

CPP

- Eine der frühen objektorientierten Programmiersprachen
- Entwickelt von Bjarne_Stroustrup
- (https://en.wikipedia.org/wiki/Bjarne_Stroustrup)
- Zuvor Simula 67 als erste objektorientierte Sprache

Ein paar Kleinigkeiten zum Einstieg

- Includes und Namensräume
- Ausgabe in C++
- Ausgabeformatierung mit Manipulatoren
- Eingabe in C++
- Defaultargumente in C++
- Überladene Funktionen
- Referenzen

Vorab das übliche Hello

```
#include <iostream>

using namespace std;

int main()
{
    cout << "hello you";
    cout << endl;
    cout << "hello you - short version" << endl;
    return 0;
}
```

Includes

```
#include <iostream>
```

- Includefiles ohne .h
- iostream – das Includefile für die Standard-I/O
- c- Includefiles werden auch ohne .h, dafür mit einem führenden c angegeben (z.B.:
`#include <cstring>`)

Namensräume

```
using namespace std;
```

- Namensräume kennen wir in C von Strukturen. In Strukturen können wir Bezeichner verwenden, die es in der Umgebung ebenfalls gibt.
- In C++ können wir Namensräume auch außerhalb von Strukturen definieren (Beispiel nächste Folie).
- Namen aus einem Namensraum müssen voll Qualifiziert angegeben werden, das heißt, Sie werden gebildet aus Namen des Namensraumes :: Bezeichner.
- Mit `using namespace ...` können wir den ersten Teil des Bezeichners weglassen

Beispiel Namensräume

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

```
namespace first
{
    int x = 5;
    int y = 10;
}
```

```
namespace second
{
    double x = 3.1416;
    double y = 2.7183;
}
```

```
int main ()
{
    using namespace first;
    cout << x << endl;
    cout << y << endl;
    cout << second::x << endl;
    cout << second::y << endl;
    return 0;
}
```

Namensräume

- Im Beispiel werden Zwei Namensräume `first` und `second` definiert
- In beiden Namensräumen gibt es die Bezeichner `x` und `y` mit unterschiedlichem Typ
- `first::x` bildet einen voll qualifizierten Bezeichner, ebenso, wie `second::x`.
- Nach Angabe von `using namespace first;` kann die Angabe `first::` entfallen.
- Es muss aber immer eine eindeutige Zuordnung gewährleistet sein.

Namensräume

- `using namespace first;` und `using namespace second;` schließen einander unbedingt aus (entweder/oder)
- Zur Nutzung der Standardheader ist die Angabe von `using namespace std;` sinnvoll, ansonsten müssen alle verwendeten Bezeichner aus der lib mit `std::` qualifiziert werden.

Die main-Funktion zurück zu hello.cpp

- Eine gute, alte Bekannte aus c- Zeiten.
- Ist aufgebaut, wie in c.

```
int main()  
{  
    . . . .  
    return 0;  
}
```

```
int main(int argc,  
         char*argv[ ] )  
{  
    . . . .  
    return 0;  
}
```

Ausgabe auf Konsole

```
cout << "hello you";  
cout << endl;
```

- `cout` ist ein Ausgabekanal.
- Der voll qualifizierte Name ist `std::cout`.
- `<<` ist der Insertionoperator. Er fügt Daten in den Datenausgabestrom ein, in unserem Falle die Zeichenkette `"hello you"`, die wie in C nullterminiert und vom Typ `char*` bzw. `const char*` ist.
- `cout<<endl;` bewirkt die Ausgabe eines Zeilenumbruchs.

Ausgabe auf Konsole

```
cout << "hello you - short version"<< endl;
```

- Mehrere Ausgaben können auch, wie oben zu sehen, zusammengefasst werden.

helloInt.cpp

```
#include <iostream>
#include <cerrno>
#include <cstdlib>

using namespace std;

int main(int argc, char*argv[])
{
    if (argc!=2)
    {
        cerr<<"usage " << argv[0] << " <numeral>" << endl;
        exit (EPERM);
    }
    else
    {
        int i=atoi(argv[1]);
        cout << "Zahl      :"<< i <<endl;
        i+=1;
        cout << "Zahl+1   :"<< i <<endl;
    }
    return 0;
}
```

C-Headerfiles

```
#include <cerrno>
#include <cstdlib>
```

- Um atoi , exit und errorcodes verwenden zu können, benötigen wir c-System-Includefiles.
- Sie werden in der Form oben inkludiert.
- Für benutzerdefinierte Headerfiles ergibt sich kein Unterschied. Sie werden weiter in "" eingeschlossen und haben die Extension .h oder .hpp oder manchmal auch .hxx.

Standardfehlerausgabe

```
cerr<<"usage " << argv[0] << " <numeral>" << endl;
```

- cerr ist der Standardfehlerausgabekanal und entspricht stderr in c
- Betriebssystemseitig entspricht cerr dem Filehandle/Filedeskriptor 2
- cout entspricht dem Filehandle/Filedeskriptor 1, eben Standardausgabe.

Standardfehlerausgabe

```
cout << "Zahl      :" << i << endl;
```

- In dieser Zeile wird in einer Anweisung eine Zeichenkette, eine Zahl und das Zeilenende ausgegeben.
- Dabei erfolgt eine Standardformatierung bei der Zahlenausgabe, ähnlich wie bei `printf` mit `%d` ohne weitere Formatangabe.

Ausgabeformatierung

- Die Formatierung erfolgt in C++ ebenfalls mit dem Insertionoperator <<.
- Zur Formatierung werden Manipulatoren benutzt.
- Manipulatoren werden in die Augabefolge, wie auszugebende Werte eingefügt.

Ausgabeformatierung

```
cout << setw(8)<< left <<"Zahl:"  
      << setw(6)<< right<< i  
      << endl;
```

```
Zahl:      12  
Zahl+1:    13
```

- Mit `setw` wird die Ausgabefeldweite eingestellt.
- Diese Angabe ist für die jeweils nächste Ausgabe gültig.
- Mit `left` und `right` wird die Ausrichtung der Ausgabe im Ausgabefeld festgelegt. Diese Einstellung bleibt bis zur nächsten Änderung erhalten.

Ausgabeformatierung

- Informationen zu den Manipulatoren sind unter <http://www.cplusplus.com/reference/iomanip/> zu finden.
- Die wichtigsten Manipulatoren hier sind:
 - `setfill` (Füllzeichen setzen)
 - `setw` (Ausgabefeldlänge in Zeichen setzen)
 - `setprecision` (Anzahl der Kommastellen)

Ausgabeformatierung

- Weitere Manipulatoren beeinflussen die Ausgabe über Flags
- Informationen zu diesen Manipulatoren sind unter <http://www.cplusplus.com/reference/ios/> im Abschnitt „Format flag manipulators (functions)“ zu finden.
- Die wichtigsten Manipulatoren hier sind:
 - `hex`, `oct`, `dec` (Zahlenbasis für die Ausgabekonvertierung)
 - `left`, `right` (Orientierung links-/rechtsbündige Ausgabe – nur wirksam in Verbindung mit `setw`)

Eingabe

<http://www.cplusplus.com/reference/istream/istream/operator>>>

- Die Eingabe erfolgt von cin, dem Datenstrom der Standardeingabe.
- Der Operator für die Eingabe ist der Extractionoperator >>, der Daten aus dem Datenstrom herauszieht, und in den Typ der Zielvariablen konvertiert.
- Text wird jeweils nur bis zum nächsten Trennzeichen eingelesen
- Aber Achtung!!! Bei Eingabe von Zeichenketten, die länger als der bereitgestellte Zielspeicherbereich sind, kommt es zum Überschreiben von angrenzenden Bereichen!!!

Eingabe Beispiel1

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

int main(int argc, char*argv[])
{
    char myStr1[16], myStr2[16];
    cout<< "Eingabe 1: ";
    cin >> myStr1;
    cout<< "Eingabe 2: ";
    cin >> myStr2;
    cout<< "Str1: " <<myStr1<<endl;
    cout<< "Str2: " <<myStr2<<endl;

    return 0;
}
```

Eingabe Beispiel1

```
Eingabe 1: hallo  
Eingabe 2: spass  
Str1:hallo  
Str2:spass
```

```
Eingabe 1: 123456789012345678901234567890  
Eingabe 2: hallo  
Str1: 1234567890123456hallo  
Str2: hallo
```

- Im linken Kasten funktioniert alles, wie erwartet, die eingegebenen Zeichen passen problemlos in die Variablen.
- Im rechten Kasten wurden mehr als 15 Zeichen (terminierende 0 beachten!!) eingegeben, es kommt zu Überschreibung.

Leicht geänderter Quelltext

```
char myStr1[16]="", myStr2[16]="";  
cout<< "Eingabe 1: ";  
cin >> myStr1;  
cout<< "Str1: " <<myStr1<<endl;  
cout<< "Str2: " <<myStr2<<endl;  
cout<< "Eingabe 2: ";  
cin >> myStr2;  
cout<< "Str1: " <<myStr1<<endl;  
cout<< "Str2: " <<myStr2<<endl;
```

Nach jeder Eingabe erfolgt hier die Ausgabe beider Eingabepuffer.

```
./a.out  
Eingabe 1: 123456789012345678901234567890  
Str1: 123456789012345678901234567890  
Str2: 78901234567890  
Eingabe 2: Hallo  
Str1: 1234567890123456Hallo  
Str2: Hallo
```

Kommentar dazu

- Nach der ersten Eingabe ist zu sehen, dass sich die eingegebenen Zeichen über myStr1 hinaus bis in myStr2 erstrecken.
- Nach der zweiten Eingabe wurde dieser Teil dann überschrieben, da myStr1 aber nun keine terminierende 0 hat, taucht hallo auch in mystr1 auf.
- Schlussfolgerung: Diese Eingabevariante ist sehr bequem, aber nicht sehr sicher. Der Eingabepuffer sollte hinreichend groß sein.
- Es wird immer nur bis zum nächsten Leerzeichen eingelesen.

Numerische Eingabe

- Numerische Werte können ebenfalls mit dem Extractionoperator eingegeben werden.
- Natürlich muss der Datenstrom dann auch Ziffernzeichen enthalten.
- Es werden nun solange Zeichen extrahiert, wie zu dieser Zahl gehören, bis zum nächsten andern Zeichen.
- Folgen in dem Datenstrom weitere Zeichen, so werden Sie in der nächsten Eingabeoperation aus dem Datenstrom übernommen.

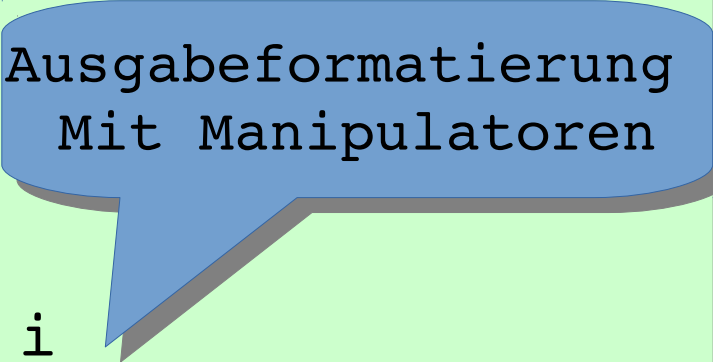
Numerische Eingabe

```
#include <iostream>
#include <iomanip>

using namespace std;

int main(int argc, char*argv[])
{
    int i;
    cout<< "Eingabe einer Zahl: ";
    cin >> i;
    cout << "Eingegebene Zahl: " << i
        <<" 0x" <<hex << setw(8)<< setfill('0')<< i
        <<endl;

    return 0;
}
```



Ausgabeformatierung
Mit Manipulatoren

Eingabe von Strings

- Es gibt für die Eingabe von Zeichenketten eine Funktion, die ähnlich fgets arbeitet.
- Aufruf: `cin.getline(buf,len);`
- Dies bewirkt die die Eingabe einer Zeile von der Standardeingabe nach buf, höchstens aber len-1 Bytes gefolgt von der terminierenden 0.
- Mit dieser Funktion können nun auch Zeichenketten mit Leerzeichen eingelesen werden.

Eingabe mit cin.getline

```
#include <iostream>
#include <iomanip>

using namespace std;

int main(int argc, char*argv[])
{
    char myStr1[16]="", myStr2[16]="";
    cout<< "Eingabe 1: ";
    cin.getline(myStr1,16);
    cout<< "Str1: " <<myStr1<<endl;
    cout<< "Str2: " <<myStr2<<endl;
    cout<< "Eingabe 2: ";
    cin.getline(myStr2,16);
    cout<< "Str1: " <<myStr1<<endl;
    cout<< "Str2: " <<myStr2<<endl;

    return 0;
}
```

Eingabe von Strings

```
./a.out  
Eingabe 1: 123456789012345678901234567890  
Str1: 123456789012345  
Str2:  
Eingabe 2: Str1: 123456789012345  
Str2:
```

- Was ist denn hier nun wieder passiert?
- Eingegeben wurden 30 Zeichen.
- Die gesamte Zeile wurde aus dem Eingabestrom entfernt, außer newline.
- Übernommen wurden 15 Zeichen, und mit einer terminierenden 0 abgeschlossen.

Funktion zur Eingabe von c-Strings

```
ios_base::iostate getcstr(char* ptr, int n)
{
    cin.getline(ptr,n);
    ios_base::iostate state=cin.rdstate();
    if(cin.good());
    else
    {
        cin.clear();
        while (cin.get() != '\n')
        {
            continue;
        }
    }
    return state;
}
```

```
./a.out  
Eingabe 1: 12345678901234567890  
Str1: 123456789012345  
Str2:  
Eingabe 2: hallo  
Str1: 123456789012345  
Str2: hallo  
Eingabe 3: spass  
Str1: 123456789012345  
Str2: spass
```

- Obwohl die erste Eingabe unzulässig lang ist, werden nur $n-1$ Zeichen in den Buffer übernommen und mit 0 terminiert.
- Die weiteren Eingaben finden unbeeinflusst korrekt statt.
- Die überschüssigen Zeichen werden in der while-Schleife aus dem Eingabepuffer entfernt.

Die komplette main-Funktion dazu

```
int main(int argc, char*argv[])
{
    char myStr1[16]="", myStr2[16]="";
    cout<< "Eingabe 1: ";
    getcstr(myStr1,16);
    cout<< "Str1: " <<myStr1<<endl;
    cout<< "Str2: " <<myStr2<<endl;
    cout<< "Eingabe 2: ";
    getcstr(myStr2,16);
    cout<< "Str1: " <<myStr1<<endl;
    cout<< "Str2: " <<myStr2<<endl;
    cout<< "Eingabe 3: ";
    getcstr(myStr2,16);
    cout<< "Str1: " <<myStr1<<endl;
    cout<< "Str2: " <<myStr2<<endl;
    return 0;
}
```