Elementare Datentypen in C++

- bool
- signed/unsigned char
- signed/unsigned short int
- signed/unsigned int
- signed/unsigned long int
- (signed/unsigned long long int)
- float
- double
- long double
- void
- enum

char

- Der **`char`** Datentyp ist groß genug, um ein Zeichen, des auf dem System benutzten Zeichensatzes, zu speichern
- Typischerweise: 1 Zeichen = 8 Bit = 1 Byte
- Es hängt von Compiler ab, ob ein **`char`** vorzeichenbehaftet ist oder nicht
- z.B. `char a = 'a';`

ASCII Tabelle 1/2

- Zeichen werden als char gespeichert, wobei jedes Zeichen einem numerischen Wert entspricht
- auf den meisten Systemen sind Zeichen 1 Byte gross und die Zuordnung der Zahlenwerte zu den Zeichen wird über eine Tabelle erledigt
- die am weitest verbreitete Tabelle ist die ASCII Tabelle (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange)

ASCII Tabelle 2/2

- Auf den meisten Linux/Unix Sytemen kann man die ASCII Tabelle mit dem Befehl **man ascii** ansehen
- Die Tabelle enthält 128 Zeichen
- Die Buchstaben A-Z haben die Werte 65-90
- Die Buchstaben a-z haben die Werte 97-122
- Die Ziffern 0-9 haben die Werte 48-57

char Definition

```
char a='0';
char a=48;

char b='A';
char b=65;
b+=2;              // b = 67 = 'C'
```

(unsigned) short int

- short int = short = signed short = signed short int
- wenn nicht anders angegeben ist ein **`short int`** immer vorzeichenbehaftet
- z.B. ***`unsigned short x = 12;`***

(unsigned) long int

- long int = signed long int = long
- wenn nicht anders angegeben ist ein **`long int`** immer vorzeichenbehaftet
- z.B. ***`Long x = -341245671L;`***

(unsigned) long long int

- long long int = long long
- wenn nicht anders angegeben ist ein **`long long int`** immer vorzeichenbehaftet
- ist eine **Erweiterung zu C** und nicht C++ !!!
- die meisten C++ Compiler stellen den Datentyp dennoch zur Verfügung, auch wenn er manchmal anders heisst

float / double / long double

- float, double and long double =

  Gleitkommazahlen
- Werte sind immer vorzeichenbehaftet
- z.B.

```
float f = 3.14159f;
double d = 3.14159;
long double ld = 3.14159L;
```

sizeof

- mit sizeof kann man die Grösse eines Objektes oder eines Datentyps in Byte ermitteln.

- zwei Notationen
  - sizeof(Name des Datentyps)
  - sizeof VARIABLENNAME

sizeof Beispiel

Größen der Datentypen in GCC 4.0.1 auf MacOS X 10.4:

```
short boolSize=sizeof(bool);                // 1
short charSize=sizeof(char);                // 1
short shortSize=sizeof(short);              // 2
short intSize=sizeof(int);                  // 4
short longSize=sizeof(long);                // 4
short longLongSize=sizeof(long long);       // 8
short floatSize=sizeof(float);              // 4
short doubleSize=sizeof(double);            // 8
short longDblSize=sizeof(long double);      // 16
```

Größenbeschränkungen der einfachen Datentypen

- sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)
- sizeof(*) = sizeof(unsigned *) = sizeof(signed *)
- sizeof(float) <= sizeof(double) <= sizeof(long double)

void

- Pseudodatentyp
- es gibt keine Objekte vom Typ **`void`**

  z.B. ***`void v;  // das ist falsch`***
- **`void`** wird zum Beispiel benutzt, um anzugeben, dass eine Funktion keinen Rückgabewert besitzt.

Was sind Funktionen?

Eine Funktion ist ein in sich geschlossener, wiederverwendbarer Programmteil, der eine bestimmte Aufgabe erfüllt.

Funktionen können Argumente übernehmen und einen Wert zurückliefern.

Aufbau einer Funktion

- Funktionen besitzen:
  - einen Namen
  - einen Rückgabewert
  - eine Liste von Argumenten
  - einen Funktionskörper

```
double meineFunktion(int zahl1,int zahl2)
{
    ....
}
```

Allgemeines zu Funktionen

- Werden meist aufgeteilt in eine Deklaration und in eine Definition.
- Jede Funktion für die ein Rückgabewert deklariert wurde, muss ein Objekt dieses Typs mit der **`return`** Anweisung zurückliefert werden
- Es darf mehrere **`return`** Anweisungen in einer Funktion geben und sie dürfen an beliebigen Stellen stehen

Funktionsnamen

- Funktionsnamen müssen mit einem Buchstaben oder einem Unterstrich (_) beginnen, der Rest darf aus Buchstaben Zahlen und Untertrichen bestehen.
- Funktionsnamen dürfen nicht identisch sein mit irgendwelchen C++ Schlüsselwörtern wie z.B. **`double`** oder **`class`**

Funktionssignatur

- Die Kombination aus dem Namen und der Argumentenliste bezeichnet man als die Signatur der Funktion

  double meineFunktion(int,double,char);

- Der Rückgabewert ist nicht Teil der Signatur!
- Es darf keine zwei Funktionen mit derselben Signatur in einem Programm geben

Funktionsdeklaration

- Eine Funktion muss deklariert werden bevor man sie benutzen kann.
- Die Argumente in der Argumentenliste dürfen Namen haben, es ist aber nicht zwingend notwendig.
- Die Argumentenliste darf leer sein.
- Soll eine Funktion keinen Wert zurückliefern, hat sie den Rückgabewert **`void`**.

Deklarations Beispiele

```
int meineFunktion();
int meineFunktion(void);

void meineZweiteFunktion(int a);
void meineZweiteFunktion(int);

double divide(int,int);
```

Funktionsdefinition

- Definition sieht aus wie Deklaration, nur wird der Funktionskörper anstelle das Semikolons angegeben.
- Eine Funktion darf genau einmal in einem Programm definiert werden
- Die Argumente in der Argumentenliste müssen Namen haben, wenn sie im Funktionskörper benutzt werden
- Funktionen mit Rückgabetyp **void** brauchen **keine return** Anweisung

Funktionsdefinition - Beispiel

```
int meineFunktion(double x)
{
     ...
      return EIN_INT;
}
```

- die **`return`** Anweisung kann an beliebiger Stelle im Funktionskörper stehen.
- es kann auch mehrere **`return`** Anweisung in einem Funktionskörper geben.

Benutzen von Funktionen

```
unsigned int fakultaet(unsigned int zahl)
{
    ...
}

int main()
{
    int x=6;
    std::cout << x << "!" << " = ";
    std::cout << fakultaet(x) << "\n";
    return 0;
}
```

Felder (Arrays)

- Datenstruktur um eine Menge gleichartiger Objekte zu speichern.
- Felder können beliebig viele Dimensionen haben.
- Um ein Feld zu deklarieren werden hinter den Variablennamen noch **eckige Klammern** geschrieben.
- In den Klammern steht die Länge des Feldes
- Bei mehrdimensionale Feldern gibt man so viele Klammerpaare an, wie das Feld Dimensionen besitzt.

Deklaration von Feldern

```
int a[5];

char c[5];

int a[2][2];

float c[3][3][4];
```

Deklaration und Initialisierung von Feldern

```
int a[5]={2,4,6,8,10};

c[4]={'T','e','s','t'};


int a[2][2]={{1,2},{3,4}};

// folgendes ist auch gültig,
//sollte man aber nicht machen

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

Deklaration und Initialisierung von Feldern

- Wird ein Feld bei der Deklaration gleichzeitig initialisiert, so kann man die Größe der äußersten Dimension weglassen

  z.B.

```
int a[]={1,2,3,4}; // int a[4]={1,2,3,4}

int a[][]={{1,2},{3,4}}; // falsch
int a[][2]={{1,2},{3,4}}; // richtig
```

Zugriff auf Felder

- Einzelne Elemente werden über ihren Index im Feld angesprochen.
- Das erste Element hat den Index 0.

z.B.

```
int a[]={1,2,3,4,5};
int x=a[2];
++x;
a[4]=x;
```

C Strings (char Felder)

- Zeichenkette in C sind eindimensionale Felder vom Typ char.
- C Strings sind 0 terminiert, d.h. enden mit dem Zeichen '\0'
- es gibt eine Reihe von Funktionen, um solchen Zeichenketten zu manipulieren (#include <string.h>)
- Das Arbeiten mit C Strings birgt gewisse Sicherheitsrisiken (e.g. Pufferüberlauf)

Deklaration and Initialisierung eines C Strings

```
char s[]=”Test”;

char s[]={'T','e','s','t','\0'};
char s[]={'T','e','s','t',  0 };
```

Einfache Eingabe mit cin

- entsprechend der Ausgabe mit **cout** gibt es in **iostream** auch ein Objekt, um Daten von der Tastatur einzulesen (**cin**)
- z.B.

```
int x=0;
std::cin >> x;
```

Einlesen einer Zeichenkette von der Tastatur

```
char nachname[20];
std::cin >> nachname;
```

Die Eingabe wird mit RETURN beendet und alles bis zum ersten Leerzeichen-Zeichen (whitespace, tab, return) wird in der Variable **nachname** gespeichert.

Pufferüberlauf

```
char nachname[20];
std::cout << "Nachname: ";
std::cin >> nachname;
```

"Leutheusser-Schnarrenberger"

Eingabe mit getline

- Das Objekt **cin** verfügt über eine Methode mit der sich eine ganze Zeile einlesen lässt.
- Der Methode werden zwei Argumente übergeben
  - 1) die C String Variable, in die die Zeichenkette gespeichert werden soll
  - 2) wieviele Zeichen die Funktion lesen soll
- Sicherer als über >>-Operator in Verbindung mit C Strings
- Liest eine ganze Zeile mit Leerzeichen (whitespaces)

getline - Beispiel

```
#include <iostream>

int main()
{
    char name[200];
    std::cout << "Name: ";
    std::cin.getline(name,200);
    std::cout << "Sie heissen ";
    std::cout << name << ".\n";
    return 0;
}
```