angenommen es gibt in der Firma Angestellte, Arbeiter, Azubis, Teilzeitkräfte. Man könnte für jede Gruppe eine eigene Klasse erstellen, da die Daten nicht die gleichen sind (Arbeiter bekommen Lohn statt Gehalt usw.). Aber die meisten Daten sind jedoch die selben. Was macht man nun? Zuerst wird eine allgemeine Mitarbeiter-Klasse für alle Gruppen erstellt:

```
class Mitarbeiter
{
int Urlaub_Gesamt = 30;
// und so weiter
};
```

Und nun leitet man Klassen ab. Die abgeleiteten Klassen haben dabei auch die Members (Variablen und Funktionen) die die alte Klasse (hier also von class Mitarbeiter) hat.

Das geht dann so:

```
class Azubi : public Mitarbeiter              // class Azubi wird abgeleitet von class Mitarbeiter
{
// nur Azubispezifische Daten
};
```

Neu ist hierbei also der Doppelpunkt (und zwar hier nur einer) und die Klasse von welcher abgeleitet werden soll.

12h. “static”

Bei fast jeden Abschnitt lernt man ein neues Wort. Diesmal geht es um “static”. Wo steht dies nun und was hat es zu bedeuten?

Mal angenommen man packt in die Mitarbeiter-Klasse noch folgendes mit hinein:

```
static int Anzahl;
static void GebeMitarbeiterAnzahlAus();
```

Gut, wo da steht ist jetzt klar. Nämlich vor Deklrationen von Funktionen und Variablen.Die Nutzung von diesen erfolgt wie üblich:

```
Anzahl = 12;
```

Aber was heisst dies nun genau? Was ist wenn man den nächsten Quellcode aufruft von außerhalb der Klasse?

```
Steffen.Anzahl = 12;
Thomas.GebeMitarbeiterAnzahlAus();
```

Was hat der Mitarbeiter Thomas nun mit dieser Ausgabe zu tun und was hat der Mitarbeiter Steffen mit den 12 Mitarbeitern zu tun die insgesamt drin sind? Garnichts stimmt, man könnte dies auch als globale Variable irgendwo schreiben außerhalb der Klasse oder wie hier innerhalb der Klasse und dann stratic davor. Dies heisst dann das dies für alle Instanzen (hier also für alle Mitarbeiter) gleich ist. Statt des grad gezeigten Codes könnte man auch schreiben:

```
Thomas.Anzahl = 12;
Ulf.GebeMitarbeiterAnzahlAus();
```

Und wenn man dann die Anzahl ausgibt, dann ist es egal von welchen Mitarbeiter, statische Variablen und Funktionen von Klassen sind über alle Instanzen hinweg gleich. Ob nun..

```
cout >> Thomas.Anzahl;
```

.. dasteht oder ..

```
cout >> Ulf.Anzahl;
```

... ist egal. Damit wollt ich jetzt demonstrieren wie sich statische Variablen und Funktionen verhalten. Es gibt schon noch Gelegenheiten das richtig zu nutzen (welche hier einfach nicht reinpassen), drum sollte man hier das besser doch als globale Variable schreiben irgendwo außerhalb von der Klasse:

```
int Mitarbeiteranzahl = 12;
void GebeMitarbeiterAnzahlAus();
```

13. Dateien lesen und schreiben

Um Dateien lesen/schreiben zu können muss wie folgt “stdio.h” für die Standard Input and Output-Funktionen eingebunden werden:

```
#include <stdio.h>
```

Die folgenden Texte beziehen sich auf das Dateihandling mit C++. Es gibt in der WindowsAPI (Fensterbasierend) noch Funktionen, um Texte in Editfelder zu laden und aus diesen Editfeldern heraus wieder in Variablen hineinzulesen.

13a. Öffnen und Schließen

Um eine Datei nutzen zu können, benötigt man eine Dateivariable (Eine Variablentyp, in der eine Datei steht). Im folgenden “musterdatei” genannt. Bevor man irgendetwas macht mit einer Datei, braucht man dies zuerst.

```
FILE* musterdatei;        // kann irgendwo stehen, Hauptsache vor der Nutzung
```

Dann wird die Datei geöffnet. Theoretisch kann das auch irgendwo stehen vor der Nutzung, praktisch jedoch besser erst kurz bevor man die Datei auch offen haben muss.

```
musteratei = fopen( "test.txt", "w" );        // mit Anführungszeichen jeweils da dies ein String ist
```

Das beides zusammen ist auch nur mit einer Zeile möglich. Der Buchstabe “w” im 2.ten Parameter stellt klar, dass die Datei neu überschreiben werden soll. Steht schon was drin, wird dies gelöscht. Folgende Flags (so nennt man das) sind noch möglich für den 2.ten Parameter:

a+	in vorhand. Datei	anhängen und lesen
a	in vorhand. Datei	anhängen
w+	neue Datei	lesen und überschreiben
w	neue Datei	überschreiben (write)
r+	in vorhand. Datei	lesen und schreibenr
r	in vorhand. Datei	nur lesen (read)

Öffnet man eine Datei mit “r” und will was schreiben, gibt dies einen Fehler.

13b. Schreiben

Nun folgt eine kurze Erklärung wie man in Dateien reinschreibt, später wie man draus heraus liest. Wozu man dies benötigt, dürfte klar sein: In einen Programm selbst (die fertige .exe mit eventuell noch ein paar DLL´s) kann man nichts ändern. Also werden Dinge die zum nächsten Programmstart noch da sein müssen gespeichert.

Hier gibt es die Funktion “fprintf”, deren Aufruf etwa wie folgt aussieht.

```
fprintf (Datei, “Flags”, Variable);
```
oder
```
fprintf (Datei, “Flags”, “Textstring”);
```

Die Funktion übernimmt also 3 Parameter: Das Dateiobjekt (Datei muss offen sein!), Flags (=Optionen) und die Variable, dessen Inhalt in die Datei geschrieben werden soll oder gleich ein Text.

Will man nun in “test.txt” die Variable zahl schreiben, die 5 enthält, schreibt man

```
int zahl = 5;
fprintf (musterdatei, “%d”, zahl);
```

Oder gleich die 5 als String.

```
fprintf (musterdatei, “%s”, “5“);
```

Und beim Text “Dies ist ein Text”, der in einer Variable steht, schreibt man

```
char text = “Dies ist ein Text”;
fprintf (musterdatei, “%c”, text);
```

oder

```
fprintf (musterdatei, “%s”, “Dies ist ein Text.“);
```

Der 2. Parameter ist ein Flag. Es stellt klar, was der Inhalt sein soll. Das % darf nicht fehlen.

“%s”	für String, Text, *char
“%d”	für int
“%c”	für char
“%f”	für float, double (double = wie float aber mit mehr Nachkommastellen)

Sollen mehrere Sachen reingeschrieben werden mit einen gemeinsamen Aufruf, schreibt man einfach den 2. und 3. Parameter mehrmals:

```
int zahl1 = 5;
float zahl2 = 2.3;          // Der Punkt steht für ein Komma

FILE* datei2 = fopen( "zahlen.txt", "w" );          // erstmal öffnen

fprintf (datei2, “%d%s%f”, zahl1, “ “, zahl2);      // und schreiben: int - string - int

fclose(datei2);                                     // und schließen
```

Dieses mehrfache reinschreiben geht natürlich auch mit 4, 5 oder gar 20 Variablen.

Geschlossen muss die Datei natürlich auch wieder. Dafür reicht ein einfacher Aufruf von “fclose” mit der Dateivariablen als Parameter. Es kann immer nur mit Dateiobjekten gearbeitet werden, nie mit einen Dateinamen. Sowas wie folgt geht nicht:

```
fclose(“zahlen.txt”);               // sieht gut aus aber nicht möglich und somit falsch
```

Beim zweiten Parameter stehen die Flags einfach hintereinanderweg. Aber warum ist hier noch ein String dazwischen nur mit 2 Leerzeichen? Dazu ein Ausgabebeispiel wie das hier aussehn würde:

```
5  2.3
```

Sowas kann als 2 Zahlen gelesen werden. Und ohne diesen String wäre alles zusammen:

```
52.3
```

Wenn man Dateien schreibt, merkt man dies nicht. Aber bestimmt will man die Datei auch wieder lesen. Und wenn man sie liest, gibts nur noch eine Zahl.

Statt den beiden Leerzeichen geht auch das hier:

```
fprintf (datei2, “%s%d%s%f”, “Nr1=”, zahl1, “\nNr2= “, zahl2);
```

Ausgabe:

```
Nr1=5
Nr2=2.5
```

Und so lässt sich das schon besser ordnen.

13c. Lesen

Vor dem Lesen muss entweder wieder eine FILE-Variable angelegt werden, wenn dies nicht schon erledigt ist, oder man nutzt die selbe wie oben.

```
FILE* datei = fopen(“tralalala.txt”, “r”);
```

Nach dem Lesen muss die Datei wieder geschlossen werden:

```
fclose(datei);
```

Doch jetzt zum Lesen der Datei:

```
char zeile1[100];              // Platz für max. 100 Zeichen einer ganze Zeile aus der Datei
char zeile2[100];              // Platz für max. 100 Zeichen einer ganze Zeile aus der Datei
fgets (zeile1, 100, datei);    // speichert nun maximal 100 Zeichen aus der ersten Zeile
fgets (zeile2, 100, datei);    // speichert nun maximal 100 Zeichen aus der ersten Zeile
```

Man kann auch nur eine char-Variable für eine Zeile reservieren und dazwischen das rausscannen was man haben möchte:

```
char zeile[100];               // Platz für max. 100 Zeichen einer ganze Zeile aus der Datei

fgets (zeile, 100, datei);     // speichert nun maximal 100 Zeichen aus der ersten Zeile
// hier wird gescannt, die Funktion dazu gibt es später
fgets (zeile, 100, datei);     // speichert nun maximal 100 Zeichen aus der ersten Zeile
// hier wird gescannt, die Funktion dazu gibt es später
fgets (zeile, 100, datei);     // speichert nun maximal 100 Zeichen aus der ersten Zeile
// hier wird gescannt, die Funktion dazu gibt es später

// usw.
```

Dies macht man dann am besten als Schleife bis das Dateiende erreicht ist.

Bei jeden Aufruf von “fgets” wird eine Zeile weiter gelesen. Hat die Zeile weniger Zeichen als dort jeweils angegeben, so bleibt der Rest eben leer. Auf keinen Fall wird schon die nächste Zeile angefangen.

Gut, und jetzt hat man den Inhalt in dieser char-Variable, aber was weiter? Dafür gibt es eine Scanfunktion. Die scannt das raus, was man haben möchte und schreibt es wiederrum in Variablen. Also das ganze wieder andersherum wie oben beim schreiben. Sinn des ganzen? Nun wenn das Programm geschlossen wird, und wieder geöffnet würde, wäre alles wieder beim Anfang. Das Programm selbst speichert nichts intern. Die Daten die ein Spiel oder Programm aufnimmt, müssen irgendwo auf der Platte gespeichert werden, sonst sind sie weg beim nächsten Programmstart.

Doch jetzt zurück zu dieser Scanfunktion:

```
sscanf(zeile, "Nr=%d", &Zahl1);
```

Hier wird der String “zeile” gescannt (1. Parameter).
Gescannt wird nach “Nr=%d” (2. Parameter).
Und findet er also eine Stelle wie “Nr=5”. speichert er den “int”. Da dort “%d” steht in der Variable, die im 3. Parameter angegeben ist, passiert dies nach dem “&”. Das “&” gehört nicht zum Variablenname sondern ist nur in etwa zu lesen als “Speichere in der Variablen”.

Und auch hier können wieder mehrere Dinge gleichzeitig rausgelesen werden.

Die Lese- und Schreibfunktionen in C++ sind sehr umfangreich.
Mehr dazu ist auch in einigen Compilern zu lesen. Als mögliche Übung wäre hier ein Programm möglich, dass 10 Werte abfragt für irgendwas, diese Werte dann speichert in einer Datei aber man jederzeit abfragen kann was Nummer 2, 5 oder andere für einen Wert hat.

14. Wie gehts weiter?

Das war eine kleine Einleitung zu C++, wobei das wichtigste kurz erklärt wurde. Ausgelassen wurden genauere Beschreibungen. Mit den Wissen könnte man jetzt z.B. ein Textadventure programmieren, ein Mitarbeiterverwaltungsprogramm oder simple Rechenprogramme, alles natürlich erstmal nur in DOS.