Hochschule Esslingen
University of Applied Sciences

Objektorientierte Programmierung mit C++:

Änderungen und Ergänzungen gegenüber C

A. Freymann, Profs. M. Dausmann, D. Schoop, A. Rößler

Fakultät Informationstechnik, Hochschule Esslingen

Agenda

- Typkonzept
  - Typkonvertierung
  - Aufzählungstypen
  - Null-Pointer
- Funktionen in C++
  - Defaultparameter
  - Overloading
- Referenzkonzept
- Neue Schleifenform: for each-Schleife
- Speicherverwaltung mit **`new`** und **`delete`**

Typkonzept – Typkonvertierung

- Typkonvertierung
  - **`(Typname) Ausdruck`** (**alt**, cast-Notation)
  - **`Typname (Ausdruck)`** (**neue**, funktionale Notation)
  - In C++ sind beide Schreibweisen möglich.
- Beispiel: Konvertierung Integerzahl 10 in reelle Zahl 10.0

```
float zahl = (float) 10; (alt)
float zahl = float (10); (neu)
```

- Was tun bei komplexeren Typnamen?

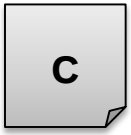
```
unsigned long long value = (unsigned long long) 1234.3;
typedef unsigned long long uul;
uul value = uul (1234.3);
```

TypeCasting.cpp

Typkonzept – Aufzählungstypen

- Aufzählungstypen gibt es weiterhin in einer Form wie in C, aber sicherer:

```
enum ECOLOR { RED, GREEN, BLUE };
enum ECOLOR x;
  // Schlüsselwort enum in C notwendig
   // in C++ nicht mehr
x = GREEN;
x = RED;
x = 1; // in C korrekt, da Aufzählungstypen
       // identisch zu int sind
// in C++ falsch, da Aufzählungstypen
// eigener Typ; Typkonvertierung notwendig
```



Typkonzept – Aufzählungskonstanten

- C++ hat zusätzlich eine neue Form von Aufzählungstypen.
- Mit diesen sind die Aufzählungskonstanten an den Typ gebunden.
- Name der Aufzählungskonstanten kann mehrfach verwendet werden.
- Aber nun muss der Typname mit dem Scope-Operator :: vor eine Aufzählungskonstante gestellt werden.

Typkonzept – Beispiel für Aufzählungstypen

```
enum class Kartenfarbe {KREUZ, PIK, HERZ, KARO};
enum class Organ {LEBER, HERZ, NIERE, LUNGE};

// Organ krankesOrgan = HERZ; geht so nicht mehr
// sondern immer mit Scope-Operator:
Organ krankesOrgan = Organ::HERZ;

Kartenfarbe trumpf = Kartenfarbe::HERZ;
```

Typkonzept – Null-Pointer

- In C gibt es den Wert `NULL`, eingeführt durch

```
#define NULL 0
```

- Damit ist nicht klar, ob `NULL` ein int-Wert oder ein Wert für einen Pointer ist.
- C++ definiert jetzt einen eindeutigen Wert für einen Pointer, der den Wert 0 haben soll: `nullptr`.
- `nullptr` kann man für alle Typen von Zeigern verwenden.
- Beispiel:

```
int * ptr = nullptr;
```

Defaultwerte für Funktionsparameter

```
struct point {
     float x;
     float y;
};

point makepoint (float _x = 0.0, float _y = 0.0)
{
   point temp;
   temp.x = _x;
   temp.y = _y;
   return temp;
}
```

Regeln für Defaultparameter

- Wenn ein Parameter mit Defaultwert angegeben wird, müssen alle folgenden Parameter auch einen Defaultwert besitzen.
- Wird bei einem Funktionsaufruf ein Parameter weggelassen, so müssen auch die folgenden weggelassen werden.

Beispiele für Defaultparameter

```
float dummy_func1 (int x, int y = 0, int z) // falsch
{ … }
float dummy_func2 (float x = 7.5, char* s)  // falsch
{ … }
float dummy_func3 (int x, float y = 7.5, float z = 0)
{ … }

a = dummy_func3(1)
a = dummy_func3(1.5, 1) // implizite Typkonvertierung
a = dummy_func3(1, 5)
a = dummy_func3(1, 1.5, 2)
```

Overloading von Funktionen

```
// Overloading = Mehrfachbelegung von Funktionsnamen
char max(char a, char b){
  return ( a >= b )? a : b; }

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

double max(double a, double b){
  return ( a >= b )? a : b; }

max('1', '1');
max(1, 2);
max(1.5, 2.1);
```

Regeln für Overloading

- Die Signatur der Funktion muss verschieden sein, d.h. gleicher Name aber verschiedene Parameterliste. (*Signatur = Funktionsname + Parameterliste*, *Parameterliste = Anzahl, Reihenfolge und Typen*)
- Der Rückgabetyp spielt keine Rolle. D. h. es gibt keine Überladung mit gleicher Signatur und verschiedenen Rückgabetypen.

Overloading versus Defaultparameter

- Overloading verwenden, bei unterschiedlichen Parameter***typen*** und gleicher Funktionalität.
- Defaultparameter verwenden, bei unterschiedlicher Parameter***anzahl*** und gleicher Funktionalität (gleicher Funktionsrumpf).

Referenzkonzept
& ist Referenz- und Adressoperator

```
int i = 5;

int& r = i;             // Referenzen
r = 5;                  // Referenzen

int* p = &i;            // Pointer
*p = 5;                 // Pointer

int* q = &r;            // Pointer

// Unterschied zwischen p und q?
```

ReferencesVSPointer

Call by Reference in C++

- In C (und C++) ist *call by reference* über Zeiger möglich.
- In C++ ist *call by reference* auch durch Referenztyp möglich.

```
void incr(int & zahl) {
  zahl++;
}

void main() {
  int x = 1;
  incr( x );    // int & zahl = x;
}               // d.h. zahl referenziert x
```

Keine implizite Typumwandlung bei Call-By-Reference

```
void incr(int & zahl) {// Call-by-Reference
   ++zahl;
}

int main()
{
   int ganzzahl = 10;
   float reellezahl = float(ganzzahl);
   // incr(reellezahl);                  // geht nicht
   int zurueck = int(reellezahl);
   incr(zurueck);
   cout << ganzzahl << " " << reellezahl;
   return 0;
}
```

Referenzen als Rückgabewert

```
int a[3] = {10, 11, 12};

int& element (int index) {
  return a[index];
}

int main(){

  element( 0 ) = 11; // int &element= a[0] = 11
  element( 2 ) = 22; // int &element= a[2] = 22
  a[1] = element(0) = 33; // a[0] = 33,a[1] = 33

  return 0;
}
```

Unterschiede: Pointer und Referenzen

- Referenzen sind vergleichbar mit konstanten Pointern:
  - Referenzen werden bei der Deklaration fest mit einem Objekt verbunden und können dann nicht mehr geändert werden, um mit einem anderen Objekt verbunden zu werden.
- Deshalb:
  - Referenzen können keine (Pointer-)Arithmetik.
  - Referenzen können nicht auf „NULL“ gesetzt werden.

Neue Schleifenform: for each-Schleife

- Die for each-Schleife (auch *range-based loop* genannt) ist neu in C++ (C++11).
- Man muss sich fast um nichts mehr kümmern.
- Die Schleife läuft immer automatisch vom Anfang bis zum Ende aller Daten.
- Beispiel:

```
int myArray[] = {9,8,7,6,5,4,3,2,1,0};
for (int value : myArray) {
   // value nimmt in jedem Schleifendurchlauf
   // einen Wert des Arrays an
   cout << value << endl; }
```

Neue Schleifenform: for each-Schleife

- Möchte man die Elemente eines statischen Arrays in der Schleife ändern, so kann man mit Referenzen arbeiten.
- Beispiel:

```
int myArray[] = {9,8,7,6,5,4,3,2,1,0};
for (int & element : myArray) { element++; }
for (int value : myArray) {
  cout << value << " ";
}
// Ausgabe: 10 9 8 7 6 5 4 3 2 1
```

- Die for each-Schleife muss Anfang und Ende der Daten kennen und funktioniert daher nicht mit dynamischen Arrays (z.B. int * arr = new int[5]).

Speicherverwaltung mit *new* und *delete*

- In C: **`malloc()`** und **`free()`**

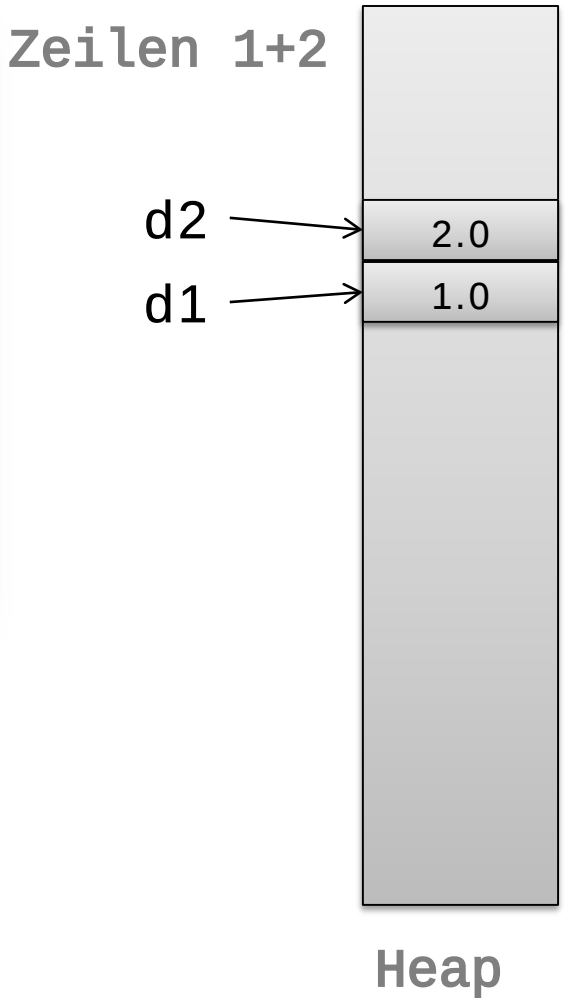
- In C++: **`new`** und **`delete`**
- **`new`** erzeugt Objekt **vom angegebenen Typ** und liefert Zeiger auf Objekt zurück.
- **`new`** berechnet die Speichergröße.
- Initialisierung mit Klammern () hinter **`new`** möglich.
- Erzeugtes Objekt kann mit **`delete`** wieder gelöscht werden.

Verwendung von new und delete

```
int * int_objekt = new int (5);

*int_objekt = 2;
delete int_objekt;

int * array = new int [7];
array[0] = 3;
delete [] array;
```

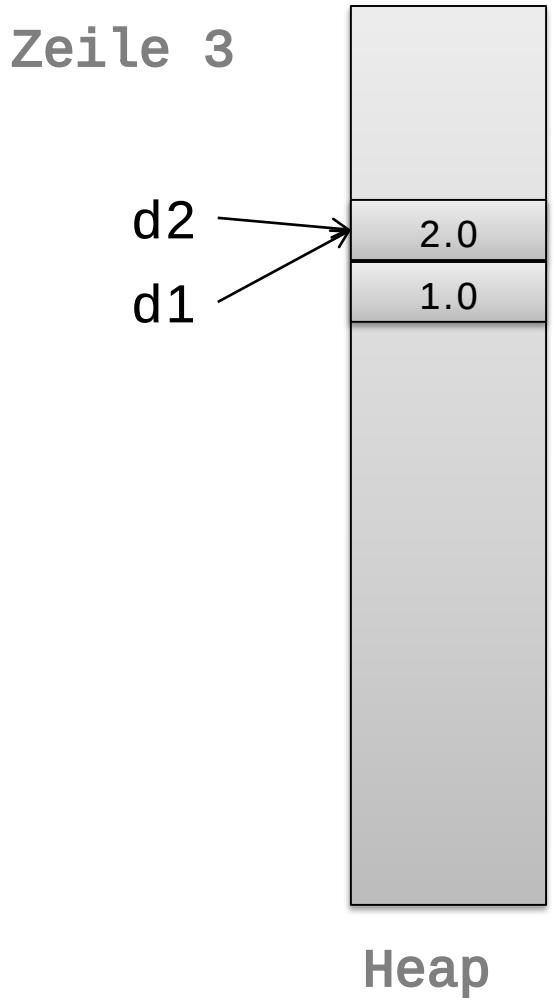
Beispiel mit new und delete

```
1  double* d1 = new double(1.0);
2  double* d2 = new double(2.0);
3  d1 = d2;

4  double* d3 = new double(3.0);
5  delete d1;
```



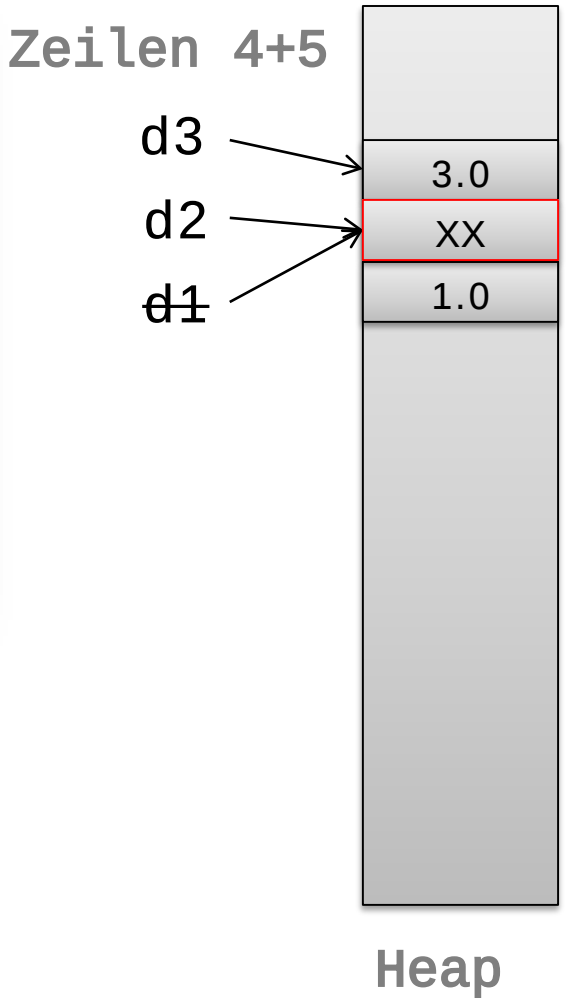
Beispiel mit new und delete

```
1  double* d1 = new double(1.0);
2  double* d2 = new double(2.0);
3  d1 = d2;

4  double* d3 = new double(3.0);
5  delete d1;
```



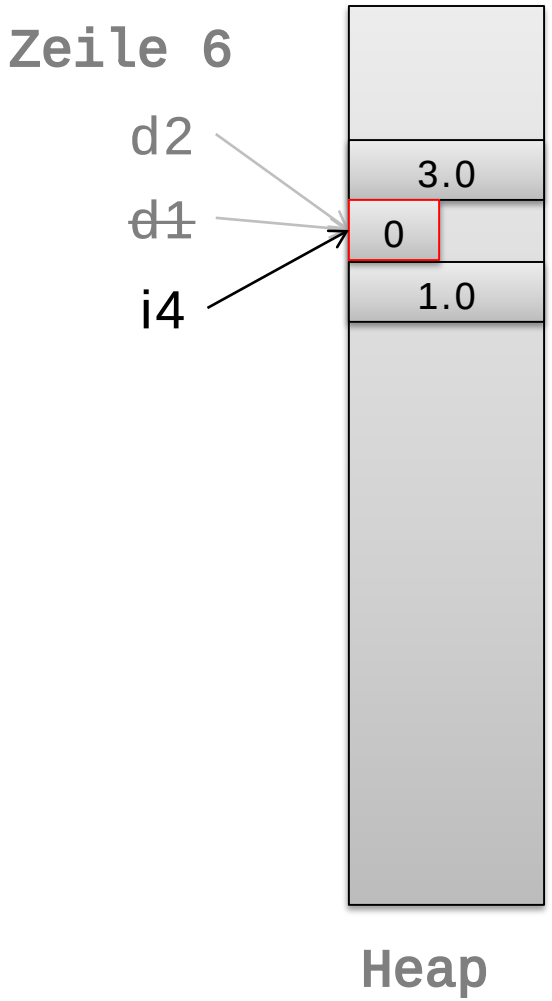
Beispiel mit new und delete

```
1  double* d1 = new double(1.0);
2  double* d2 = new double(2.0);
3  d1 = d2;

4  double* d3 = new double(3.0);
5  delete d1;
```



Beispiel mit new und delete

```
1  double* d1 = new double(1.0);
2  double* d2 = new double(2.0);
3  d1 = d2;

4  double* d3 = new double(3.0);
5  delete d1;

6  int* i4 = new int(0);
```



Zusammenfassung

- Casts sind auch in funktionaler Schreibweise möglich.
- Enums sind sicherer als in C.
- Enum-Klassen bieten eigenen Namensraum.
- Konstante für den Null-Pointer: `nullptr`.
- Defaultwerte für Parameter
- Overloading von Funktionsnamen
- Referenzen erlauben einfacheres Call-by-Reference.
- for each-Schleife (*range-based loop*)
- Anlegen von Objekten im Heap mit `new` und das Freigeben ihres Speicherplatzes mit `delete`