BBS

Herzlich willkommen!

Dozent: Dipl.-Ing. Jürgen Wemheuer

Mail: wemheuer@ewla.de

Online: http://cpp.ewla.de/

Wer hoch hinaus will, muss lange beim Fundament verweilen...

Teil 2:

- Gestaltungs-Grundregeln
- Variablen und Konstanten
- Zuweisungen
- Von der Aufgabe zur Lösung
- Programmablaufpläne
- Struktogramme

Gestaltung: wie ein Kochrezept

BBS

Ein Programm ist wie ein Kochrezept:

1. Beschreibung der Daten:
   – Deklarationen
     (Beschreibung der Zutaten, mit denen gearbeitet wird, „man nehme...“)
2. Beschreibung des Ablaufs:
   – Anweisungen
     (Befehle, die Schritt für Schritt nacheinander ausgeführt werden, „Zubereitung“)

Gestaltungs-Grundregeln (1)

1. **Die Zeile:**
   Sie ist die Grundstruktur eines C++-Programms.
   Wenn sie eine Anweisung oder Vereinbarung beinhaltet,
   muss sie mit einem Semikolon ; abgeschlossen werden.
2. **Präprozessor-Vereinbarung:**
   Beginnt im C++-Programm mit einem # und bewirkt,
   dass der vom Compiler aufgerufene Präprozessor
   die so genannten "Header-Dateien" hinzufügt.
   Die Zeile endet nicht mit einem Semikolon ;
3. **Die main()-Funktion:**
   Das ist die Hauptfunktion des C++-Programms.
   Sie wird vom Betriebssystem des Computers aufgerufen.

Gestaltungs-Grundregeln (2)

4. **Schreibweise:**
   Alle vorkommenden Variablen und Funktionen müssen einer definierten Schreibweise genügen, z.B. Groß- und Kleinschreibung. Die einmal getroffene Entscheidung über die Schreibweise darf im Programm an keiner Stelle verändert werden.
5. **Variablen-Vereinbarung:**
   Definition aller im C++-Programm verwendeten "lokalen" und "globalen" Programmvariablen in festen Variablentypen.
6. **Belegtes/gesperrtes Wort:**
   Bestimmte Worte, sogenannte Schlüsselworte, der Sprache C++ sind fest vergeben (reserviert) und dürfen nicht für projekteigene Variablen benutzt werden, z.B. die Worte do, delete, if, int, while usw.

Gestaltungs-Grundregeln (3)

7. **Kommentar:**
   Ist für den Programmablauf ohne jede Bedeutung und wird deshalb nicht vom Compiler übersetzt.
   Zwei Möglichkeiten gibt es:
   - In der Zeile kommt an beliebiger Stelle **//** und der Kommentar folgt bis zum Zeilenende
   - In der Zeile kommt an beliebiger Stelle **/*** und alles ist Kommentar bis zum nächsten ***/**

8. **Auskommentieren:**
   Elegante Möglichkeit der dynamischen Programmkorrektur ohne Löschung des heraus zu nehmenden Teils.
   (Obacht beim Auskommentieren von Kommentaren...!)

Hello World in C++

```
#include <iostream>
using namespace std;
int main()
{
   cout << "Hello World" << endl;
   system("pause");
}
```

Einbinden der *Bibliothek* iostream,
die **`cout`** und **`endl`** enthält.

Hello World in C++


```
#include <iostream>
using namespace std;
int main()
{
  cout << ”Hello World” << endl;
  system(”pause”);
}
```

Festlegen des *Namensraums* **`std`**, damit automatisch die richtigen Elemente der Bibliothek **`iostream`** angesprochen werden.

(Alternative z.B.: `std::cout << ”Hello World” << std::endl;`)

Hello World in C++

```
#include <iostream>
using namespace std;
int main()
{
  cout << ”Hello World” << endl;
  system(”pause”);
}
```

- Deklaration der Funktion `main`
- Keine Parameter nötig (aber möglich)
- Rückgabetyp `int`

Hello World in C++


```
#include <iostream>
using namespace std;
int main()
{
  cout << ”Hello World” << endl;
  system(”pause”);
}
```

- Beginn und Ende des Funktionsrumpfes von `main`

Hello World in C++


```
#include <iostream>
using namespace std;
int main()
{
   cout << "Hello World" << endl;
   system("pause");
}
```

- Ausgabe von “Hello World“ auf dem Stream `cout`
- Danach Zeilenumbruch (`endl` : End Of Line)
- Danach auf Bedienerreaktion warten

Variablen und Konstanten

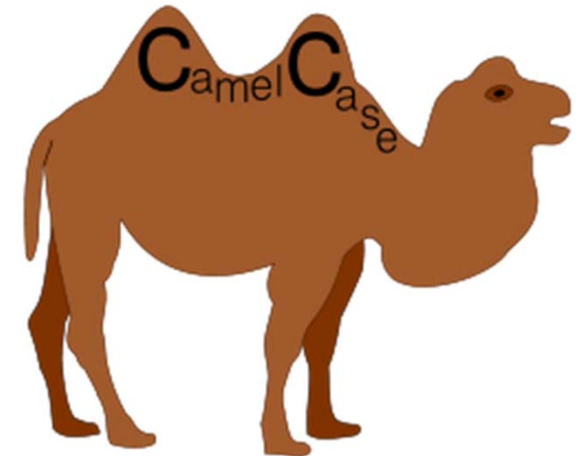
- Variable:
  Eine Variable ist eine veränderliche Größe mit einem fest vereinbarten Namen, der ein fester Platz im Hauptspeicher des Computers zugewiesen wird. Der Wert der Variablen kann sich während der Laufzeit des Programms verändern. Eine Variable muss VOR der ersten Verwendung einmalig DEKLARIERT (als Datentyp „angemeldet“) werden!
- Konstante:
  Eine Konstante ist gleich einer Variablen, nur dass sich ihr Wert nicht ändert, d.h. er bleibt fest (konstant) über die gesamte Laufzeit des Programms. Dazu <u>muss</u> eine Konstante bei ihrer Erzeugung initialisiert werden.

Variablen haben:

- einen eindeutigen Namen (Bezeichner):
  - Vom Benutzer wählbar, durch Deklaration festgelegt
    - Regeln für Namen beachten (nächste Folie)!
    - Sinnvolle Variablennamen sind wesentliches Element der Lesbarkeit!!!
- einen Typ (int, double, const char *, ...):
  - Durch Deklaration festgelegt
(bestimmt Art, Interpretation der Werte und den Platzbedarf)
- einen (aktuellen) Wert:
  - Durch Initialisierung oder Zuweisung (=)
(und durch Parameter-Übergabe, siehe später)
- einen Speicherplatz (Adresse & Größe):
  - Vom Compiler zugewiesen, siehe „Zeiger“...
- einen „Scope“ („Sichtbarkeit“, Lebensdauer):
  - Siehe später...

Variablennamen (in C/C++)

- Buchstaben und Ziffern, immer beginnend mit Buchstabe
- Groß- und Kleinschreibung wird unterschieden!
- Unterstrich _ zählt als Buchstabe und ist überall erlaubt
- sonst keine Sonderzeichen!
- Keine Umlaute und ß, keine Akzente
- Keine Schlüsselworte, __... ist auch reserviert



Üblich:

- Entweder “CamelCase” (Anfangsbuchstaben der Teilworte groß)
- oder Teilworte mit _ getrennt
- “Nur Großbuchstaben” ist üblich für #define-Namen, sonst nie!

Datum: Datentyp und Werte

Datentypen bestimmen:

- welchen Speicherplatz (Größe, Anzahl Bytes) ein Datum physikalisch (im Arbeitsspeicher, auf einem Speichermedium) beansprucht
- wie ein Wert intern codiert wird:
  4 als Ganzzahl: 0000 0000 0000 0100 (2 Byte Short Integer)
  4 als ASCII-Zeichen: 0011 0100 (1 Byte Character)
  4 als Dezimalzahl: 0100 0000 1000 0000 0000 0000 0000 0010
- wie ein Wert zu interpretieren ist und verarbeitet werden kann: Zahl 5 multipliziert mit Zeichen ‚Y' ist unsinnig…
- den Wertebereich, den ein Datum annehmen kann:
  (z.B. unsigned short int für Ganzzahlen von 0…65535)

Variablen: Basis-Datentypen C

Typ	Größe	Wertebereich	
(short) int	2 Byte	-32,768	+32,767
unsigned (short) int	2 Byte	0	+65,535
long (int)	4 Byte	- 2,147,483,648	+2,147,483,647
unsigned long (int)	4 Byte	0	+4,294,967,295
char	1 Byte	256 Zeichenwerte (erweiterter ASCII)	
bool	1 Byte	1-255: true (wahr)	0: false (falsch)
float	4 Byte	1.2 e-38	3.4 e+38
double	8 Byte	2.2 e-308	1.8 e+308

Alle nummerischen Datentypen mit den nach ANSII-C++ garantierten Mindestwertbereichen -> je nach System unterschiedlich! Wertbereichsgrenzen sind MODULO!

Variablen: Ganzzahlentypen int

Datentyp	Größe	Wertebereich	
signed char	1 Byte	-128	+127
unsigned char	1 Byte	0	+255
signed short int	2 Byte	$- 2^{15}$	$+ 2^{15} -1$
unsigned short int	2 Byte	0	$+ 2^{16} -1$
signed int	4 Byte	$- 2^{31}$	$+ 2^{31} -1$
unsigned int	4 Byte	0	$+ 2^{32} -1$
signed long int	4 Byte	$- 2^{31}$	$+ 2^{31} -1$
unsigned long int	4 Byte	0	$+ 2^{32} -1$
signed long long int	8 Byte	$- 2^{63}$	$+ 2^{63} -1$
unsigned long long int	8 Byte	0	$+ 2^{64} -1$

Wertebereich der Ganzahltypen int

- ACHTUNG: Die Größe der Integer-Datentypen ist implementationsabhängig!
- Auf jeden Fall gilt:
  - short int <= int <= long int
  - long long int hat mindestens 64 bit (8 Byte)
- Im allgemeinen ist der int-Datentyp auf
  - 16-bit-Compilern mit **`short int`** und auf
  - 32-bit-Compilern mit **`long int`** identisch!

Wertebereich für Gleitkommazahlen

Datentyp	Bit (Mantisse/Exponent)	Wertebereich	Dezimalstellen
float	32 (23 / 8)	$3.4*10^{-38}$ ... $3.4*10^{38}$	7
double	64 (52 / 11)	$1.7*10^{-308}$ ... $1.7*10^{308}$	15
long double	80 (64 / 15)	$1.2*10^{-4932}$ ... $1.2*10^{4932}$	19

Variablen - Wertzuweisungen


```
01 // Einsatz von Variablen am Beispiel Flaechenberechnung
02 #include <iostream>
03 using namespace std;                          // Standardzuweisung
04
05 int main()
06 {
07       unsigned short Breite, Laenge;
08       unsigned short Flaeche;
09       Breite = 5;
10       Laenge = 10;
11       Flaeche = Breite * Laenge;
12       getchar();                          // Anzeige alpha-numerische
13       getchar();                          // Oberflaeche (zwei Zeichen)
14       return 0;
15 }
```

ODER AUCH:

```
...      unsigned short Breite = 5, Laenge;   // Doppeldefinition
...      unsigned short Flaeche = Breite * Laenge;
```

Variablen - typedef-Zuweisungen


```
01 // typedef = define type = Typendefinition => Schluesselwort
02 #include <iostream>
03 using namespace std;                    // Standardzuweisung
04 typedef unsigned short int iVoV; // definiert iVoV
05 int main()
06 {
07       iVoV Breite = 5;
08       iVoV Laenge;
09       Laenge = 10;
10       iVoV Flaeche = Breite * Laenge;
11       getchar();                       // Anzeige alpha-numerische
12       getchar();                       // Oberflaeche (zwei Zeichen)
13       return 0;
14 }
```

(weiter: eigene Datentypen / Datentypumwandlung)

Und WO sind unsere Daten?

- Jedes Datum ist an einer physikalischen Adresse im RAM gespeichert.
- Auf den **WERT** können wir bequem zugreifen, indem wir im Compiler einen **NAMEN** dafür vergeben.
- Die **ADRESSE** können wir erfahren, wenn wir dem Variablennamen ein **&** voranstellen (Adressoperator, Kaufmanns-UND, Ampersand …)
- Beispiel:
  int xyz=78; cout << &xyz; // -> 0x123456

Zeigervariablen

Um mit diesen Adressen auch rechnen zu können, merken wir uns die Adresse der Daten in eigenen Zeigervariablen (kurz: in Zeigern / Pointern):

- Diese Zeiger beinhalten Adressen – keine Werte!
- und sollten deshalb auffällig anders benannt werden, zum Beispiel:
  - ptrMeineVariable
  - p_Var
  - ZeigerAufEtwas

Zeigervariablen: Deklaration

Ein Variablenname muss wissen, ob er für einen Wert oder eine Adresse steht…

Und ein Zeiger muss wissen, wohin (auf welche Variable) und worauf (auf welchen Datentyp) er zeigt:

Dafür notieren wir den Zeigeroperator *:

- `int *Zeiger_auf_Ganzzahl; // C-Stil klassisch`
- `int* Zeiger_auf_Ganzzahl; // C++-Stil`
- `int * Zeiger_auf_Ganzzahl; // C as you like`

Bei Mehrfachzuweisungen problematisch -> besser für jede Zeigerdeklaration eine eigene Zeile spendieren…

Zeiger: Regeln

Nicht initialisierte Zeiger zeigen IRGENDWOHIN

- Abhilfe evtl.: `int *MeinZeiger = Null;`
- oder noch besser gleich initialisieren:
  `int* ptrAufVar1 = &Var1;`
- Auf Zeiger können alle arithmetischen und Vergleichs-Operationen angewendet werden:

  +, -, ++, --, *, /, =, ==, <, >, <=, >=, !=

- **Bei jeder Rechenoperation ändert sich die in der Zeigervariablen gespeicherten Adresse gemäß dem dort hinterlegten Datentyp um die von diesem Datentyp beanspruchten Bytes!**

Zeiger: Datenfelder (Arrays)

Bei Datenfeldern entspricht der Name des Datenfeldes der Adresse:

```
float Fliesskomma[2] = {12.3456,98.76543};
float * ZeigerAufFloatFeld = Fliesskomma;
float * ZeigerAufFloatFeld = &Fliesskomma[0];
```

Achtung: Bei Datenfeldern des Typs char wird bei der Ausgabe mit cout nicht die Adresse des ersten Datenfeldes angezeigt, sondern das char-Datenfeld als Zeichenkette:

```
char Text[] =„Mein Text“; char *ptrText = Text;
cout << ptrText; // Anzeige: Mein Text (statt 0x123456)
cout << (int*)ptrText; // Abhilfe: Typumwandlung…
```

Zeiger: Wert anzeigen

Zeigt ein Zeiger auf einen Wert im Speicher, so kann dieser Wert mit Hilfe des „Indirektionsoperators“ („Verweisoperator“) ausgelesen werden:

```
int variable = 5;
int* zeiger = &variable;
cout << zeiger; // Adresse:0x123456
cout << *zeiger; // Wert:5
```

Zeiger: Werte in einem Array

Als Zugriff auf die Array-Elemente haben wir bisher den Operator [] benutzt:

```
int Array[9] = {1,2,3,4,5,6,7,8,9};
cout << Array[4]; // Ausgabe: 5
```

Da der Array-Name der Zeiger auf Element 0 ist, wäre genauso gut möglich:

```
int Array[9] = {1,2,3,4,5,6,7,8,9};
cout << *(Array+4); // Ausgabe: 5
```

Allgemein: a[n] entspricht *(a+n)

Zeiger auf Strukturen

Zeigt ein Zeiger auf eine Struktur, so kann man auf deren Werte mit dem Pfeil-Operator **->** zugreifen (auch der Verweisoperator * ist nicht mehr erforderlich):

```
struct Person{
   string Name;
   int Alter;
};
Person Hans;
Person* P = &Hans;
P->Name = „Hänschen Klein“;
P->Alter = 29;
```

String(-Zeiger) in C und C++ (1)


```
char name1[] = „Anna“;
char name2[] = „Bernd“;
name1 = name2; // Was passiert hier?
```

In C zeigt `„name1“` nun ebenfalls auf „Bernd“.

In C++ ist die Zuweisung von Array-Pointern nicht erlaubt und gibt einen Compiler-Fehler…

```
string name1 = „Anna“;
string name2 = „Bernd“;
name1 = name2; // Was passiert hier?
```

In C++ erhält `name1` nun eine Kopie von `name2`!

String(-Zeiger) in C und C++ (2)

In C deshalb benötigt:
Funktionen der <string>-Bibliothek:

```
char *strcpy(s,ct);
char *strcat(s,ct);
int strcmp(cs,ct);
char *strchr(cs,s);
char *strstr(cs,ct);
size_t strlen(cs);
```

String(-Zeiger) in C und C++ (3)

In C++ kann ein String wie andere Datentypen behandelt werden:

```
string name1 = „Anna“;
string name, name2;
name2 = name1;
name = name1 + name2; // Verkettung
… name1 == name2 …
… name1 != name2 …
… name1 > name2 …
… name1 < name2 …
```

Zeiger: zulässig… (1)

Man kann einem Zeiger auch direkt einen Speicherplatz zuweisen:

```
int* zeiger;
zeiger = (int*) 1000; // Adresse 1000
```

(Hier wird ein „cast“ benötigt, da 1000 eine Ganzzahl (Typ int) ist und erst in einen „int-Pointer“ umgewandelt werden muss…)

Das geht allerdings nicht, wenn das Betriebssystem Schutz-mechanismen hat, die den Zugriff auf nicht freigegebenen / angeforderten Speicher verwehren. Der Compiler kompiliert zwar, doch hier käme es dann zu einem Laufzeitfehler…

Zeiger: zulässig… (2)

Man kann auch Zeigertypen casten:

```
float f = 5.0f;
float* zeiger = &f;// Adresse von f
cout << *zeiger;    // Indirekter Zugriff
/* Aber jetzt kommts…: */
char* z;
z = (char*) zeiger;   // Zeiger auf Bytes
cout << (int) *z;     // Ausgabe als int
cout << (int) *(z+1);// Ausgabe als int
```

Dynamische Speicherverwaltung

Mithilfe der Zeiger können wir „dynamisch“ (= zur Laufzeit des Programms) neuen Speicher vom Betriebssystem anfordern und freigeben:

```
int Anzahl = 200;
int* DynArray;
DynArray = new int[Anzahl];
...
delete[] DynArray;
```

Obacht: Der Zeiger DynArray zeigt jetzt auf keinen sinnvoll nutzbaren Speicher mehr, denn das Array ist jetzt ja „weg“…

Dynamische Speicherverwaltung

new/delete geht auch mit elementaren Variablen, aber wozu?

new/delete geht aber auch mit eigenen Datentypen (enum, struct) und Klassen (z.B. string).

Strukturen müssen wieder elementweise initialisiert werden:

```
Person* P = new Person;
P->Name = "Hannes Maier";
P->Alter = 19;
cout << P->Name << " " << P->Alter;
```

Diese Struktur lässt sich **nur** über den Zeiger und den Pfeil-Operator ansprechen, denn einen Namen gibt es dafür nicht…

Typenumwandlungen

- Verschiedene Datentypen können miteinander kombiniert und ineinander umgewandelt werden
- Das geschieht entweder automatisch (implizit) oder gezielt beim Codieren (explizit)

OBACHT: Stolperfalle...

Implizite Typenumwandlungen

VON	(Werte)	NACH	(Werte)
bool	(false, true)	int	(0, 1)
int	(0, n)	bool	(false, true)
char	(‘A‘-‘Z‘, ‘a‘-‘z‘,...)	int	(ASCII-Wert)
int	(ASCII-Wert)	char	(entspr. Zeichen)
int	(n)	float oder double	(n.0)
float oder double	(n.m)	int	(n, Abschneiden der Dezimalstellen)

Achtung:
Ganzzahl- oder Gleitkomma-Arithmetik?
Beispiel:

```
float a = 2 / 3;             // a = 0.0
float b = 2.0 / 3.0;         // b = 0.666667
float c = (2 / 3) * 2.5;     // c = 0.0 !!
```

Implizite Typenumwandlungen

```
#include <iostream>
using namespace std;
// Beispiel für implizite (automatische) Typenumwandlung
int main() {
   char x=65, y='B';
   int i;

   cout << "x = " << x << ", y = " << y << "\n";
   i = y;
   cout << "Summe = " << (x+y) << ", i = " << i << endl;
   cout << "x = " << x << ", y = " << y << "\n";

   return 0;
}
```

Implizite Typenumwandlung

Aha, so etwas gibt es auch:

```
bool Wahrheit = false;
Wahrheit++;
// Wahrheit ist jetzt true
Wahrheit--;
// Wahrheit ist wieder false
```

Explizite Typenumwandlungen

In C wird der Typ eines Ausdrucks explizit gewandelt (cast), indem man den gewünschten Typ in Klammern davor schreibt:

```
(long)i    // wandelt den Wert von i nach long
```

In C++ kann man alternativ auch den Typ vor einen Klammer-Ausdruck schreiben, der daraufhin konvertiert wird:

```
long(i)    // funktionale Schreibweise
```

Die C++-Version ist manchmal leichter zu lesen: die Klammern begrenzen deutlich den Ausdruck, der gewandelt wird.

Die neue Schreibweise ist nur möglich, wenn der zu erzielende Typ aus einem Wort besteht. Ansonsten muss man ihn wieder klammern und kommt damit zur alten Schreibweise zurück.

```
(long)(i) // für Unentschlossene…
```

Explizite Typenumwandlungen

Beispiele:

```
int i = (int)'A'; // i=65
int i = int('A'); // i=65
```

Klammerregeln beachten!

```
double dx = (double)(1/2); // dx=0
double dx = double(1)/2; // dx=0.5
```

Explizite Typenumwandlungen

Obacht: Informationsverlust / „Schrott“ möglich:

```
double dx;
float fx;
dx = (double) 1.23456789012345678 9;
fx = float (dx);
// dx = 1.23456789012345669000
// fx = 1.23456788063049316400

int ix = -100;
unsigned ux = (unsigned) ix;
// ux = 4294967196
```

Explizite Typenumwandlungen

- int i = 3; up((double)i);
  konvertiert eine lokale Kopie von i (ein temporäres Objekt) aktiv in den Typ double. Das ist tatsächlich eine sinnvolle Konversion: der an up() übergebene Wert ist 3.0
- int i=3; *(double*)&i = 3.0 dagegen konvertiert nichts!
  Vielmehr wird &i, also die Adresse von i, als Zeiger auf ein double betrachtet ((double*)&i). Dann wird dahin, worauf diese Adresse zeigt (an die Stelle *(double*)&i im Speicher) ein double-Wert (3.0) geschrieben. Auf üblichen Rechnern ist ein int 4 Byte groß, ein double dagegen 8 oder mehr Byte. Es wird also über die 4 Byte von i hinaus weiterer dahinter liegender Speicher mit dem Bitmuster der Gleitkommazahl 3.0 überschrieben; in i sowie in dem nicht mehr dazu gehörigen, dahinter liegenden Speicher steht Schrott...

Explizite Typenumwandlungen OOP

Man kann auch eigene Datentypen, z.B. Objekte casten...

- dynamic_cast<T>
  konvertiert einen Zeiger auf ein Objekt einer Basisklasse in einen Zeiger auf ein Objekt einer davon abgeleiteten Klasse
- static_cast<T>
  Konvertierung im Sinne des ursprünglichen cast, entspricht also (T)wert
- reinterpret_cast<T>
  das Bitmuster eines gegebenen Ausdrucks wird einfach als ein anderer Typ angesehen. Der Compiler führt tatsächlich keine Konversion durch!
- const_cast<T>()
  von einem const-Objekt wird das const-Attribut *abgestreift!*
- typeid()
  liefert Informationen zum aktuellen Datentyp

Konstanten

Es gibt bei C++ zwei Arten: literale und symbolische Konstante

1. Literale Konstante (Literal)
   - Der Wert wird „hardcoded“ an der Stelle des Vorkommens gesetzt
   - Dieser Wert kann natürlich zur Laufzeit nicht mehr geändert werden

```
Raeder = Fahrzeuge * 4;   // Anzahl der Raeder
```

2. Symbolische Konstante
   - Sie wird genau wie eine Variable durch einen Namen repräsentiert.
   - Der Wert der Konstanten kann aber nicht geändert werden

```
#define RaederProFahrzeug 4    // ODER:
const unsigned short int RaederProFahrzeug = 4;
/* kontrollierte Zuweisung des Datentyps */

Raeder = Fahrzeuge * RaederProFahrzeug
```

Aufzählungskonstante

Aufzählungskontanten (enum) erzeugen einen Satz von Konstanten mit einem festen Bereich von Werten:

```
enum FAHRZEUG {Pkw, Lkw, Bike, Traktor, Roller};
```

1. Erzeugung eines (neuen) Typs

- Erzeugung => enum des Datentyps => FAHRZEUG

2. Symbolische Konstante

- Symbolische Konstante Pkw erhält den Wert 0, Lkw wird 1 usw.

Die Wertezuweisung kann auch definitiv erfolgen:

```
enum FAHRZEUG {Pkw=100, Lkw, Bike=500, Traktor, Roller};
```

-> Pkw=100, Lkw=101, Bike=500, Traktor=501, Roller=502

Aufzählungskonstante

Intern werden enum-Konstanten wie Ganzzahlen behandelt.

So könnte man statt:

```
enum FAHRZEUG {Pkw, Lkw, Bike, Traktor, Roller};
```

auch schreiben:

```
const int Pkw = 0;
const int Lkw = 1;
const int Bike = 2;
const int Traktor = 3;
const int Roller = 4;
```

enum-Konstanten sind jedoch typsicher mit einem definierten Wertebereich!

Andere Werte als Pkw, Lkw, Bike, Traktor, Roller kann es nicht geben!

Aufzählungskonstante

Beispiel:

```
enum FAHRZEUG {Pkw, Lkw, Bike, Traktor, Roller};

FAHRZEUG MeinFahrzeug = Pkw;
FAHRZEUG DeinFahrzeug = Roller;
if (MeinFahrzeug == DeinFahrzeug)
    cout << "Wir haben das gleiche Fahrzeug!";
MeinFahrzeug = 4;        // nicht zulässig (?)
DeinFahrzeug = Cabrio;   // nicht zulässig (logisch!)
```

Aufzählungskonstante

Beispiel:

```
enum FAHRZEUG {Pkw, Lkw, Bike, Traktor, Roller};

FAHRZEUG MeinFahrzeug = Lkw;
cout << MeinFahrzeug;        // Anzeige: 1
switch (MeinFahrzeug) {
    case Pkw:     cout << "Pkw"; break;
    case Lkw:     cout << "Lkw"; break;
    case Bike:    cout << "Bike"; break;
    case Traktor: cout << "Traktor"; break;
    case Roller:  cout << „Roller"; break;
    default:;     // kann es eigentlich nicht geben…!
}
```

Datenfelder (Arrays) 1

BBS

1. Definition (1)

- Ein Feld (engl. Array) ist eine definierte Zusammenfassung von Speicherplätzen für Daten beliebigen Typs
- Aber die Daten der Speicherplätze eines Felds sind alle vom gleichen Typ
- Ein Datum des Feldes bekommt die entsprechende Menge Speicherplatz zugewiesen und bezeichnet damit ein Element des Feldes
- Das Deklarieren eines Arrays erfolgt durch Benennung des Datentyps, des Feldnamens und des Indizes, der in eckige Klammern gesetzt wird

Datenfelder (Arrays) Beispiel

```
float TauchTiefen [4]; // 4 Fließpunktzahlen für Tauchtiefen
```

Speicherbild:

XX0	1. Byte	Anfang der 1. Fließpunktzahl, Feldanfang, TauchTiefen[0]
XX1	2. Byte	
XX2	3. Byte	
XX3	4. Byte	Ende der 1. Fließpunktzahl
XX4	5. Byte	Anfang der 2. Fließpunktzahl = TauchTiefen[1]
XX5	6. Byte	
XX6	7. Byte	
XX7	8. Byte	Ende der 2. Fließpunktzahl
XX8	9. Byte	Anfang der 3. Fließpunktzahl = TauchTiefen[2]
XX9	11. Byte	
X10	12. Byte	
X11	13. Byte	Ende der 3. Fließpunktzahl
X12	14. Byte	Anfang der 4. Fließpunktzahl = TauchTiefen[3]
X13	15. Byte	
X14	16. Byte	
X15	17. Byte	Ende der 4. Fließpunktzahl = Feldende

Datenfelder (Arrays) 2

Die Zählung der Elemente eines Feldes beginnt mit NULL!!!

- Das erste Element eines Feldes bekommt also den Index 0
- Das letzte Element eines Feldes hat den Index Feldgröße - 1
- Auf Bereichsgrenze muss man aber selbst achten!!!

Das Initialisieren von einfachen Feldern kann nur mit vordefinierten Basis-Datentypen (wie int und char) erfolgen:

- Beispiel: int DekaFeld [5] = {10, 20, 30, 40, 50};
- oder so: int DekaFeld [] = {10, 20, 30, 40, 50};

Die Ermittlung einer Feldgröße durch den Compiler ist möglich:

const int DekaFeldDim = sizeof(DekaFeld)/sizeof(DekaFeld[0]);

5 10 / 2 !

Mehrdimensionale Datenfelder 1

- Die Definition von mehr-dimensionalen Feldern ist denkbar und möglich
- Das Anlegen von mehr-dimensionalen Feldern ist „nach oben" theoretisch unbeschränkt möglich
- Praktisch von Bedeutung sind ein-, zwei- und manchmal noch dreidimensionale Felder: *Vektoren und Matrizen:*
  - Der Vektor, das eindimensionale Feld: long Array [n];
  - Die Matrix, das zweidimensionale Feld: long Array [n][m];
  - Die Matrix, das mehr-dimensionale Feld:
    long Array [n][m][p][q]...[z];

Mehrdimensionale Datenfelder 2

BBS

Beispiel mit 4 Zeilen und 2 Spalten:

```
int Array [4][2]
        = { {1,2}, {2,4}, {4,8}, {8,16} };
```

Mit:

```
Array [0][0] = 1, Array [0][1] = 2,
Array [1][0] = 2, Array [1][1] = 4,
Array [2][0] = 4, Array [2][1] = 8,
Array [3][0] = 8, Array [3][1] = 16
```

entspricht im Speicherabbild:

```
int Array[8]={ 1, 2, 2, 4, 4, 8, 8, 16};
```

Anwendungsbeispiel (1)

Die Bestimmung der Höhe der Bäume in einem Park erfolgt nach der Methode des Strahlensatzes. Dafür werden von verschiedenen Personen Messungen vorgenommen. Aus den vorliegenden Messergebnissen kann man mit Hilfe eines Computerprogramms die Höhen aller Bäume errechnen.

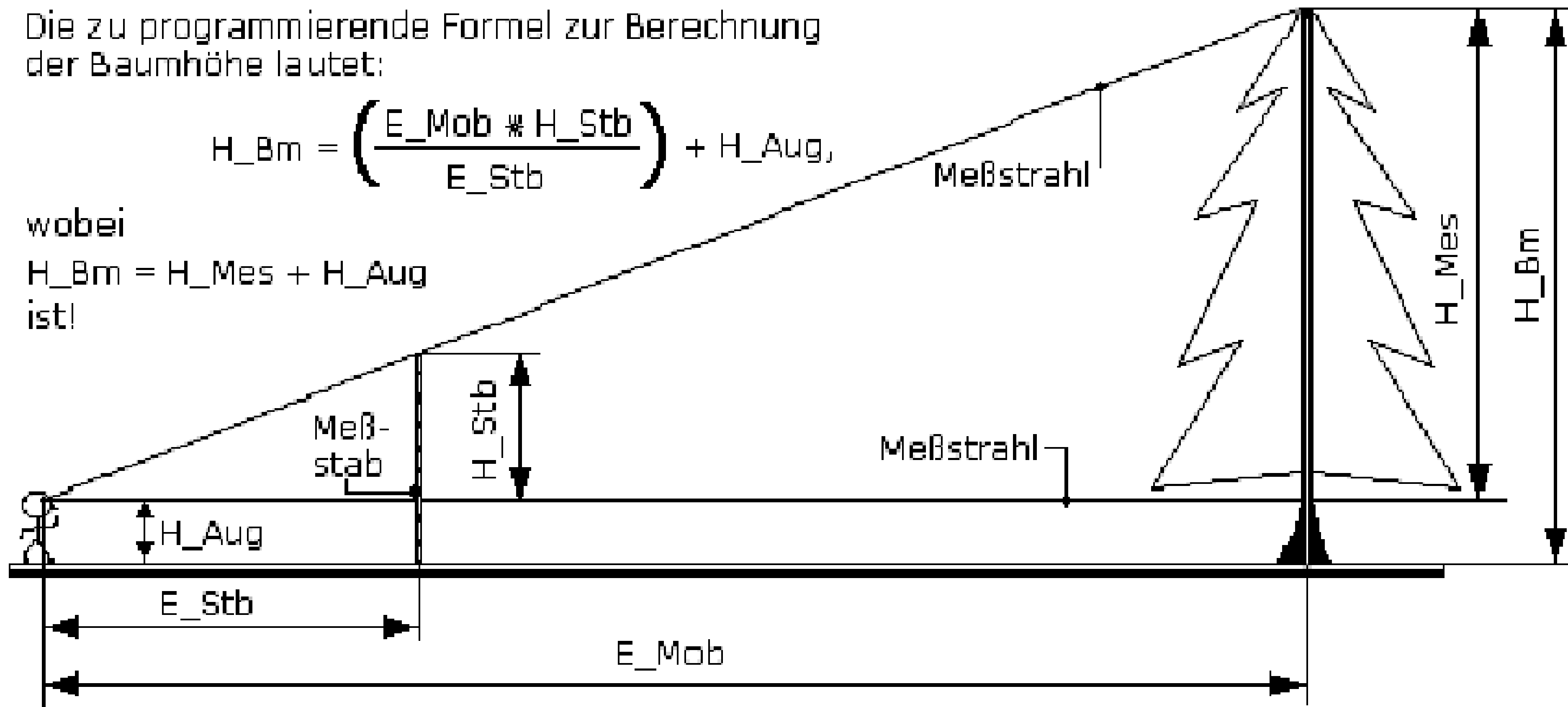
Die ermittelten Messergebnisse für jeden einzelnen Baum sind;

1. Die Augenhöhe (H_Aug) der messenden Person in Meter [m]
2. Die abgelesene Höhe (H_Stb) am Messstab in Meter [m]
3. Die Entfernung (E_Stb) zum Messstab in Meter [m] und
4. Die Entfernung (E_Mob) zum Messobjekt Baum in Meter [m] (siehe Skizze)

Die Anzahl der vermessenen Bäume wird nach dem Programmstart als erstes eingegeben.

Die Ausgabe der Baumhöhen soll am Ende in Tabellenform mit der Nummer des Messobjektes und der Höhe des Messobjekts erfolgen.

Skizze Strahlensatz



Anwendungsbeispiel (2)

BBS

```
int main() {
int Anz_Bme, i;
float H_Auge, E_Mobj, E_Stab, H_Stab, H_Bm[100];
cout << "\nBestimmung der Baumhoehen im Park\n\n";
cout << "Wieviel Baeume wurden vermessen? „  ; cin >> Anz_Bme;
for (i = 0; i < Anz_Bme; i++) {
  cout << "\nBaum-Nr.: " << (i + 1) << "\n";
  cout << "Entfernung Vermesser-Messobjekt: "; cin >> E_Mobj;
  cout << "Entfernung Vermesser-Messstab  : "; cin >> E_Stab;
  cout << "Hoehe, am Messstab abgelesen   : "; cin >> H_Stab;
  cout << "Augenhoehe des Vermessers      : "; cin >> H_Auge;
  H_Bm[i]= (E_Mobj * E_Stab / H_Stab) + H_Auge;
}
cout << "\n\tLfd.-Nr.\tBaumhoehe\n\t----------------------\n";
for (i = 0; i < Anz_Bme; i++) {
  cout << "\t" << (i+1) << "\t\t" << H_Bm[i] << "\n";
} cout << "Programmende\n\n"; return 0;}
```

Von der Aufgabe zur Lösung (1)

BBS

1. Aufgabenstellung
(Die Textaufgabe ist die Wirklichkeit)

Gesucht wird die größte und die kleinste Zahl einer Zahlenreihe von beliebiger Länge (maximal 200 Zahlen!).

Die Länge der Zahlenreihen und die Zahlen sollen über die Tastatur eingegeben werden.

Die jeweils größte und kleinste Zahl soll auf dem Bildschirm erkennbar ausgegeben werden.

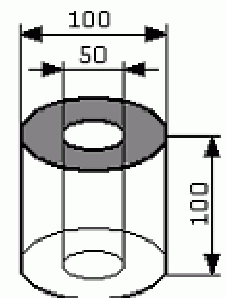
Von der Aufgabe zur Lösung (2)

2. Aufgabenstellung

An beliebig vielen Hohlzylindern von konstanter Länge sollen die Innendurchmesser so verändert werden, dass jeder Hohlzylinder die Masse eines Einheitszylinders bekommt.

Die Hohlzylinder aus Stahl haben eine Länge von 10 cm. Der Einheitszylinder besitzt einen Außendurchmesser von 10 cm und einen Innendurchmesser von 5 cm (siehe Skizze, alle Maße in mm!).

Nach Eingabe des aktuellen Außendurchmessers soll der Innendurchmesser neu berechnet werden. Die Eingabe eines zu kleinen Durchmessers beendet das Programm. Dichte von Stahl: 7,86 g/cm^3.



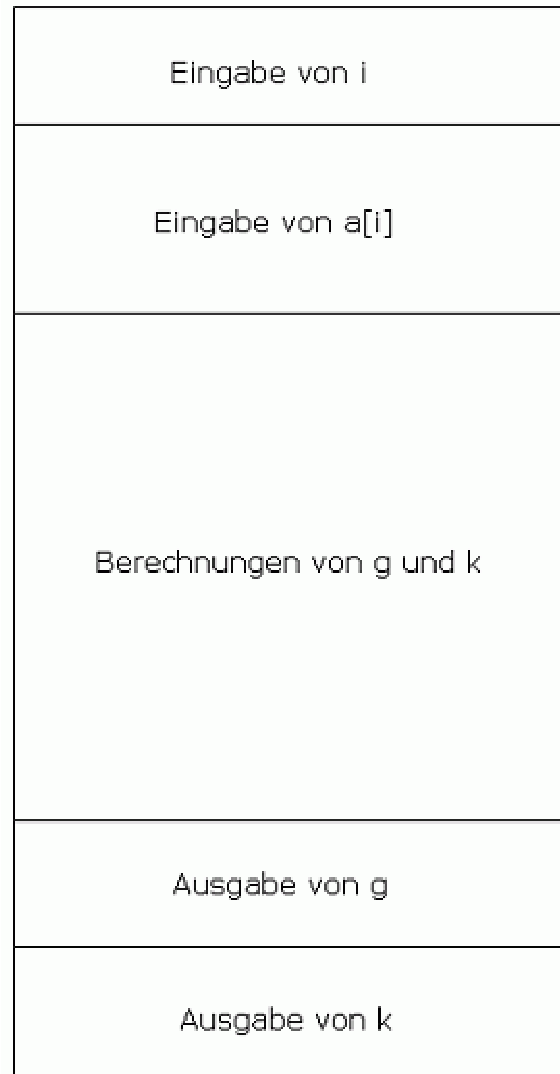
Lösungsansätze (1 und 2)

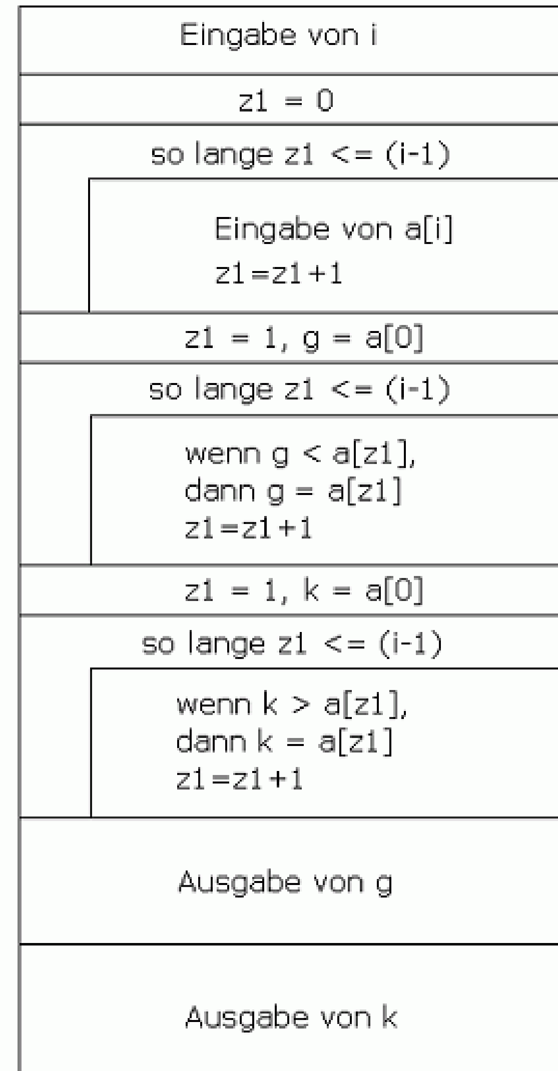
- Suche nach mathematischen Funktionen und Operationen
- Aufschreiben eines kleinen Beispiels
- Festlegung der Größen für eine Initialisierung
- Mit dem PAP oder Struktogramm beginnen: Eingabe — Verarbeitung — Ausgabe.

Lösungsansatz Aufgabe 1

1. Notieren einer Beispiel-Folge:
   i = 10
   5 2 3 1 9 2 4 6 8 2
2. Definieren der Zählvariablen z1:
   z1: 0 1 2 3 4 5 6 7 8 9
   5 2 3 1 9 2 4 6 8 2
3. Definition a[z1] für z1 = 0: g = a[0] (Achtung [0])
   a[0] gesetzt als erste größte Zahl der Folge
4. Vergleich von g mit a[(z1=z1+1)] mit Verzweigungs-Entscheidung:
   Ist a[z1] größer als g, dann wird g = a[z1];
   Ist a[z1] nicht größer als g, dann bleibt g erhalten,
   und z1 wird erneut z1=z1+1
5. Vergleich von k mit a[(z1=z1+1)] mit Verzweigungs-Entscheidung:
   Ist a[z1] kleiner als k, dann wird k = a[z1];
   Ist a[z1] nicht kleiner als k, dann bleibt k erhalten,
   und z1 wird erneut z1=z1+1

Struktogramm Aufgabe 1





Quellcode-Beispiel Aufgabe 1v1


```
#include <iostream>
using namespace std;
int main() {
    int i, g, k, z1;
    int a[200];          // Datenfeldvereinbarung mit 200 int-Elementen 0-199

    cout << ″Bestimmung der groeszten und kleinsten Zahl einer Reihe\n″;
    cout << ″Wie lang soll die Reihe sein (Anzahl der Zahlen)? ″;
    cin >> i;
    cout << ″\nEingabe der Zahlenreihe a[i]\n“;
    for (z1=0; z1<=(i-1); z1++) {
         cout << ″a[″ << z1 << ″] = ″;
         cin >> a[z1];
    }
    g = a[0];
    for (z1=1; z1<i; z1++) {
         if (g<a[z1]) g=a[z1];
    }
    cout << ″\n Die groeszte Zahl lautet: ″ << g << endl;
    k = a[0];
    for (z1=1; z1<i; z1++) {
         if (k>a[z1]) k=a[z1];
    }
    cout << ″\n Die kleinste Zahl lautet: ″ << k << endl;
    getchar();
    return 0;
}
```

Quellcode-Beispiel Aufgabe 1v2

```
/* Beispielloesung von Yannick Richter */
#include <iostream>
using namespace std;
int main() {
    int i, g=-32768, k=32767;
    char oe=148, sz=225;

    cout << ”Bestimmung der gr”<<oe<<sz<<”ten und kleinsten Zahl einer Reihe\n”;
    cout << ”Wie lang soll die Reihe sein (Anzahl der Zahlen)? ”;
    cin >> i;
    int a[i};                  // Datenfeldvereinbarung mit nur i int-Elementen

    cout << ”\nEingabe der Zahlenreihe a[i]\n“;
    for (int z1=0; z1<=(i-1); z1++) {
         cout << ”a[” << z1 << ”] = ”;
         cin >> a[z1];
         if (g<a[z1]) g=a[z1];
         if (k>a[z1]) k=a[z1];
    }

    cout << ”\n Die gr”<<oe<<sz<<”te Zahl lautet: ” << g << endl;
    cout << ”\n Die kleinste Zahl lautet: ” << k << endl;
    getchar();
    return 0;
}
```

Lösungsansatz Aufgabe 2

BBS

1. Die mathematischen Zusammenhänge

1.1 Bestimmung der Masse des Einheitszylinders

$Me = V * y$ mit

$V = (\pi * Ar^2 - \pi * Ir^2) * h$; $y = 7{,}86\ g/cm^3$

$Me = (\pi * 10^2\ cm^2 - \pi * 5^2\ cm^2) * 10\ cm * 7{,}86\ g/cm^3 = 18519{,}7\ g = 18{,}52\ kg$

1.2 Bestimmung des inneren Radius Ir durch Umstellung der Formel

$Me = \pi * Ar^2 * h * y - \pi * Ir^2 * h * y \quad | - (\pi * Ar^2 * h * y) \quad | * (-1)$

$\pi * Ar^2 * h * y - Me = \pi * Ir^2 * h * y \quad | : (\pi * h * y)$

$Ir^2 = Ar^2 - Me/(\pi * h * y)$ = D; mit $D > 0$!

$Ir = \sqrt{Ar^2 - Me/(\pi * h * y)}$

2. Der algorithmische Ablauf (für ein Struktogramm)

2.1 Einmalige Berechnung von Me

2.2 Ausgabe von Me

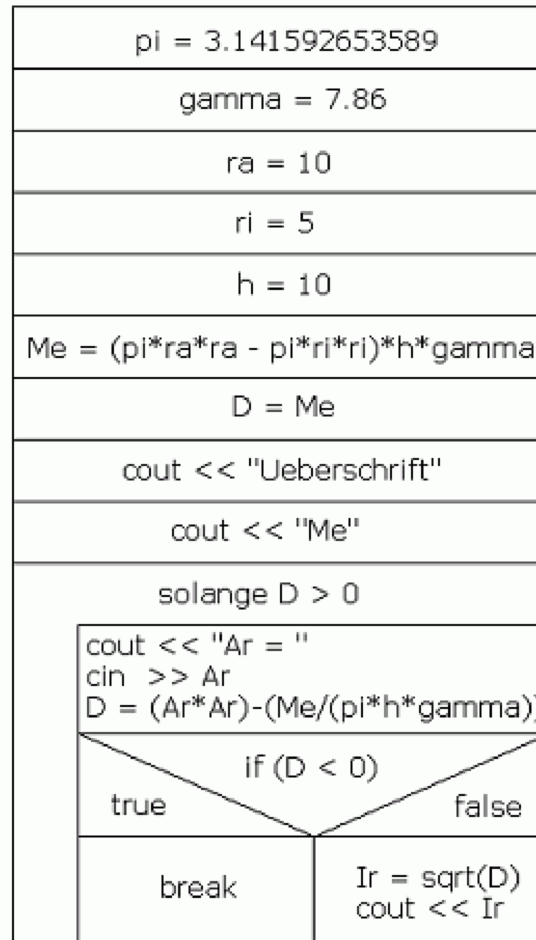
2.3 Eingabe eines aktuellen Außenradius Ar

2.4 Berechnung von Ir nach obiger Formel

2.5 Ausgabe des berechneten Ir

2.6 Wenn D > 0 wieder zu Punkt 2.3 zurück, sonst Ende

Struktogramm Aufgabe 2



Quellcode-Beispiel Aufgabe 2

```
#include <iostream>
#include <cmath>          // wird benoetigt fuer sqrt()
using namespace std;
int main {
   double Me, Pi, gamma, Ar, Ir, ra, ri, h, D;
   gamma = 7.86;          // Spezifische Dichte von Stahl
   Pi = 3.141592653589;   // Konstante Pi
   ra = 10;               // Vorgegebener Einheitshohlzylinder
   ri = 5;                // mit vorgegebenen Radien ra, ri
   h = 10;                // konstante Höhe aller Hohlzylinder
   Me = (Pi*ra*ra – Pi*ri*ri)*h*gamma;
   D = Me;
   cout << ”Bestimmung des Innenradius eines Stahlhohlzylinders\n”;
   cout << ”Masse des Einheitszylinders: Me = ” << Me << endl;
   while (D>0) {
        cout << ”\n Bitte geben Sie den neuen Auszenradius Ar ein: ”;
        cin >> Ar;
        D = (Ar*Ar)-(Me/(Pi*h*gamma));
        if (D<0) break;
        Ir = sqrt(D);
        cout << ”\n Der Innenradius Ir betraegt ” << Ir << ” cm.” << endl;
   }
   cout << ”\n\n EOF - End Of Fun :-) ”;
   getchar();
   return 0;
}
```

Gut geschrieben...

"Ein gut geschriebenes Programm

- *hat ein sauberes Layout,*
- *verwendet sinnvolle Namen,*
- *ist ausführlich kommentiert und*
- *verwendet Konstrukte der Sprache derart,*
  dass maximale Sicherheit und Lesbarkeit des Programmes erreicht werden.

Die Erstellung eines solchen Programms erfordert vom Programmierer Sorgfalt, Disziplin und ein gutes Stück handwerklichen Stolz."

Ian Sommerville, "Software Engineering", Addison-Wesley 1987