Vorkurs C++ Programmierung

Typkonvertierungen – type casting
Sichtbarkeiten in Klassen
Zeigerarithmetik
Namensräume (namespace)

Letzte Stunde

- Klassen: Zentrales Konzept der OOP
- Zusammenfassung/Kapselung von Daten und Funktionen
- Schablone zur Erzeugung konkreter Instanzen (Objekte)

Heute

- Typkonvertierungen – type casting
- Zeigerarithmetik
- Namensräume (namespace)

Typkonvertierung

- C++ ist eine getypte Sprache, d.h. alle Daten müssen einen Typ haben
- Bei Operationen auf Daten, die unterschiedlichen Typ haben, finden Typkonvertierungen statt („type casting“)
- Bsp.: Zuweisung eines ganzzahligen Typs zu einer Fließpunktzahl:

```
long i;
double d;
i = 42;
d = i;
```

- Konvertierung wird implizit durch den Compiler vorgenommen: Wert von **`i`** wird zu **`double`** gecastet, dann zugewiesen
- Zulässig, weil der **`long`**-Wertebereich (32 bit) in den **`double`**-Wertebereich (64 bit) hineinpasst.

Typkonvertierung

```
int i;        //int-Wertebereich: -(2^31) ... 2^31-1
short s;      //short-Wertebereich -(2^15) ... 2^15-1
i = 1234;
s = i;        //Zuweisung eines größeren zu einem kleineren
              //Datentyp
```

→ Compiler warnt vor möglichem Datenverlust (abhängig von der Warnstufe)

→ 32 bit werden in 16 bit gepresst – das kann nicht gut gehen

Typkonvertierungen – type casting

```
int a = 3;
float b = 14.3;
int ergebnis = a + b;
```

- Wert von a wird vor der Zuweisung zunächst implizit in einen float gecastet und dann das Ergebnis berechnet.
- a + b = 17.3, aber der Wert lässt sich mit einem **`int`** nicht darstellen!
- Was passiert?

  Der Compiler gibt (hoffentlich) eine Warnung aus:

```
$ g++ cast.cpp
cast.cpp: In function `int main(int, char**)':
cast.cpp:7: warning: converting to `int' from `float'
```

  Die Nachkommastellen werden abgeschnitten (kein Runden!)

  (Falsches) Ergebnis: 17

Typkonvertierungen – type casting

- Problemlos sind Typkonvertierungen vom kleineren zum größeren Typ
  - sie geschehen automatisch (implicit typecast)
- Muss von einem größeren Typ in einen kleineren konvertiert werden, muss man explizit casten
  - Operator (C++): `static_cast<zieltyp>(variable)`
  - Man muss sich bewusst sein, dass dabei Daten verloren gehen können!
  - Beispiel:

```
double i = 55.93;
int s = static_cast<int>(i);
std::cout << s;
```

  - Ergebnis: 55

Sichtbarkeiten in Klassen

- Klassen kapseln Daten und Funktionen
- Viele Daten sollen nur von der Klasse manipuliert werden dürfen, in der sie deklariert wurden
- C++ erlaubt es, für Daten und Funktionen Sichtbarkeiten zu definieren

  `public` → **öffentlich, sichtbar auch außerhalb der Klasse**

  `private` → **privat, sichtbar nur innerhalb der Klasse**

  (protected → Sichtbar nur für Kindklassen (Nur bei Vererbung))
- Sinn dahinter: Eigene Klassen vor dem Zugriff von außen schützen

Sichtbarkeiten in Klassen

```
//Datei: Rechteck.h

class Rechteck
{

public:
 Rechteck(float fL, float fB);
 ~Rechteck();

 float berechneFlaeche();
 float berechneUmfang();

private:
 float m_fLaenge;
 float m_fBreite;
};
```

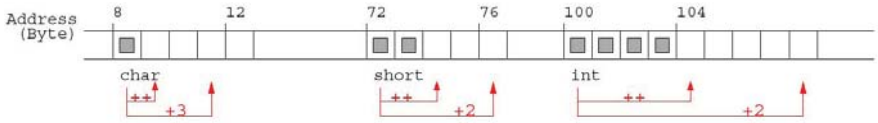
```
//Datei: main.cpp

int main(int argc, char* argv[])
{
 Rechteck r = new Rechteck(..);

 //nicht erlaubt, da privates
 //Attribut
 //→ programm kompiliert nicht
 float l = r->m_fLaenge;

 //erlaubt, da public
 float f = r->berechneFlaeche();

 std::cout << f;
 return 0;
}
```


Zeiger-Arithmetik

- Arithmetische Ganzzahl-Operationen wie **++, --** oder **+2** können auf Zeiger angewandt werden
- Die Umsetzung dieser Operationen unterscheidet sich jedoch von „normaler“ Arithmetik
- Zeiger-Arithmetik basiert auf der Größe des adressierten Typs in Byte
  → **char = 1 Byte, short = 2 Byte, int = 4 Byte, ...**

→ Ein Inkrement um 1 erhöht die Adresse um die Anzahl Bytes des Typs

Umsetzung im Speicher:



Rückblick: Arrays

- Arrays liegen im Speicher „en bloc“ vor.
  Beispiel **int**-Array
  int expOf2[] = {1, 2, 4, 8, 16, 32, 64};

- Indexgrenzen **[0]** und **[count-1]** müssen unbedingt eingehalten werden, sonst findet ein *Speicherzugriffsfehler* statt.
- expOf2[7] ist nicht zulässig, wird aber nicht vom Compiler abgefangen, sondern führt zu einem Laufzeitfehler!

Zeiger-Arithmetik

- Direkte Operationen auf Zeigern (nicht auf deren Inhalten!)
- Beispiel: Alle Elemente eines Arrays auf 0 setzen

Bisher haben wir das so gelöst:

```
short buffer[100];
int i;

for(i = 0; i < 100; i++)
{
   buffer[i] = 0;
}
```

Zeiger-Arithmetik

- Direkte Operationen auf Zeigern (nicht auf deren Inhalten!)
- Beispiel: Alle Elemente eines Arrays auf 0 setzen

Es geht auch mit Zeigern:

```
short buffer[100];
int i;
short *p;          //Pointer vom Typ der Arrayelemente (short)

p = buffer;        //Pointer auf das erste Arrayelement setzen
for( i = 0; i < 100; i++)
{
    *(p++) = 0;    //den Inhalt an der Stelle p auf 0 setzen
                   //p um eine Stelle weiterschieben
}
```


Rückblick: Arrays

- Arrays liegen im Speicher „en bloc“ vor.
  Beispiel int-array
  `int expOf2[] = {1, 2, 4, 8, 16, 32, 64};`

- Eine Array-Variable ist ein Zeiger auf das erste Array-Element!

```
double array[] = {0, 1, 2, 3, 4, 5, 6};
double* p_array_1 = &(array[0]);   // Zeiger auf erstes Element
double* p_array_2 = array;         // Zeiger auf erstes Element
if ( p_array_1 == p_array_2 ) {    // die Bedingung ist true
...
}
```


Noch ein Beispiel

```
// Array mit 6 Geschwindigkeiten
double speedValue[6] = {24.1, 28.1, 23.4, 19.5, 27.1, 24.8};
double* p_speed = speedValue; // Zeiger auf erstes Element
// Ausgabe der Geschwindigkeiten
for ( int i = 0; i < 6; i++, p_speed++ ) {
   std::cout << "Geschwindigkeit " << i << ": " << *p_speed <<
   std:endl;
}
```

→ Der Ope
den Zug

Achtung: **Klammerung!**
Was würde `std::cout << *p_speed + 3 << std::endl;` ausgeben?

```
std::cout << p_s    d[3] << std::endl;
std::cout << *(p_speed + 3) << std::endl;
```


Namespaces - Namensräume

- Ausschluss von Konflikten zwischen gleich benannten Objekten
- Gruppierung von logisch zusammenhängenden Funktionen
- Definition eines Namespaces über das Codewort “namespace“:

```
namespace HausA
{
   char* adresse = ”Zum weiten Feld 7”;
};
```

- Mit Hilfe der “using“-Anweisung übernimmt der Compiler den angegebenen Namensraum in den aktuellen Namensraum

```
using namespace HausA;
```

Namespaces - Namensräume


```
namespace HausA
{
   char* adresse = "Zum weiten Feld 7";
};

namespace HausB
{
   char* adresse = "Zum langen Feld 13a";
};

using namespace std;

int main()
{
   cout << HausA::adresse;
   cout << HausB::adresse;
   return 0;
}
```

Mit Hilfe der unterschiedlichen Namespaces können beide Variablen "adresse" benutzt werden.