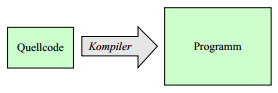
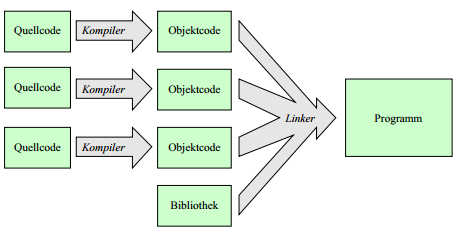
**1.1.**

**Kompilierung** – die Umwandlung von dem Quellekode in die Programmdatei (den Maschinencode), die darin besteht, dass einige Scripts ausgeführt werden müssen, um es alles ermöglichen zu können. Das Endergebnis ist ein Programm, das direkt auf dem Betriebssystem aufsetzt.



Wenn das Programm riesig ist, d.h. wenn es in ein paar Objektdateien geteilt ist, muss man die noch zusätzlich linken.

**Linken** – die Zusammenbindung der Objektcodedateien zum endgültigen Programm.



**1.2.**

**Typ** –  gibt an, von welcher Art die [Daten](http://wirtschaftslexikon.gabler.de/Definition/daten.html) sind, die mit ihm beschrieben werden, und welche Operationen auf diesen ausgeführt werden können.

Datentypen in C++:

* *boolean* – speichert Werte, die nur *true* (wahr) oder *false* (falsch) sein können.
* *char* – legt fest, dass die einzelnen Zeichen eines Speicherbereichs aus je 8 Bits bestehen, die je ein darstellbares Zeichen (Buchstabe, Ziffer, Sonderzeichen,...) repräsentieren.
* *string* – besteht aus einzelnen Zeichen, jedes dieser Zeichen repräsentiert ein char–Datentyp (ASCII).
* *int* – speichert ganzzahlige Werte. Der Wertebereich ist endlich – C++ passt den Datenwert für *int* – je nach Betriebssystem (16–, 32– und 64–Bit) – in der Speichergröße an (in der Tabelle werden verschiedene Wertebereiche für *int* mit einem Stern gekennzeichnet).
* *short* – dieser Datentyp wird genauso wie *int* verwendet (der korrekte Typname heißt *short int*). Man benutzt ihn, wenn keine so große Ganzzahl benötigt wird oder wenn das Programm nicht so viel Speicherplatz verbrauchen darf.
* *long* – entspricht wie der Datentyp *int* auch einer Ganzzahlvariablen (der korrekte Typname heißt *long int*). Bei 16–Bit–Systemen hat *long* einen größeren Zahlenbereich und verbraucht somit auch mehr Speicherplatz als der Datentyp *int*. Der Datentyp *long* wird benutzt, wenn Berechnungen mit größeren Zahlen durchgeführt werden.
* *long long* – der erweiterte *long*–Datentyp (der korrekte Typname heißt *long long int*).
* *float* – lässt Werte aus einem diskontinuierlichen Teilbereich der reellen Zahlen zu. Werte von diesem Typ werden als Gleitpunktzahlen mit einfacher Genauigkeit   
  (6 Stellen) gespeichert. Für den Aufbau einer Gleitpunktzahl gibt es einen IEEE 754 Standard.
* *double* – entspricht wie der Datentyp *float* auch einer Gleitpunktzahl mit doppelter Genauigkeit (15 Stellen).
* *long double* – entspricht wie der Datentyp *float* auch einer Gleitpunktzahl mit erweiterter Genauigkeit (19 Stellen).

|  |  |  |  |
| --- | --- | --- | --- |
| **Datentyp** | **Minimum, kleinster Wert** | **Maximum, größter Wert** | **Speicherplatz** |
| *bool (boolean)* | 0 (false) | 1 (true) | 1 Byte (8 Bit) |
| *Char* | –128 (–27) | +127 (27–1) | 1 Byte (8 Bit) |
| *unsigned char* | 0 | +255 (28–1) | 1 Byte (8 Bit) |
| *int (16 Bit)\** | –32.768 (–215) | +32.767 (215–1) | 2 Byte (16 Bit) |
| *unsigned int (16 Bit)\** | 0 | +65.535 (216–1) | 2 Byte (16 Bit) |
| *int (32 Bit)\** | –2.147.483.648 (–231) | +2.147.483.647 (231–1) | 4 Byte (32 Bit) |
| *unsigned int (32 Bit)\** | 0 | +4.294.967.295 (232–1) | 4 Byte (32 Bit) |
| *int (64 Bit)\** | –9.223.372.036.854.775.808 (–263) | +9.223.372.036.854.775.807 (263–1) | 8 Byte (64 Bit) |
| *unsigned int (64 Bit)\** | 0 | +18.446.744.073.709.551.615 (264–1) | 8 Byte (64 Bit) |
| *Short* | –32.768 (–215) | +32.767 (215–1) | 2 Byte (16 Bit) |
| *unsigned short* | 0 | +65.535 (216–1) | 2 Byte (16 Bit) |
| *Long* | –2.147.483.648 (–231) | +2.147.483.647 (231–1) | 4 Byte (32 Bit) |
| *unsigned long* | 0 | +4.294.967.295 (232–1) | 4 Byte (32 Bit) |
| *long long* | –9.223.372.036.854.775.808 (–263) | +9.223.372.036.854.775.807 (263–1) | 8 Byte (64 Bit) |
| *unsigned long long* | 0 | +18.446.744.073.709.551.615 (264–1) | 8 Byte (64 Bit) |
| *Float* | –3,4E+38 | +3,4E+38 | 4 Byte (32 Bit) |
| *Double* | –1,7E+308 | +1,7E+308 | 8 Byte (64 Bit) |
| *long double* | –1,1E+4932 | +1,1E+4932 | 10 Byte (80 Bit) |

**Objekt** – Informationsstruktur, die Daten zusammenfasst, die einen Zustand besitzt und für die definiert ist, wie sie auf bestimmte Nachrichten mittels vorgesehener Methoden zu regieren hat. Durch den Empfang von Nachrichten können Objekte ihren Zustand verändern oder Informationen über ihren Zustand an den Sender zurückgeben.

**Variable** – ein [Datenelement](http://wirtschaftslexikon.gabler.de/Definition/datenelement.html) oder eine [Datenstruktur](http://wirtschaftslexikon.gabler.de/Definition/datenstruktur.html), die bei der Ausführung des Programms verschiedene Werte annehmen kann. Eine Variable besitzt einen [Datentyp](http://wirtschaftslexikon.gabler.de/Definition/datentyp.html).

**Wert** – der Inhalt, der in einer Variablen gespeichert ist. Der Wert einer Variablen kann jederzeit geändert werden.

**Konstante (*const)*** – repräsentiert einen Wert, der nicht verändert werden kann. Eine Konstante ist entweder eine ganzzahlige Konstante, eine Gleitpunktkonstante, eine Zeichenkonstante oder eine Stringkonstante.

**Typkonvertierung** – mittels Typkonvertierung kann der ursprüngliche Datentyp, z.B. einer Variable, einer Konstante oder eines Ausdrucks, in einen anderen Datentyp konvertiert werden.

Problemfälle, die bei der Typumwandlung auftreten können:

* wenn die Gleitpunktzahl mit „höherer“ Genauigkeit in die jeweils „niedere“ (kürzere) Genauigkeit konvertiert wird (z.B. von *double* in *float*), trifft das Problem der Rundung auf.
* bei der Umwandlung von *float*/*double* in *int* wird der gebrochene Anteil (die Nachkommastellen) abgeschnitten.
* werden allgemein größere in kleinere Ganzzahltypen umgewandelt (z.B. von *int* in *short* oder von *long* in *int*), werden die oberen ("höherwertigen") Bits abgeschnitten, was zur Ergebnisverfälschung führt.

**Initialisierung** – eine Variable erhält ihren Anfangswert. Initialisierung trifft immer auf eine leere Variable. Variablen sollten immer initialisiert werden, um zu vermeiden, dass es mit einem Zufallswert gearbeitet wird.

**1.3.**

**Initialisierung** – eine Variable erhält ihren Anfangswert.

**Zuweisung** – in eine Variable wird ein neuer Wert geschrieben.

Die Initialisierung beginnt immer mit einer Typangabe, der Zuweisung fehlt sie. Eine Initialisierung trifft immer auf eine leere Variable, während bei einer Zuweisung der alte Wert zuerst aus der Variablen entfernt werden muss, bevor der neue Wert darin abgelegt werden kann.

Beispiel:

int main**()**

{  
int y = 8; // initialisiert *y* mit 8  
int z = 10; // initialisiert *z* mit 10

z = y + 5; // weist y + 5 der Variablen *z* zu (d.h. 13)

y = z + 50; // weist z + 50 der Variablen *y* zu (d.h. 63)

}

**1.4.**

**Deklaration** – Programmanweisung, die angibt, wie ein Codeelement verwendet werden kann.

**Definition** – Anweisung, die einen neuen Namen in einem Programm einführt und Speicher für eine Variable reserviert.

Eine Deklaration legt die Bedeutung eines Bezeichners fest (also Datentyp, Speicherklasse, …), es wird aber kein Speicherplatz reserviert. Mit einer Deklaration macht man den Compiler mit einem Bezeichner bekannt und verknüpft man diesen Namen mit einem Datentyp. Bei der Definition wird zusätzlich noch Speicherplatz reserviert und eine Definition darf nur einmal vorkommen. Man kann sagen, dass eine Definition ist eine Deklaration mit der Speicher–Allozierung.

**1.5.**

**Signatur** – die formale Schnittstelle einer [Funktion](http://de.wikipedia.org/wiki/Funktion_(Programmierung)) oder [Prozedur](http://de.wikipedia.org/wiki/Prozedur_(Programmierung)). Sie besteht aus dem Namen der Funktion und der Anzahl und Reihenfolge der zuweisungskompatiblen [Parameterdatentypen](http://de.wikipedia.org/wiki/Parameter_(Informatik)).

Beispiel:

double fun(double x, char\* nap) // Funktion

{

…

}

Signatur der obigen Funktion:

fun(double, char\*)

**Gültigkeitsbereich** – jede Variable gehört zu einem bestimmten Gültigkeitsbereich (engl. *scope*). Dieser legt fest, wann eine Variable von uns „gesehen“ und damit benutzt werden kann (Variablen und Konstanten haben nur begrenzte Lebensdauer). Vereinfacht gesagt bildet jedes Paar aus geschweiften Klammern ({}) einen eigenen Definitionsbereich. Dazu gehören beispielsweise *if*, *else*, Schleifen und Funktionen.

Es gibt verschiedene Arten von Gültigkeitsbereichen:

* *globaler Gültigkeitsbereich* – der Textbereich, der außerhalb jedes anderen Gültigkeitsbereich liegt;
* *Namensbereich* – ein explizit benannter Gültigkeitsbereich, der innerhalb des globalen Gültigkeitsbereichs oder eines anderen Namensbereich liegt;
* *Klassenbereich* – der Text innerhalb einer Klasse;
* *lokaler Gültigkeitsbereich* – Text zwischen den {}–Klammern eines Blocks oder in der Argumentliste einer Funktion;
* *Anweisungsbereich* – zum Beispiel innerhalb einer *for*–Anweisung.

**1.6.**

#include<cmath>  
#include<unittest++/UnitTest++.h>  
  
unsigned factorial (unsigned n)  
{  
 unsigned result = 1;  
  int i;  
  for (i = 2; i <= n; i++)  
  {  
    result \*= i;  
  }  
  return result;  
};  
  
SUITE(describe\_factorial)  
{  
  TEST(factorial\_of\_zero\_should\_be\_1)  
  {  
    CHECK\_EQUAL(1,factorial(0));  
  }  
  TEST(factorial\_of\_4\_should\_be\_24)  
  {  
    CHECK\_EQUAL(24,factorial(4));  
  }  
}  
  
int main()  
{  
return UnitTest :: RunAllTests ();  
}

**Testgetriebene Entwicklung (TDD)** – bezieht sich auf eine Methode bei der Softwareentwicklung, bei der folgende Aktivitäten eng miteinander verbunden sind: Kodieren, Testen und Design. Tests werden vorbereitet, bevor man mit dem richtigen Kodieren anfängt. Dann versucht man, diese Tests zu kompilieren. Die Kompilierung geht (offensichtlich) schief. Der zweite Schritt könnte man folgendermaßen beschreiben: „Man sollte jetzt mit dem Kodieren anfangen, man sollte aber möglichst wenig Kode schreiben, sodass nur die Tests erfolgreich laufen.“ Der letzte Schritt, dem man nicht unbedingt folgen muss, ist die „Korrektur“ des Kodes (und so der Tests), sodass er veranschaulich und funktonal ist.

**Unittest++** – die C++ Bibliothek, die TDD–Softwareentwicklung unterstützt.

**1.7.**

#include<cmath>

#include<math.h>  
#include<unittest++/UnitTest++.h>  
  
unsigned checksum (int n)  
{  
  unsigned sum = 0;  
  unsigned num = std::abs(n);  
  while (num > 0)  
  {  
    sum = sum + num%10;  
    num = num/10;  
  }  
  return sum;  
};  
  
SUITE(describe\_checksum)  
{  
  TEST(checksum\_of\_1256\_should\_be\_14)  
  {  
    CHECK\_EQUAL(14,checksum(1256));  
  }  
  TEST(checksum\_of\_401246\_should\_be\_17)  
  {  
    CHECK\_EQUAL(17,checksum(401246));  
  }  
  TEST(checksum\_of\_999\_should\_be\_27)  
  {  
    CHECK\_EQUAL(27,checksum(999));  
  }

  TEST(checksum\_of\_minus\_40\_should\_be\_4)  
  {  
    CHECK\_EQUAL(4,checksum(-40));  
  }  
}  
  
int main()  
{  
return UnitTest :: RunAllTests ();  
}

**1.8.**

#include<cmath>  
#include<math.h>  
#include<unittest++/UnitTest++.h>  
  
unsigned is\_prime (unsigned n)  
{  
  if (n == 2)  
  {  
    return 1;  
  }  
  else  
  {  
    float b = sqrt(n);  
    b = ceil(b);  
    int c=2;  
    while (c <= b)  
    {  
      if (n%c != 0)  
      {  
        c = c + 1;  
        if (c > b)  
        {  
          return 1;  
        }  
        else  
        {  
          return 0;  
        }  
      }  
    }   
  }  
};  
  
SUITE(describe\_is\_prime)  
{  
  TEST(is\_prime\_of\_13\_should\_be\_1)  
  {  
    CHECK\_EQUAL(1,is\_prime(13));  
  }  
  TEST(is\_prime\_of\_40\_should\_be\_0)  
  {  
    CHECK\_EQUAL(0,is\_prime(40));  
  }  
  TEST(is\_prime\_of\_3119\_should\_be\_1)  
  {  
    CHECK\_EQUAL(1,is\_prime(3119));  
  }  
}

// int main() befindet sich auf der 7. Seite

int main()  
{  
return UnitTest :: RunAllTests ();  
}

**1.9.**

#include<cmath>  
#include<math.h>  
#include<unittest++/UnitTest++.h>  
  
double fract (double n)  
{  
  double bet = fabs (n);  
  unsigned gan = floor (bet);  
  double gk = bet - gan;  
  return gk;  
};  
  
SUITE(describe\_fract)  
{  
  TEST(fract\_of\_1\_25632573238\_should\_be\_0\_25632)  
  {  
    CHECK\_CLOSE(0.25632,fract(1.25632573238),0.00001);  
  }  
  TEST(fract\_of\_40\_01246\_should\_be\_0\_01246)  
  {  
    CHECK\_CLOSE(0.01246,fract(40.01246),0.00001);  
  }  
  TEST(fract\_of\_minus\_999\_9999995\_should\_be\_0\_99999)  
  {  
    CHECK\_CLOSE(0.99999,fract(-999.9999995),0.00001);  
  }  
}  
  
int main()  
{  
return UnitTest :: RunAllTests ();  
}

**1.10.**

#include<cmath>  
#include<math.h>  
#include<unittest++/UnitTest++.h>

#define \_USE\_MATH\_DEFINES

double surface (unsigned r, unsigned h)  
{  
  float sur = 0;  
  float pi = M\_PI;  
  sur = 2\*pi\*r\*(r+h);  
  sur = round(sur\*100);        //Funktion *round(x)* rundet das Argument *x* auf eine ganze Zahl  
  sur = sur/100;                  
  return sur;  
};  
  
float volume (unsigned r, unsigned h)  
{  
  float vol = 0;  
  float pi = M\_PI;  
  vol = pi\*r\*r\*h;  
  vol = round(vol\*100);        //Funktion *round(x)* rundet das Argument *x* auf eine ganze Zahl  
  vol = vol/100;          
  return vol;  
};  
  
SUITE(describe\_surface)  
{

  TEST(surface\_of\_10\_and\_50\_should\_be\_3769\_91)  
  {  
    CHECK\_CLOSE(3769.91,surface(10,50),0.01);  
  }  
  TEST(surface\_of\_25\_and\_69\_should\_be\_14765\_49)  
  {  
    CHECK\_CLOSE(14765.49,surface(25,69),0.01);  
  }  
  TEST(surface\_of\_19\_and\_99\_should\_be\_14086\_90)  
  {  
    CHECK\_CLOSE(14086.90,surface(19,99),0.01);  
  }  
}  
  
SUITE(describe\_volume)  
{  
  TEST(volume\_of\_10\_and\_50\_should\_be\_15707\_96)  
  {  
    CHECK\_CLOSE(15707.96,volume(10,50),0.01);  
  }  
  TEST(volume\_of\_25\_and\_69\_should\_be\_135481\_18)  
  {  
    CHECK\_CLOSE(135481.18,volume(25,69),0.01);  
  }  
  TEST(volume\_of\_19\_and\_99\_should\_be\_112277\_38)  
  {  
    CHECK\_CLOSE(112277.38,volume(19,99),0.01);  
  }  
}

int main()  
{  
return UnitTest :: RunAllTests ();  
}

**1.11.**

#include<cmath>  
#include<math.h>  
#include<unittest++/UnitTest++.h>

#include<iostream>

#include<string>

float mileToKilometer (unsigned m)

{

float meile = 0;

meile = m\*1.609344;

meile = round(meile\*100); //Funktion *round(x)* rundet das Argument *x* auf eine ganze Zahl

meile = meile/100;

return meile;

};

SUITE(describe\_mileToKilometer)

{

TEST(mileToKilometer\_of\_10\_should\_be\_16\_09)

{

CHECK\_CLOSE(16.09,mileToKilometer(10),0.01);

}

TEST(mileToKilometer\_of\_450\_should\_be\_724\_20)

{

CHECK\_CLOSE(724.20,mileToKilometer(450),0.01);

}

TEST(mileToKilometer\_of\_250\_should\_be\_402\_34)

{

CHECK\_CLOSE(402.34,mileToKilometer(250),0.01);

}

}

int main()

{

UnitTest :: RunAllTests ();

unsigned mila;

mila = 0;

std::cout << "Bitte geben Sie Meilen ein:" << std::endl;

std::cin >> mila;

std::cout << mila << " Meilen entsprechen " << mileToKilometer(mila) << " Kilometern." << std::endl;

return 0;

}

**1.12.**

#include<cmath>  
#include<unittest++/UnitTest++.h>

#include<iostream>

#include<string>

#include<cctype>

int main()  
{  
  int zahl1, zahl2, zahl3, x, z, y;  
  std::cout << "Bitte geben Sie die erste Zahl ein:" << std::endl;  
  std::cin >> zahl1;  
  if (std::cin.fail())  
  {  
    std::cout << "Alle Argumente müssen ganze Zahlen sein!" << std::endl;  
    return 0;  
  }  
  std::cout << "Bitte geben Sie die zweite Zahl ein:" << std::endl;  
  std::cin >> zahl2;  
  if (std::cin.fail())  
  {  
    std::cout << "Alle Argumente müssen ganze Zahlen sein!" << std::endl;  
    return 0;  
  }  
  std::cout << "Bitte geben Sie die dritte Zahl ein:" << std::endl;  
  std::cin >> zahl3;  
  if (std::cin.fail())  
  {  
    std::cout << "Alle Argumente müssen ganze Zahlen sein!" << std::endl;  
    return 0;  
  }  
  if ((zahl1 > zahl2) and (zahl1 > zahl2))  
  {  
    if (zahl2 > zahl3)  
    {  
      x = zahl1;  
      y = zahl2;  
      z = zahl3;  
    }  
    else  
    {  
      x = zahl1;  
      y = zahl3;  
      z = zahl2;  
    }  
  }

if ((zahl2 > zahl1) and (zahl2 > zahl3))  
  {  
    if (zahl1 > zahl3)  
    {  
      x = zahl2;  
      y = zahl1;  
      z = zahl3;  
    }  
    else  
    {  
      x = zahl2;  
      y = zahl3;  
      z = zahl1;  
    }  
  }

if ((zahl3 > zahl1) and (zahl3 > zahl2))  
  {  
    if (zahl1 > zahl2)  
    {  
      x = zahl3;  
      y = zahl1;  
      z = zahl2;  
    }  
    else  
    {  
      x = zahl3;  
      y = zahl2;  
      z = zahl1;  
    }  
  }  
  
  std::cout << "Eingegebene Zahlen der Größe nach (absteigend):\n" << x << "\n" << y << "\n" << z << std::endl;  
  return 0;  
}

**1.13.**

**C++** - ist eine von der [ISO](http://de.wikipedia.org/wiki/Internationale_Organisation_f%C3%BCr_Normung) (Internationale Organisation für Normung) genormte [Programmiersprache](http://de.wikipedia.org/wiki/Programmiersprache). Sie wurde ab 1979 von [Bjarne Stroustrup](http://de.wikipedia.org/wiki/Bjarne_Stroustrup) als Erweiterung der Programmiersprache [C](http://de.wikipedia.org/wiki/C_(Programmiersprache)) entwickelt. C++ ermöglicht sowohl die [effiziente](http://de.wikipedia.org/wiki/Effizienz_(Informatik)) und maschinennahe Programmierung als auch eine Programmierung auf hohem Abstraktionsniveau.

**Quellcode** – Code, der von einem Programmierer entwickelt wurde und (im Prinzip) von anderen Programmierern gelesen werden kann.

**Compiler** – ein Programm, das Quellcode in Objektcode umwandelt.

**Linker** – ein Programm, das Objektcodedateien und Bibliotheken zu einem ausführbaren Programm zusammenbindet.

**Objektcode** – Ausgabe eines Compilers, die als Eingabe für den Linker dient (damit der Linker ausführbaren Code erzeugt).

**Ausführbare Datei** –eine kompilierte Datei, bei der das Betriebssystem „weiß“, wie es sie behandeln kann, ohne dass irgendwelche andere Softwarekomponenten erforderlich sind. Die entsprechenden Datei–Endungen sind bei den Betriebssystemen verschieden.

**main()** – Funktion, die den Start jedes Programmes bezeichnet (das Hauptprogramm ist in C++ eine Funktion mit dem Namen *main*). Die *main*–Funktion darf pro Programm nur einmal definiert werden und sie muss sich im globalen Bereich befinden.

**#include** – Direktive, die den Computer anweist, die Hilfsmittel aus einer Datei/Bibliothek verfügbar zu machen.

**Kommentar** – zusätzliche Information, die sich nicht gut durch Code ausdrücken lässt. Kommentare sind ein wichtiges Mittel, um den Zweck des Codes zu dokumentieren.

**Header** – eine Datei, die Deklarationen enthält und dazu dient, den verschiedenen Teilen eines Programms Schnittstellen zur gemeinsamen Nutzung zur Verfügung zu stellen.

**Programm** – Code (unter Umständen verbunden mit Daten), der ausreichend vollständig ist, um von einem Computer ausgeführt zu werden.

**Ausgabe** – Werte, die durch eine Berechnung erzeugt werden (z. B. ein Funktionsergebnis oder eine Zeile mit Zeichen, die auf dem Bildschirm ausgegeben werden).

**std::cout** – ein Objekt des Typs *ostream*. Der Name *cout* bezieht sich auf den Standardausgabestream, der in der Bibliothek *iostream* zur Verfügung gestellt wird.

**std::cin** – ein Objekt des Typs *istream*. Der Name *cin* bezieht sich auf den Standardeingabestream, der in der Bibliothek *iostream* zur Verfügung gestellt wird.

**<<** – Ausgabeoperator, mittels dessen Objekte verschiedener Typen auf einem Ausgabe–Datenstrom ausgegeben werden können.

**>>** – Eingabeoperator, der dem Einlesen von Objekten verschiedener Typen von einem Eingabe–Datenstrom dient.

**Funktion** – eine benannte Codeeinheit, die aus verschiedenen Teilen eines Programms aufgerufen werden kann; eine logische Berechnungseinheit.

**Funktionssignatur** – die formale Schnittstelle einer [Funktion](http://de.wikipedia.org/wiki/Funktion_(Programmierung)). Sie besteht aus dem Namen der Funktion und der Anzahl und Reihenfolge der zuweisungskompatiblen [Parameterdatentypen](http://de.wikipedia.org/wiki/Parameter_(Informatik)).

**Deklaration** – Programmanweisung, die angibt, wie ein Codeelement verwendet werden kann.

**Definition** – Anweisung, die einen neuen Namen in einem Programm einführt und Speicher für eine Variable reserviert.

**Typ** –  gibt an, von welcher Art die [Daten](http://wirtschaftslexikon.gabler.de/Definition/daten.html) sind, die mit ihm beschrieben werden, und welche Operationen auf diesen ausgeführt werden können.

**Typkonvertierung** – mittels Typkonvertierung kann der ursprüngliche Datentyp, z.B. einer Variable, einer Konstante oder eines Ausdrucks, in einen anderen Datentyp konvertiert werden.

**Variable** – ein [Datenelement](http://wirtschaftslexikon.gabler.de/Definition/datenelement.html) oder eine [Datenstruktur](http://wirtschaftslexikon.gabler.de/Definition/datenstruktur.html), die bei der Ausführung des Programms verschiedene Werte annehmen kann. Eine Variable besitzt einen [Datentyp](http://wirtschaftslexikon.gabler.de/Definition/datentyp.html).

**Name** – Bezeichner eines Objektes. Ein Name beginnt mit einem Buchstaben und enthält nur Buchstaben, Ziffern und Unterstriche.

**Wert** – der Inhalt, der in einer Variablen gespeichert ist. Der Wert einer Variablen kann jederzeit geändert werden.

**Initialisierung** – eine Variable erhält ihren Anfangswert.

**Zuweisung** – in eine Variable wird ein neuer Wert geschrieben.

**Konstante (*const)*** – repräsentiert einen Wert, der nicht verändert werden kann. Eine Konstante ist entweder eine ganzzahlige Konstante, eine Gleitpunktkonstante, eine Zeichenkonstante oder eine Stringkonstante.

**Gültigkeitsbereich** – jede Variable gehört zu einem bestimmten Gültigkeitsbereich (engl. *scope*). Dieser legt fest, wann eine Variable von uns „gesehen“ und damit benutzt werden kann (Variablen und Konstanten haben nur begrenzte Lebensdauer). Vereinfacht gesagt bildet jedes Paar aus geschweiften Klammern ({}) einen eigenen Definitionsbereich. Dazu gehören beispielsweise *if*, *else*, Schleifen und Funktionen.