

Themen

- Formatierte und unformatierte Eingabe
- Bedingungsoperator
- Namespaces
- Kommandozeilenargumente

Formatierte Eingabe mit `cin`

- Die Formatierung der Eingabe ist der Ausgabe sehr ähnlich:
 - Die Flags werden auf dieselbe Art und Weise gesetzt und ausgelesen wie bei der Ausgabe:
setf, unsetf, flags
 - Es werden **dieselben Flags** verwendet
 - Die Flags haben **dieselbe Bedeutung** wie bei der Ausgabe

Unformatierte Eingabe mit `cin`

- `cin` ist eine Instanz der Klasse `istream`
- besitzt z.B. die Methoden `get` und `getline` zur unformatierten Eingabe
- Die Methoden zur unformatierten Eingabe lesen im Gegensatz zum Eingabeoperator auch Leerzeichen

Beispiel – unformatierte Eingabe mit cin

```
char name[25];  
cout << "Name: ";  
cin.getline(name, 25);  
cout << "Sie heissen ";  
cout << name << ".\n";
```

```
Name: Tom Schmidt  
Sie heissen Tom Schmidt.
```

Beispiel – formatierte Eingabe von boolean Werten

```
bool b=false;
cout << "0 oder 1 eingeben: ";
cin >> b;
cout.setf(ios_base::boolalpha);
cout << b;
```

```
0 oder 1 eingeben: 1
true
```

Beispiel – formatierte Eingabe von boolean Werten

```
bool b=false;
cout << "false oder true eingeben: ";
cin.setf(ios_base::boolalpha);
cin >> b;
cout.setf(ios_base::boolalpha);
cout << b;
```

false oder true eingeben: true
true

Das Eingeben von 0 oder 1 funktioniert aber **nicht** !!!

Beispiel – formatierte Eingabe von Integer-Zahlen

```
int i=0;  
cout << "Zahl: ";  
cin.setf(ios_base::hex,  
         ios_base::basefield);  
cin >> i;  
cout << i;
```

Zahl: 1B
27

Beispiel – formatierte Eingabe von Zeichenfeldern 1/3

```
char name[6];  
cout << "Vorname: ";  
cin.width(6);  
cin >> name;  
cout << "Sie heissen ";  
cout << name << ".\n";
```

Vorname: Tom
Sie heissen Tom.

Beispiel – formatierte Eingabe von Zeichenfeldern 2/3

```
char name[60];  
cout << "Name: ";  
cin.width(60);  
cin >> name;  
cout << "Sie heissen ";  
cout << name << ".\n";
```

Name: Tom Schmidt
Sie heissen Tom.

Der Eingabeoperator liest nur bis zum **ersten** Leerzeichen !!!
Der **Rest** der Eingabe verbleibt im **Eingabepuffer**.

Beispiel – formatierte Eingabe von Zeichenfeldern 3/3

```
char name[50];  
char ort[100];  
cout << "Name: ";  
cin >> name;  
cout << "Ort: ";  
cin >> ort;  
cout << "Sie heissen " << name;  
cout << " und wohnen in ";  
cout << ort << "." << endl;
```

Name: **Hans Peter**

Ort: Sie heißen Hans und wohnen in Peter.

Eingabefehler 1/2

```
float f=0.0f;  
cout << "Gewicht: " << endl;  
cin >> f;  
cout << "Sie wiegen " << f;  
cout << "Kilo." << endl
```

Gewicht: Hans

Sie wiegen 0 Kilo.

Eingabefehler 2/2

```
float f=0.0f;
cout << "Gewicht: " << endl;
cin >> f;
for( ; !cin.good() ; )
{
    cin.clear();
    cin.ignore(2000000, '\n');
    cout << "Gewicht: " << endl;
    cin >> f;
}
```

strtod / strtof / strtol / strtoul

- Teil der C Bibliothek
- deklariert in `<stdlib.h>`
- konvertieren C Strings in Zahlen
- ersetzen die veralteten Funktionen *atof*, *atoi*, etc.
- Dokumentation:
 - **man strtod**
 - **man strtol**

Bedingungsoperator: ? :

- Besitzt 3 Operanden, die durch ? bzw. : voneinander getrennt werden:
 - `operand1 ? operand2 : operand3`
 - `bedingung ? ausdruck1 : ausdruck2`
- Anwendung:
 - `variable = bedingung ? ausdruck1 : ausdruck2`
 - Variable muss **skalar** sein
 - Wert wird auf Grund der Bedingung bestimmt:
 - Bedingung **erfüllt**: `variable = ausdruck1`
 - Bedingung **nicht erfüllt**: `variable = ausdruck2`

Bedingungsoperator - Bsp

```
x = (a > b) ? a : b;
```

```
// ist identisch zu
```

```
if (a > b) {  
    x = a;  
}  
else {  
    x = b;  
}
```

Bedingungsoperator – Bsp 2

// Verschachtelung möglich

```
x = (a > b) ? ((a > c) ? a : c) : ((b > c) ? b : c);
```

// Was passiert hier?

Bedingungsoperator – Bsp 2

```
// Verschachtelung möglich
```

```
x = (a > b) ? ((a > c) ? a : c) : ((b > c) ? b : c) ;
```

```
// Vorsicht: Mehrfache Verschachtelung macht  
Programme unübersichtlich und schwerer  
nachvollziehbar – hier wären if-else  
Anweisungen besser geeignet
```

Bedingungsoperator – Bsp 3

// häufige Verwendung:

```
int zahl;  
std::cout << "Bitte eine Zahl eingeben: ";  
std::cin >> zahl;  
std::cout << "Die eingebene Zahl ist ";  
std::cout << (zahl%2) ? "ungerade" : "gerade";
```

Namespace

- **Namensraum:** kennzeichnet Einheiten, die zusammengehören
 - z.B. Die gesamte C++ Standard Library befindet sich im namespace **std**
- Deklarationen und Definitionen unter einem Namen **zusammenfassen** und gegen andere Namen **abgrenzen**
- Eigene Namensräume erstellen:
 - Vorsicht bei Verwendung von Namen, die es bereits in C++ gibt

namespaces definieren

// Schlüsselwort: `namespace`, gefolgt vom `Namen` und
Code in geschweiften Klammern

```
namespace meinNS
```

```
{
```

```
    int var;
```

```
    void ausgabe() {
```

```
        ...
```

```
    }
```

```
}
```

namespaces definieren

```
// Prototypen in Header-Dateien
```

```
namespace myNS
{
    int var;
    void ausgabe();
    void eingabe();
    ...
}
```

namespaces - Zugriff

```
// Bereichsaufloesungsoperator ::
```

```
void main()  
{  
    myNS::ausgabe();  
}
```

namespaces benutzen

```
// using: freigeben aller Namen eines Namensraums  
für die implizite Verwendung
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main()  
{  
    cout << "test" << endl;  
}
```

```
// Vorsicht bei der Verwendung:  
Kollisionen mit anderen Namensräumen möglich
```

namespaces - Beispiel

```
#include <iostream>
using namespace std;

namespace myNS {
    int var;
    class Test {
    public:
        void ausgabe() {
            cout << ::var << " globale Variable";
            cout << var << " lokale Variable";
        }
    }
}

int main () {
    int var = 3;
    myNS::var=7;
    myNS::Test t;
    t.ausgabe();
}
```


namespaces zusammenfassen

- Wenn Namensräume sich nicht überschneiden, können sie zu einem zusammengefaßt werden
- Dazu diese global in einem neuen bekannt machen

```
// header datei: AlleNS.h
```

```
namespace AlleNS  
{
```

```
    using namespace myNS;
```

```
    using namespace myNS2;
```

```
}
```

namespaces verschachteln

- Namensräume lassen sich beliebig verschachteln

```
namespace myNS
{
    namespace myNS2
    {
        void ausgabe() {...}
    }
}

int main()
{
    myNS::myNS2::ausgabe()
}
```

namespaces - Aliasnamen

- Bei langen Namen oder mehreren Verschachtelungen sinnvoll

```
namespace Das_ist_ein_langer_Name
{
    namespace NS_Zwei
    {
        void ausgabe() {...}
    }
}
```

```
// Aliasnamen
```

```
namespace myNS = Das_ist_ein_langer_Name;
namespace myNS2 = Das_ist_ein_langer_Name::NS_Zwei;
```

Anonyme Namensräume

- Anstatt statischer Funktionen
- Um lokale Funktionen vor anderen Modulen zu verbergen
- Funktionen innerhalb der gleichen Quelltextdatei direkt verwendbar, aber nach außen unsichtbar

Anonyme Namensräume - Bsp

```
#include <iostream>
using namespace std;

namespace
{
    void ausgabe(int a) {
        cout << "Zahl=" << a << endl;
    }
} // anonymer namespace

int main() {
    ausgabe();
}
```

namespaces-Zusammenfassung

- Übergeordneten **Gültigkeitsbereich** für Klassen und Funktionen schaffen:
 - sinnvoll bei mehreren Entwicklern eines Projekts
- Deklaration ist **kumulativ**:
 - weitere Deklarationen des gleichen Namensraums fügen lediglich Elemente hinzu
- **Verwendung**:
 - Explizite Angabe des Namen:
 - `myNS : : ausgabe ( ) ;`
 - Verwendung von **using**, dann sind die Namen im aktuellen Gültigkeitsbereich bekannt
 - `using namespace myNS ;`

Kommandozeilenargumente

- Werden benutzt um einem Programm beim Start Optionen und Daten zu übergeben

z.B.

```
g++ -o Bruch Bruch.cpp
```

```
./fakultaet 12
```

main 1/2

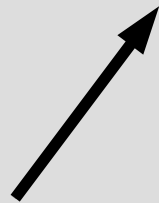
```
int main()
```

```
int main(int argc, char** argv)
```

```
int main(int argc, char* argv[])
```


main 2/2

```
int main(int argc, char* argv[])
```



Anzahl der Argumente Feld mit **argc** Argumenten

- Jedes Arguments ist ein **C-String** (**char**-Feld oder Zeiger auf **char**)
- Der **erste** Argument hat den Index 0 und ist immer der **Name des Programmes**
- Auf die einzelnen Argumente kann man über den Index und die Feldvariable **argv** zugreifen.

main Beispiel

```
// fakultaet.cpp
#include <iostream>

...

int main(int argc, char* argv[])
{
    char* s=argv[1];
    int number=convertToNumber(s);
    cout << fakultaet(number);
    cout << std::endl;
    return 0;
}
```