Ein paar Kleinigkeiten zum Einstieg

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

Defaultargumente

- Jede Funktion muss in C++ vor ihrer Verwendung bekannt (definiert oder deklariert) sein.
- Parametern können im Funktionskopf Werte zugeordnet werden, sogenannte Defaultargumente.
- Dies muss aber unbedingt von rechts nach links geschehen.

Defaultargumente

- Beim Funktionsaufruf können nun Parameter von rechts beginnend weggelasssen werden.
- Es ergeben sich dadurch unterschiedliche Aufrufmöglichkeiten für eine Funktion.
- Gibt es zu einer Funktion sowohl eine Deklaration (Funktionsprototyp) als auch die Funktionsdefinition, so dürfen die Defaultargumente nur im Prototyp angegeben werden.

Defaultargumente

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

void myFunc(int i=5, double d=1.234 )
{
  cout<< i<<'\n' << d << '\n';
}

int main()
{
  myFunc(10,999.99);
  myFunc(10);
  myFunc();
  return 0;
}
```

Defaultargument

Defaultargument

2. Parameter weggelassen
- wird durch Defaultargument ersetzt!

beide Parameter weggelassen
- durch Defaultargumente ersetzt!

Defaultargumente

```
10
999.99
10
1.234
5
1.234
```

10 und 999.99 sind Ergebnis des ersten Aufrufs mit 2 Parametern

```
myFunc(10,999.99);
```

10 und 1.234 sind Ergebnis des zweiten Aufrufs mit 1 Parameter

```
myFunc(10);
```

5 und 1.234 sind Ergebnis des dritten Aufrufs mit ohne Parametern

```
myFunc(10,999.99);
```

Aber:
Weglassen von Aufrufparametern immer nur von rechts nach links!

Funktionsüberladung

- Wichtiger Begriff – merken:
- Überladene Funktionen oder **Funktionsüberladung**
- In objektorientierten Sprachen, nicht nur in c++ kann man **mehrere Funktionen mit gleichem Namen aber verschiedenen Parameterlisten** definieren, bzw. deklarieren.
- Man spricht von **überladenen Funktionen.**

Funktionsüberladung

- In c konnte in einem Gültigkeitsbereich immer nur eine Funktion mit einem Funktionsnamen vereinbart werden.
- In c++ gilt das nicht mehr.
- Intern werden dem Funktionsnamen Kürzel für die einzelnen Parameter angefügt, so dass sich wieder ein eindeutiger Name am Ende ergibt.
- Die Funktion `int calc(int,int)` könnte damit etwas vereinfacht unter dem Namen calc_ii geführt werden.

Funktionsüberladung Beispiel 1

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

int    myabs (int z)    {return abs(z);}
long   myabs (long z)   {return labs(z);}
double myabs (double z){return fabs(z);}

int main()
{
  double d=-123.5;
  cout << "Double:" <<myabs(d)<<endl;
}
```

Aufruf von myabs mit Parameter double

Funktionsüberladung

- Im Beispiel gibt es drei Funktionen myabs mit je einem Parameter unterschiedlichen Typs.
- Grundbedingung bei Funktionsüberladung ist, dass die Parameterlisten der gleichnamigen Funktionen in Typ und/oder Anzahl der Parameter voneinander verschieden sind.
- Auf Grund der Aufrufparameter findet der Compiler die passende Funktion.

Funktionsüberladung Beispiel 2

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

namespace beck
{
int      abs (int z)      {return abs(z);}
long     abs (long z)     {return labs(z);}
double   abs (double z)   {return fabs(z);}
}

int main()
{
  double d=-123.5;
  using namespace beck;
  cout << "Double:" <<beck::abs(d)<<endl;
}
```

Aufruf der Standardfunktion
std::abs
Kein rekursiver Aufruf!!!

Funktionsüberladung

- Da es in diesem Beispiel den Namensraum beck gibt, dürfen unsere Funktionen den Namen abs tragen, den sonst die Standardfunktion hat.
- Die Standardfunktion abs können wir dennoch verwenden.
- Wir müssen nun nur den Namen `abs` unserer Funktionen voll qualifiziert verwenden. `beck::abs`
- Nachfolgendes Beispiel „dreht den Spieß um“.

Funktionsüberladung Beispiel 3

```
#include <iostream>
#include <cstdlib>
#include <cmath>

namespace beck
{
int    abs (int z)   {return std::abs(z);}
long   abs (long z)  {return std::labs(z);}
double abs (double z){return std::fabs(z);}
}
using namespace beck;
int main()
{
  double d=-123.5;
  using namespace beck;
  std::cout << "Double:" <<abs(d)<<std::endl;
}
```

Funktionsüberladung

- An dem Beispiel hat sich in puncto Überladung nichts geändert.
- Wir benutzen hier nicht `using namespace std;` statt dessen benutzen wir `using namespace beck;`
- In der Folge müssen wir alle Elemente von `std` voll qualifiziert ansprechen, können aber unsere abs-Funktionen ohne beck:: aufrufen.

Funktionsüberladung

- Bisher haben sich die Datentypen der Funktionen unterschieden um Funktionsüberladung zu realisieren.
- Im folgenden Beispiel unterscheiden sich die Funktionen durch die Anzahl der Parameter.
- Wichtig: Bei der kombinierten Anwendung von Überladung und Defaultargumenten darf es nicht zu mehrdeutigen Situationen kommen, dies hätte Compilerfehler zur Folge.

Funktionsüberladung Beispiel 4

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

int Max (int a, int b)
{
  return (a>b?a:b);
}

int Max (int a, int b, int c)
{
  return Max(Max(a,b),c);
}

int main()
{
  cout << Max(47,11)  <<endl;
  cout << Max(5, 9, 2)<< endl;
  return 0;
}
```

```
./a.out
47
9
```

Funktionsüberladung

- Die Funktion Max ermittelt aus zwei bzw. drei int-Werten den jeweils größten Wert.
- Auch hier wird aus der Parameterliste des Aufrufs die passende Funktion ermittelt und aufgerufen.
- Eine Funktion

  `int Max (int a, int b, int c=99)` mit Defaultargument führt unweigerlich zum Compilerfehler!
- `... call of overloaded ‘Max(int, int)’ is ambiguous`

Inlinefunktionen

- Eine Funktion kann in c++ mit dem Attribut **`inline`**

  versehen werden.
- Das Attribut bewirkt, dass die Funktion nicht über den üblichen Funktionsaufrufmechnismus aufgerufen wird, sondern es wir ihr Code direkt in die Aufrufstelle kopiert. Dies spart den Overhead, den sonst der Funktionsaufruf mit sich bringt.

Inlinefunktionen

- Inlinefunktionen sind mit expandierten Macros vergleichbar, nur dass hier eine vollständige Typprüfung der Parameter erfolgt.
- Inline sollte nur für kleine Funktionen, die aus nur wenigen Anweisungen bestehen, verwendet werden.

Typisierte Konstanten

- C++ gestattet mit Hilfe des Attributes const, typisierte Konstanten zu definieren.
- Die Syntax entspricht der einer Variablen-definition mit Initialisierung.
- Typisierte Konstanten sind in der Verwendung sicherer als defines, wie wir sie von c her kennen weil Typprüfung möglich ist.
- Eine so definierte Konstante darf überall, wo ein konstanter Wert erwartet wird, stehen.

Typisierte Konstanten

```
#include <iostream>

using namespace std;

const int N=5;
char vC[N];

int main()
{
  cout << "size of vC: "
       << (sizeof vC) << endl;
  return 0;
}
```

```
./a.out
size of vC: 5
```

Das geht in c nicht

Typisierte Konstanten

- Besondere Bedeutung hat const in c++ in Verbindung mit Pointern.
- Ein Pointer kann selbst eine Konstante sein, dann kann dieser Pointer nicht geändert werden, der Wert, auf den der Pointer zeigt, ist aber sehr wohl veränderbar.
- Ein Pointer kann auf eine Konstante zeigen. Dann kann der Pointer modifiziert werden, der Wert, auf den der Pointer zeigt, jedoch nicht.

Typisierte Konstanten

- Schließlisch kann es auch einen konstanten Pointer geben, der auf eine Konstante zeigt.
- Regel:

`const` vor dem Typ eines Pointers definiert einen Pointer auf einen Wert, der nicht überschrieben werden darf.

`const` nach dem Typ eines Pointers definiert einen Pointer, der nicht modifiziert werden kann.

Typisierte Konstanten

- Die erste Variante kennen wir schon von c:
- `int strcmp(const char *s1, const char *s2);`
- Hier zeigen die Pointer auf Werte, die von der Funktion nicht verändert werden (können/dürfen).
- Die zweite Variante sieht etwa so aus:

Typisierte Konstanten

```
#include <iostream>
#include <iomanip>
#include <cstring>
using namespace std;
// Das Urteil des Herrschers:
char vbuf[]="haengen, nicht  laufen lassen";

char * const pbuf=vbuf; // Initialisierung ist zwingend

int main()
{
  pbuf++;
  cout << "size of vbuf: " << setw(4)<<(sizeof vbuf)
       << " Text: " << vbuf << endl;
  cout << "size of pbuf: " << setw(4)<<(sizeof pbuf)
       << " Text: " << pbuf << endl;
  return 0;
}
```

Typisierte Konstanten

- Hier habe wir ihn, den konstanten Pointer.
- Er muss initialisert werden.
- Der Versuch ihn zu incrementieren wird mit mit einem Compilerfehler geahndet:

```
error: increment of read-only variable ‘pbuf’
```

- Das Ändern der Daten, auf die der Pointer zeigt, ist kein Problem, wie nachfolgendes Beispiel zeigt, dieser Umstand rettet Leben!!

Typisierte Konstanten

```
. . .
// Das Urteil des Herrschers
char vbuf[]="haengen, nicht  laufen lassen";

char * const pbuf=vbuf;

int main()
{
  pbuf[ 7]=' ';
  Pbuf[14]=',';
  cout << „Die Message des Boten: << endl;
  cout << "size of vbuf: " << setw(4)<<(sizeof vbuf)
       << " Text: " << vbuf << endl;
  cout << "size of pbuf: " << setw(4)<<(sizeof pbuf)
       << " Text: " << pbuf << endl;
  return 0;
}
```

Es gibt noch mehr so schöne Sätze (hier mit/ohne Komma):

Wir essen jetzt, Opa.

Was, willst Du schon wieder?

Typisierte Konstanten

Das Urteil der Herrschers:
`„haengen,  nicht  laufen lassen“`
Glücklicherweise können wir über unseren konstanten Pointer die Daten ändern in die message des Boten:

```
./a.out
Die message des Boten:
size of vbuf:   30 Text: haengen  nicht, laufen lassen
size of pbuf:    8 Text: haengen  nicht, laufen lassen
```

Typisierte Konstanten

```
// Das Utreil des Herrschers:
char vbuf[]="haengen, nicht  laufen lassen";
// Konstanter Pointer auf konstante Daten
const char * const pbuf=vbuf;

int main()
{
//  pbuf[ 7]=' '; // bringt jetzt Compilerfehler
//  pbuf[14]=','; // assignment of read-only location ...
  cout << "Die message des Boten:" << endl;
  cout << "size of vbuf: " << setw(4)<<(sizeof vbuf)
       << " Text: " << vbuf << endl;
  cout << "size of pbuf: " << setw(4)<<(sizeof pbuf)
       << " Text: " << pbuf << endl;
  return 0;
```

Typisierte Konstanten

- In diesem letzten Fall ist es unmöglich, die Daten zu manipulieren und auch der Pointer kann nicht verändert werden.
- Besondere Bedeutung haben diese Konstrukte, wenn sie in Parameterlisten von Funktionen auftauchen.
- Merke:
  - const vor dem Typ:
    - Die Daten sind konstant.
  - const nach dem Typ:
    - Der Pointer ist konstant, es ist eine Initialisierung des Pointers erforderlich.

C-Code in Headerfiles

Einbinden von C Code

C Code in Headerfiles müssen gesondert gekennzeichnet werden.

```
extern "C" {
  . . .
}
```

Nur so ist ein Linken der zugehörigen Objektdateien/Bibliotheken möglich

Der Scopeoperator ::

- Der scopeoperator beeinflusst die Regeln der Gültigkeit von Bezeichnern.
- Angenommen, es gibt lokal und global jeweils einen Bezeichner mit dem selben Namen, z.B. `amount`.
- Verwendet man innerhalb der Funktion (des Blocks), in der `amount` lokal definiert ist diesen Namen, so ist natürlich dieser lokale Bezeichner gültig.
- Setzt man den Scopeoperator davor, so werden die normalen Gültigkeitsregeln aufgehoben und es wird der globale an Stelle des lokalen Bezeichners gültig.

Der Scopeoperator ::

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

int amount=123;
int main()
{
  int amount=456;
  cout<< "extern Amount: "<<::amount<<endl;
  cout<< "local  Amount: "<<  amount<<endl;
  return 0;
}
```

```
./a.out
extern Amount123
local  Amount456
```