SAARLAND UNIVERSITY
COMPUTER SCIENCE

C/C++-Programmierung

new/delete, virtual, Typumwandlungen

Sebastian Hack
Christoph Mallon

(hack|mallon)@cs.uni-sb.de

Fachbereich Informatik
Universität des Saarlandes

Wintersemester 2009/2010

new/delete


```
X* x = new X(23);
delete x;

X* x = new X[n](); // Nur Standardkonstruktor moeglich
delete [] x;
```

- `new` besorgt Speicher und ruft Konstruktor des Objekts auf
- `delete` ruft Destruktor auf und gibt Speicher frei
- Nicht mit `malloc()`/`free()` mischen!
  $\longrightarrow$ `free(new X);` undefiniert
- Analog `new []` und `delete []` für Reihungen
  - Konstruktionsreihenfolge: $0, 1, \ldots, n-1$
  - Destruktionsreihenfolge: $n-1, n-2, \ldots, 0$
  - Reihenfolge wichtig bei Ausnahmebehandlung
- `new []` nicht mit `delete` (oder umgekehrt) mischen!
  $\longrightarrow$ `delete new X[10];` undefiniert
- Bei Fehlschlag wirft `new` eine `bad_alloc`-Ausnahme
  $\longrightarrow$ gibt niemals Nullzeiger zurück

new und native Datentypen

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
new int;         // uninitialisiert
new int();       // 0
new int[10];     // uninitialisiert
new int[10](); // alle 10 mit 0 initialisiert
```

- Normalerweise zum Aufruf des Standardkonstruktors Angabe von `()` redundant
- Außer bei nativen Datentypen (Zahlen, Zeiger)

virtual


```
struct X      { void f() { cout << "X::f()\n"; } };
struct Y : X { void f() { cout << "Y::f()\n"; } };

int main() {
  X* x = new Y();
  x->f(); // Ausgabe: X::f()
}
```

- Normalfall: Methoden sind nicht virtuell
- Modifikator `virtual`
- C++-Konzept: You don't pay for what you don't use

```
struct X      { virtual void f() { cout << "X::f()\n"; } };
struct Y : X { virtual void f() { cout << "Y::f()\n"; } };

int main() {
  X* x = new Y();
  x->f(); // Ausgabe: Y::f()
}
```

Virtuelle Destruktoren


```
struct A      { ~A() { cout << "A::~A()\n"; } };
struct B : A { ~B() { cout << "B::~B()\n"; } };

void f() {
  A* a = new B;
  delete a; // undefiniert!
}
```

- Destruktoren sind im Normalfall ebenfalls nicht virtuell
- Ausgabe oben: nur `A::~A()`
- Wenn der dynamische Typ nicht mit statischem übereinstimmt ist das Verhalten undefiniert!
- Destruktor mit `virtual` markieren, dann Ausgabe:

```
B::~B()
A::~A()
```

Abstrakte Methoden


```
struct A      { virtual void f() = 0;            };
struct B : A { virtual void f() { /* ... */ } };

void f() {
  A a; // Fehler! Klasse hat abstrakte Methode
  B b; // OK
}
```

- Virtuelle Methode ohne Implementierung in dieser Klasse
- Klasse mit abstrakten Methoden kann nicht instanziiert werden
- Erst Unterklassen, die alle abstrakten Methoden implementieren, können instanziiert werden

Typumwandlungen

- C bietet nur (Typ)x
- Erfüllt verschiedene Funktionen
  - `int` nach `float`
  - `void*` nach `int*`
  - `int const*` nach `int*`
  - Zahl nach Zeiger
- C++ bietet feinere Unterscheidung
  - `const_cast`
  - `static_cast`
  - `dynamic_cast`
  - `reinterpret_cast`
- Idee: Mehr Sicherheit bieten, Typumwandlungen im Quelltext stärker hervorheben

const_cast

- Erlaubt nur hinzufügen/entfernen von `const`
- Hinzufügen sicher und redundant
- Entfernen eventuell unsicher
  → Wenn verwiesenes Objekt nicht veränderbar, dann Verhalten undefiniert

```
void f(int* x)
{
  int const* a = const_cast<int const*>(x);    // Sicher
  int*       b = const_cast<int*>(a);          // Erlaubt
  float*     c = const_cast<float const*>(x); // Fehler
}
```

static_cast

- Umwandlung zwischen arithmetischen Typen (`int`, `float`, ...)
- Umwandlung von Unterklasse in Oberklasse (sicher, redundant)
- Umwandlung von Oberklasse in Unterklasse
  → undefiniert, wenn Oberklassenzeiger/-referenz nicht auf Objekt der Unterklasse zeigt
- Umwandlung von `void*` in andere Zeigertypen
  → undefiniert, wenn verwiesenes Objekt nicht von diesem Typ

```
struct A     {};
struct B : A {};
struct C : B {};
struct D     {};

void f() {
  B b;
  A& a = static_cast<A&>(b); // Sicher
  C& c = static_cast<C&>(b); // Undefiniert!
  D& d = static_cast<D&>(b); // Fehler
}
```

dynamic_cast

- Umwandlung von Unterklasse in Oberklasse (sicher, redundant)
- Umwandlung von Oberklasse in Unterklasse
- Wird zur Laufzeit geprüft, bei Fehlschlag:
  - Zeigertyp: Nullzeiger
  - Referenztyp: Ausnahme (`bad_cast`)
- Typ muss polymorph sein, d.h. muss eine virtuelle Methode besitzen

```
struct D     {};
struct A : D { virtual ~A() {} };
struct B : A { virtual ~B() {} };
struct C : B { virtual ~C() {} };

void f() {
  B b;
  A* a = dynamic_cast<A*>(&b); // Redundant
  C* c = dynamic_cast<C*>(&b); // Nullzeiger
  D* x = &b;
  B* y = dynamic_cast<B*>(x); // Fehler: D nicht polymorph
}
```

reinterpret_cast

- „Sonstige“ Umwandlungen
- Bitmuster des Operanden wird im Kontext des Zieltypen interpretiert
- Ganzzahl nach Zeiger
  $\longrightarrow$ Z.B. benötigt für Hardwareansteuerung: Register der Karte befindet sich an Adresse 0x1234
- Manches immernoch nicht erlaubt, z.B. float nach Zeiger (auch mit C-Cast nicht möglich)