Programmierparadigmen

Kapitel 2b
Objektorientierung am Beispiel C++

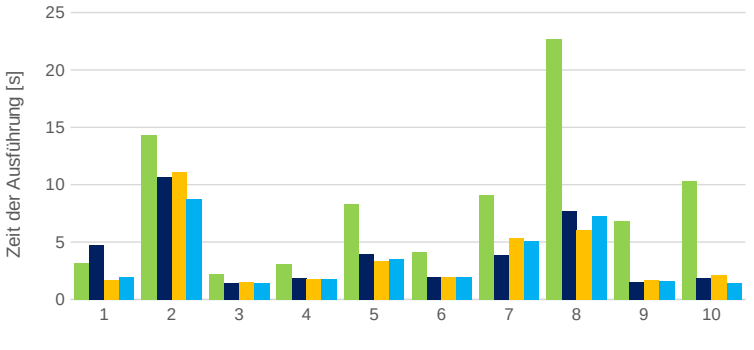
Java reicht nicht immer...

- Java ist eine relativ einfache Programmiersprache
  - Wenige Sprachkonstrukte
  - Mit Absicht kein direkter Speicherzugriff
  - Keine manuelle Speicherverwaltung
    - Freigabe *automatisch* wenn letzte Referenz zerstört wird; asynchron durch Garbage Collection; nicht-trivial! z.B. bei zyklischen Referenzen
  - Automatische Initialisierung von Variablen

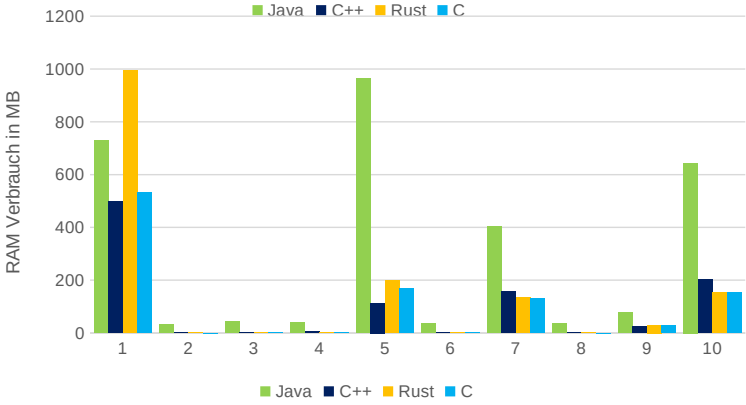
  → Geschwindigkeitsnachteile, Zeitverhalten schlechter vorhersehbar
- Übersetzt zu Byte-Code, nicht Maschinencode
  → Hardware-Unabhängigkeit
  - Führt zu langsamen Programmstart
  - Höherem Speicherverbrauch (für „Just in Time Compilation, JIT“)
  - Eigentlich Chance: man kann mit JIT auf spezielle HW optimieren, auf Workload basiert
- Wie leistungsfähig ist das im Vergleich zu anderen Ansätzen?

Java reicht nicht immer...

- Geschwindigkeit natürlich abhängig von Szenario
- Aus „Computer Language Benchmarks Game“



- Andere Sprachen an dieser Stelle signifikant schneller
- Andere Sprachen an dieser Stelle signifikant weniger Speicher → Caching effizienter!



Alternative?!

- Wenn es um Geschwindigkeit und direkten Speicherzugriff geht:
- Assembler?
  - So übersichtlich wie eine Registermaschine!?
  - Auf modernen Prozessoren nicht einfach effizient zu bekommen, weil Befehlssätze und Rahmenbedingungen zu kompliziert
- C?
  - Nur sehr wenige Instruktionen
  - Quasi: High-level Assembler ;)

  → Geringe Ausdrucksstärke

  → Relativ “gefährlich“

→ Wir brauchen manchmal etwas, dass uns die volle Kontrolle gibt, aber die Mächtigkeit von Java (und mehr)

Einführung C++

- Ziel von C++: volle Kontrolle über Speicher & Ausführungsreihenfolgen sowie skalierbarere Projekt-Größe
- Kompiliert zu nativem Maschinencode und erlaubt genauere Aussagen über Speicher-, Cache- und Echtzeitverhalten

- Viele Hochsprachenelemente
  - Wie Java objektorientiert; sogar ähnliche Syntax an viele Stellen (weil Java ursprünglich an C++ angelehnt)
- Jedoch kompromissloser Fokus Ausführungsgeschwindigkeit, d.h.
  - → Keine automatische Speicherverwaltung
  - → Keine Initialisierung von Variablen (im Allgemeinen)
  - → Kein Speicherschutz!
  - → Dinge, die Zeit kosten, müssen im Allgemeinen erst durch Schlüsselworte aktiviert werden

Einführung C++ – C++ und C

- C++ ist zu sehr großen Teilen eine Obermenge von C
  - Fügt Objektorientierung hinzu
  - Versucht fehleranfällige Konstrukte zu kapseln
  - Führt (viele) weitere Sprachkonstrukte ein, die Code kompakter werden lassen

"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off. "

Bjarne Stroustrup

- Kleine Warnung: Man kann C++ wie “C mit Klassen” benutzen, aber dann verliert man viele Fähigkeiten der Sprache. Leider ist viel “C++”-Code im Internet eher C-Code.
- Weitere Warnung: Es gibt unterschiedliche Versionen, die folgenden Beispiele funktionieren mit Version ≥ C++11

Überblick – Einführung in C++

- **Vergleich mit Java**
- Speichermanagement
- Vererbung
- Mehrfachvererbung
- Operator-Overloading
- Templates
- Container
- Shared Pointer

Einführung C++ – Vergleich mit Java (I)

```
[Hello.java]

// say that we are part of a package
// named hello:
package hello;

// declare a class called Hello:
public class Hello {
  // declare the function main
  // that takes an array of Strings:
  public static void main(String
                       args[]) {
    // call the static method
    // println on class System.out
    // with parameter "Hi Welt!":
    System.out.println("Hi Welt!");
  }
} // end of class Hello
```

```
[Hello.cpp]

// include declarations for
// I/O library where cout object
// is specified in namespace std::
#include <iostream>

// declare the function main that
// takes an int and array of
// strings  and returns an int
// as the exit code
int main(int argc, char* argv[]) {
  // stream string to cout object
  // flush line with endl
  std::cout << "Hello world!"
            << std::endl;
  return 0;
} // end of main()
```


Einführung C++ – Vergleich mit Java (II)

- Unterschiede im Aufbau:
  - C++ hat globale Funktionen, also außerhalb von Klassen, wie `main`
  - `#include` gibt Dateien mit Klassen- und Funktionsdefinitionen an, die der Compiler einlesen soll
  - Java-Programme werden in *package*s gegliedert, in C++ gibt es mit *module*s ein ähnliches Konzept, welches aber (noch) nicht verbreitet ist
  - C++-Programme können (ohne Bezug zu Dateien) in *namespace*s untergliedert werden, hier `std`
- Programmargumente:
  - In Java bekommt `main` ein String-Array übergeben, die Länge kann über `.length` abgefragt werden
  - C/C++-Programme erhalten ein Array von `char*` (Details zu Pointern folgen)
  - In C/C++ sind Arrays keine Pseudoobjekte, sondern Speicherbereiche in denen die Daten konsekutiv abgelegt sind
  - → `argc` wird benötigt die Anzahl an Elementen zu kodieren

Einführung C++ – Vergleich mit Java (III)

- Rückgabewerte:
  - In Java keine Rückgabe in der `main`-Methode
  - In C++ Rückgabe eines *exit code*
    - 0 gibt an: Programmausführung erfolgreich
    - Andere Werte geben einen Programm-spezifischen Fehlercode zurück
- Primitive Datentypen:
  - Wie in Java einfache Datentypen, die „Zahlen“ enthalten
  - `char`, `short`, `int`, `long` sind auf 64-bit Maschinen 8 bit, 16 bit, 32 bit und 64 bit breit (`char` braucht in Java 16 Bit!)
  - `long` ist auf 32 bit Maschinen 32 Bit breit, `long long` [sic!] ist immer 64 Bit
  - `bool` speichert Boolsche Werte (Breite hängt vom Compiler ab!)
  - Ein `unsigned` vor Ganzahltypen gibt an, dass keine negativen Zahlen in der Variable gespeichert werden (Beispiel: `unsigned int`) → Kann größere Zahlen speichern & zu viel Unsinn führen (beim Vergleich mit vorzeichenbehafteten Zahlen)

Einführung C++ – Beispiel für Namespaces

```
#include <iostream>

namespace my {
    void hi() { std::cout << "hi"; }
    namespace ho { // ein verschachtelter Namespace
                   // mit der gleichen Funktion nochmal!
        void hi() { std::cout << "ho"; }
    }
}
namespace n2 { // Neue Funktion cout() in einem anderen Namespace
    void cout() { std::cout << "!"; }
}

int main() {
    my::hi();
    my::ho::hi();
    n2::cout();   // Nach diesem Befehl insgesamt: hiho!
    return 0;
}
```


Einführung C++ – Klassen

- Header `Foo.hpp` deklariert Struktur und Schnittstelle

```
class Foo {      // deklariere Klasse Foo
public:          // Block ohne Zugriffsbeschränkung
  Foo();         // Konstruktor
  ~Foo();        // Destruktor
protected:       // Block von Dingen, auf die auch abgeleitete Klassen
                 // zugreifen dürfen
  int num;       // Member-Variable
};               // WICHTIGES Semikolon!
```

- Implementierung in getrennter Datei `Foo.cpp`:

```
#include "Foo.hpp"   // Klassen Deklaration einbinden
#include <iostream>  // Einbinden von Funktionen der stdlib
Foo::Foo() :         // Implementierung des Konstuktors von Foo
  num(5) {           // Statische Initialisierung von num
                     // Code in Klammern {} kann auch initialisieren
  std::cout << "c" << std::endl;
}
Foo::~Foo() {
  std::cout << "d" << std::endl;
}
```


Einführung C++ – Klassen (I)

- Reine Implementierung auch im Header möglich, aber Trennung von Implementierung und Deklaration erlaubt schnelleres Kompilieren
- Trennung nicht immer möglich (später mehr Details), aber im Allgemeinen zu bevorzugen

- Der scope-Operator :: wird zum Zugriff auf namespaces und zur Beschreibung der Klassenzugehörigkeit von Methoden verwendet

- Initialisierung von Variablen vor Funktionsrumpf etwas „merkwürdig“ zu lesen, aber erlaubt schnelle Implementierungen...
  - Syntax: nach Konstruktor `:` dann jeweils Variable`(`Wert`)`
  - Variablen durch `,` getrennt
  - Wichtig: Reihenfolge der Variablen wie in Deklaration der Klasse!

Einführung C++ – Klassen (II)

- Schlüsselworte `private`, `protected` und `public` vergleichbar zu Java, werden aber vor ganze Blöcke geschrieben
  - Kapselung nur auf Ebene von Klassen → Klassen sind immer `public`
  - `protected` erlaubt nur der Klasse selber und Unterklassen den Zugriff
- Zugriffe außerhalb der Klassenstruktur können durch `friend`-Deklaration erlaubt werden (teilweise verrufen!)

- Auch `final` ähnlich zu Java
  - Verhindert weiteres Ableiten von Klassen

- Schlüsselwort `const` markiert Methoden, die Objekte nicht verändern → Erlauben die Übergabe von Nur-Lesen-Referenzen

Einführung C++ – Klassen (III)

- Größere Unterschiede zu Java:
- Klassen können Destruktoren besitzen
  - Werden aufgerufen wenn Objekt zerstört wird
  - Kann bspw. dafür verwendet werden, um von dem Objekt allozierte Speicherbereiche freizugeben (Achtung: anschließend darf auf diese nicht mehr zugegriffen werden – problematisch wenn anderen Objekte diese Speicherbereiche bekannt gegeben wurden!)
  - Destruktor kann Zerstören eines Objekts aber nicht verhindern
  - Methodensignatur `~Klassenname()` – kein Rückgabetyp!
  - Warum gibt es das nicht in Java?

- Neben dem Standardkonstruktor oder einem expliziten Konstruktor existiert ein Copy-Constructor
  - Methodensignatur `Klassenname(const Klassenname& c)`
  - Wird aufgerufen wenn Objekt kopiert werden soll
  - Vergleichbar zu Object.clone() in Java

Einführung C++ – Überladen von Methoden

- Überladen von Methoden vergleichbar zu Java
- Parametertypen (oder const-Markierung) müssen sich unterscheiden!
- Nur Veränderung des Rückgabewertes nicht ausreichend
  - Wie in Java? Warum?

```
class Foo {
public:
  // zwei Methoden die gleich heißen!
  void doMagic(int i);
  void doMagic(std::string s);
};
```

- Standardparameter vermeiden Schreibarbeit:

```
class Foo {
public:
  void doMagic(int i = 0);
  void doMagic() { return doMagic(0); } // überflüssig
};
```


Einschub: C++-Präprozessor

- C/C++-Code kann vor dem Übersetzen durch einen Präprozessor verändert werden
- Alle Präprozessor-Makros beginnen mit #
- (Haupt-)gründe:
  - Importieren anderer Dateien
  - An- und Ausschalten von Features je nach Compile-Optionen
  - Kapselung von Plattform-spezifischem Code
  - Vermeiden von Redundanzen
- Makros sollten vermieden werden
  - Schwierig zu lesen
  - Keine Namespaces
  - Keine Typsicherheit
- Manchmal jedoch einzige Möglichkeit

Einschub: C++-Präprozessor

- Wichtige Makrobefehle:

```
#include "X.hpp" // Datei X.hpp aus Projekt-Ordner
#include <cstdio> // Datei cstdio aus System-Includes
```

```
#ifdef DEBUG // falls Konstante DEBUG definiert ist
std::cout << "Wichtige Debugausgabe" << std::endl;
#endif
```

```
#define DEBUG // Konstante setzen
#define VERSION 3.1415 // Konstante auf einen Wert setzen
#define DPRINT(X) std::cout << X << std::endl; // Macro-Fkt.
#undef DEBUG // Konstante löschen, good practice!
```

```
#ifndef __linux__ // falls nicht für Linux übersetzt
playMinesweeper();
#endif
```


Einschub: Include-Guards

- Eine (oft hässliche) Eigenschaft des #include-Befehls: kein Überprüfen ob eine Datei vorher bereits eingebunden wurde
- Problematisches Beispiel:
  `Bar.hpp`

```
#include "Foo.hpp"
...
```

  `Batz.hpp`

```
#include "Bar.hpp"
#include "Foo.hpp"
...
```

  → Fehler, weil Klasse Foo bereits deklariert wurde
- Common Practice: Include-Guards um alle Header-Dateien

```
#ifndef FOO_HPP
#define FOO_HPP
...
#endif
```


Überblick – Einführung in C++

- Vergleich mit Java
- **Speichermanagement**
- Vererbung
- Mehrfachvererbung
- Operator-Overloading
- Templates
- Container
- Shared Pointer

C++-Speichermanagement 101

- Bisher: Alle Programme ohne um Speicher zu kümmern?!

```
int main() {
    int var = 1;
    int var2 = 2;
}
```

... genau wie in Java?!

- Nicht ganz:

```
int main() {
    std::string s("hallo");
    std::cout << s;
    return 0;
}
```

Objekte können wie primitive Datentypen in Java ohne Allokation mit new verwendet werden (manchmal)

- Wo liegen die Unterschiede? → Brauchen mehr Details...

TELEMATIK Rechnernetze

C++-Speichermanagement 101

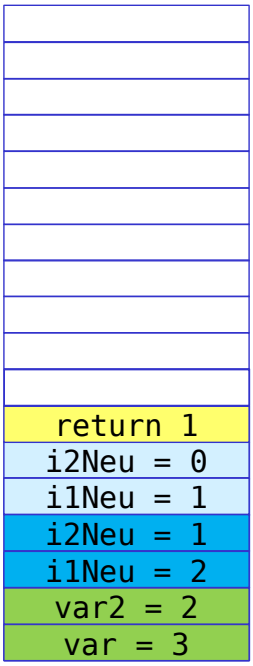
- Programmspeicher enthält Code und Daten, vom Betriebssystem i.A. auf virtuelle Adressbereiche abgebildet
- Unterschiedliche Varianten von Datenspeicher:
  - *Stack* hält alle Variablen einer Methode, aller aufrufenden Methoden, Parameter, Rückgabewerte und einige Management-Daten
  - *Heap* hält Variablen und Objekte, die nicht direkt über Methodenaufrufe übergeben werden
  - Speicher *für globale* und *statische* Objekte und Variablen

- Java legt primitive Datentypen im Stack ab und Objekte im Heap
- C++ kann sowohl primitive Datentypen als auch Objekte in Stack und Heap abbilden
- Für den Stack bieten Java und C++ automatisches Speicher-Mgmt.
- Für den Heap bietet nur Java automatisches Speicher-Mgmt.
- Warum der Unterschied? → Brauchen mehr Details...

C++-Speichermanagement 101

- Vereinfachter Ablauf der Stack-Nutzung:
- Bei Funktionsaufruf: Speicher auf Stack „gepackt“
- Bei Verlassen wieder entfernt

```
// Subtraktion über Dekrement
int sub(int i1, int i2) {
    if(i2 == 0)
      return i1;
    int i1Neu = i1 - 1;
    int i2Neu = i2 - 1;
    return sub(i1Neu, i2Neu);
}

int main() {
    int var = 3;
    int var2 = 2;
    return sub(var, var2);
}
```



TELEMATIK Rechnernetze

C++-Speichermanagement 101

Eigenschaften des Stack-Speichers:

- Variablen/Objekte haben klare Lebensdauer

→ Werden immer gelöscht wenn Funktion verlassen wird

→ Man kann Speicher nicht „aufheben“

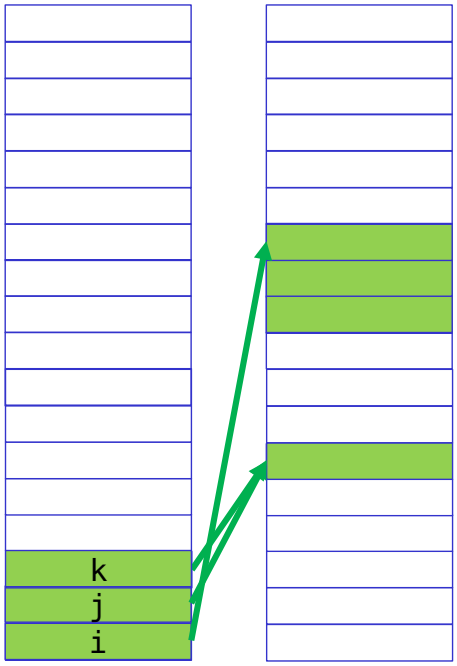
- In der Regel sehr schnell, weil im Prozessor-Cache
- In der Größe begrenzt, z.B. 8MB bei aktuellen Linux-Systemen

- Für flexiblere Speicherung brauchen wir anders organisierten Speicher...

C++-Speichermanagement 101

- Heap: Keine klare Struktur
- Anlegen: in C++ & Java mit new
- Um angelegten Speicher anzusprechen: Zeiger und Referenzen
- In Java automatisch Zeiger
- In C++ Zeiger durch * hinter Typ

```
int main() {
  int* i = new int[3];
  int* j = new int;
  int* k = j;
  delete [] i;
  delete j;
  return 0;
}
```



TELEMATIK Rechnernetze

C++-Speichermanagement 101

- Löschen von Heap-Speicher: Java automatisch
- Warum der Unterschied?
  - Nicht einfach festzustellen, wann letzter Zeiger auf Objekt gelöscht wurde
    - Zeiger können selbst auch im Heap gespeichert sein
    - Zyklische Referenzen!
  - Relativ aufwändiges Scannen, in Java durch regelmäßige Garbage Collection gelöst
    - Führt zu Jitter (Schwankung der Zeitdauer, die bestimmte Programmabschnitte zur Bearbeitung benötigen) & Speicher-Overhead, ...
- In C++ nur manuell
  - durch genau einen Aufruf von `delete`
  - Programmierer ist dafür verantwortlich, dass danach kein Zeiger auf diesen Speicher mehr benutzt wird
- Schwerer als es scheint!

C++-Speichermanagement 101

- Anlegen eines Objektes auf dem Heap:

```
std::string* s = new std::string("wiz!");
delete s;
```

- Allokation von Feldern:

```
int* i = new int[29]; // gültige Indicies 0-28
i[0] = 23;
delete [] i;          // nicht mit delete i; verwechseln!
```


TELEMATIK Rechnernetze

C++-Speichermanagement 101

- Zeiger können durch & auf beliebige Variablen ermittelt werden

```
int  i = 0;
int* j = &i;  // &-Operator erzeugt Zeiger
              // j darf nicht gelöscht werden
```

- Zeiger können durch * dereferenziert werden

```
int i = 0;
int* j = &i; // &-Operator erzeugt Zeiger
*j = 1;      // Zugriff auf Variableninhalt
```

- Zugriff auf Methoden/Member Variablen

```
std::string* s = new std::string("wiz");
(*s).push_back('?'); // manuelles Derefenzieren
s->push_back('?');   // -> Operator
delete s;
```


C++-Speichermanagement 101

- C++ übergibt alles als Kopie

```
void set(std::string s) { s = "foo"; }
int main() {
    std::string s = "bar";
    set(s);
    std::cout << s; // gibt bar aus
    return 0;
}
```

- Zeiger können verwendet werden, um schreibend zuzugreifen

```
void set(std::string* s) { *s = "foo"; }
int main() {
    std::string s = "bar";
    set(&s);
    std::cout << s; // gibt foo aus
    return 0;
}
```


C++-Speichermanagement 101

- Zeiger erlauben syntaktisch sehr viele Dinge mit unvorhersehbaren Nebenwirkungen

```
std::string* magicStr() {
    std::string s("wiz!");
    return &s;   // gibt Speicher auf Stack weiter
                 // Tun Sie das nie!
}
int main() {
    std::string* s = magicStr();
    std::cout << *s; // Stack ist bereits überschrieben!
    return 0;
}
```

- Ausgabe?
  - Korrumpierter Speicher
  - Absturz
  - Wir wissen es nicht...

C++-Speichermanagement 101

- Genauso gefährlich: Erzeugen eines Zeigers und vorzeitiges Löschen

```
std::string* magicStr() {
    std::string* s = new std::string("wiz!");
    delete s;  // löscht Speicher, s selbst unverändert!
    return s;
}
int main() {
    std::string* s = magicStr();
    std::cout << *s;
    return 0;
}
```

- Ausgabe?
  - Vermutlich: wiz!, weil Speicher noch nicht wieder verwendet wurde
  - Verlassen können wir uns darauf nicht!
  - Solche Bugs machen sich oft erst nach Wochen oder Jahren bemerkbar

TELEMATIK Rechnernetze

C++-Speichermanagement 101

- Warum wirken sich Speicherfehler so unvorhersehbar aus?
  - Speicherfehler entstehen sehr häufig durch Zugriff auf Speicherbereiche nachdem diese freigegeben worden sind
  - Ob hierdurch später ein Fehler auftritt, hängt davon ab wie der freigegebene Speicher nach der Freigabe wieder genutzt wird
  - Die insgesamte Speichernutzung wird durch die Gesamtheit aller Speicherallokationen und -freigaben beeinflusst
  - Das kann dazu führen, dass ein Speicherfehler in Modul X erst lange nach seinem Entstehen Auswirkungen zeigt, nachdem in einem anderen Modul Y eine Änderung eingeführt wurde
  - Auch eingebundene dynamische Bibliotheken haben Einfluss
  - Das macht es so schwierig, solche Fehler zu finden!
  - Es gibt aber mittlerweile Tool-Support hierfür (→ Übung)

C++-Speichermanagement 101

- Bessere Alternative: Referenzen
- Zeigen ebenfalls auf Speicher, Compiler stellt aber sicher, dass Speicher gültig ist (wenn man nicht in Zeiger wandelt etc.)!
- Markiert durch Suffix &
- Beispiel:

```
void set(std::string& s) { s = "foo"; }
int main() {
    std::string s = "bar";
    set(s);
    std::cout << s; // gibt foo aus
    return 0;
}
```

- Dereferenzierung durch * und -> nicht notwendig

C++-Speichermanagement 101

- → Referenzen sind toll, haben aber eine Einschränkung:

```
std::string& magicStr() {
  std::string s("wiz!");
  return s; //< FEHLER
}
```

```
std::string& magicStr() {
  static std::string s("wiz!");
  return s; // klappt prima
}
```

- Per Referenz übergebene Rückgabewerte müssen im Speicher noch existieren, wenn Methodenaufruf abgeschlossen ist...
  - OK für globale Variablen, Member-Variablen, statische Variablen...
  - Nicht-OK für Speicher, der wirklich dynamisch alloziert werden muss
- Allgemein bleiben nur Zeiger und Heap:

```
std::string* magicStr() {
  std::string* s = new std::string("wiz!");
  return s; // klappt prima, aber: aufpassen wann s gelöscht
            //  werden kann und vollständig vergessen wurde!
}
```


C++-Speichermanagement 101

- Konvertierung von Zeigern zu Referenzen mit „*“-Operator:

```
std::string& s = *magicStr(); // Konvertieren in Referenz
std::string s2 = *magicStr(); // Konvertieren in Referenz & Kopie!
```

- Konvertierung von Referenzen zu Zeigern mit „&“-Operator:

```
std::string s("bla");
std::string* sStar = &s; // Konvertieren in Zeiger
```

- Problem gesehen?
- Wir haben schon 2 mal `delete` vergessen!
- In einem Konstrukt können wir gar nicht löschen, in einem nicht ohne weiteres...

TELEMATIK Rechnernetze

C++-Speichermanagement 101

- Abschließende Bemerkungen zum Speicher
  - Niemals Speicher doppelt löschen – Niemals Löschen vergessen!
  - Häufige Praxis: Zeiger auf NULL setzen nach dem Löschen (Aber: gibt es danach wirklich keinen anderen Zeiger mehr?)
  - Nur Speicher löschen, der mit „new“ allokiert wurde
  - Speicher der mit „new“ allokiert wurde in jedem möglichen Programmablauf löschen (selbst wenn Exceptions auftreten)...
  - Nie über Feldgrenzen hinweg lesen/schreiben (auch negative Indizes!)
  - Programme ausgiebig testen (dabei Address Sanitizer aktivieren!)
  - Statische Code Analyse nutzen: z.B. http://cppcheck.sourceforge.net
  - `malloc`/`free` sind Äquivalente in Sprache C und nicht typsicher!

C++-Speichermanagement 101

- Verbreitetes Vorgehen in C++ (*Pattern*): Resource Acquisition Is Initialization (RAII)
- Speicher (oder Ressourcen im Allgemeinen) wird nur im Konstruktor einer Klasse reserviert
- Destruktor gibt Speicher frei
- Sicheres (Exceptions!), nachvollziehbares Konstrukt
- Beispiel:

```
class MagicString {
  std::string* s;
public:
  MagicString() : s(new std::string("wiz!")) {}
  std::string* magicStr() { return s; }
  ~MagicString() { delete s; }
};
```

- Funktioniert immer? Leider immer noch nicht...

TELEMATIK Rechnernetze

C++-Speichermanagement 101

- Kopieren von Objekten verursacht Probleme!

```
int main() {
  MagicString s1;
  MagicString s2(s1);
  return 0;
}
```

```
a.out(62025,0x10cad5dc0) malloc: *** error for object
0x7fcf33c01700: pointer being freed was not allocated
a.out(62025,0x10cad5dc0) malloc: *** set a breakpoint
in malloc_error_break to debug
Abort trap: 6
```

- Warum?
- Die Zeiger s der Objekte s1 und s2 zeigen nach Kopien auf gleichen Speicher ➞ dieser wird doppelt freigegeben

C++-Speichermanagement 101

- Der Default Copy Constructor ruft nicht den eigenen Constructor auf!
- Kopieren von Objekten muss daher ggf. explizit behandelt werden!

```
class MagicString {
  std::string* s;
public:
  MagicString() : s(new std::string("wiz!")) {}
  // Copy-Konstruktor wird beim Erzeugen einer Kopie aufgerufen
  MagicString(const MagicString& m) : s(new std::string(*m.s)) {}
  std::string* magicStr() { return s; }
  ~MagicString() { delete s; }
};
```

- Funktioniert in unserem Beispiel-Code, aber es gibt auch einen anderen Weg eine Kopie zu erzeugen, der eigentlich auch behandelt werden muss; dazu später mehr...
- Speicherverwaltung in std::string funktioniert intern genau so!

Überblick – Einführung in C++

- Vergleich mit Java
- Speichermanagement
- **Vererbung**
- Mehrfachvererbung
- Operator-Overloading
- Templates
- Container
- Shared Pointer

Wiederholung: Ziele von Vererbung

- Ziele von Vererbung:
  - Unterstützung typisch menschlicher Denkprozesse (Abstraktionsvermögen!)
  - Vermeiden von Mehrfachimplementierungen
  - Definierte Programmierschnittstellen durch Überschreiben von Methoden/abstrakte Methoden
  - Vermeiden von Dopplung interner Daten
    - Müssten für unterschiedliche Schnittstellen sonst immer neu zusammengestellt werden

Einführung C++ – Vererbung

- Vererbung syntaktisch ebenfalls ähnlich zu Java:

```
class Foo {
public:
  int magic()      const { return 23; }
  int enchanting() const { return 0xbeef; }
};

class FooBar : public Foo {
public:
  int magic()  const { return 42; }
};
```

- Ausgabe?

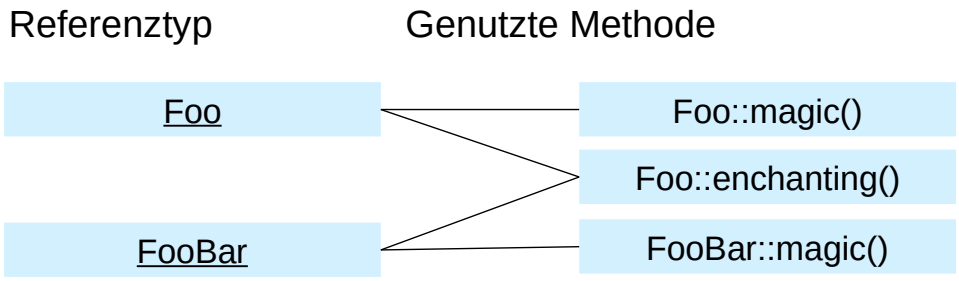
```
std::cout << FooBar().magic() << std::endl;
std::cout << FooBar().enchanting() << std::endl;
```

- Warum nur im Prinzip? – Funktioniert nicht immer richtig

```
void print(const Foo& f) {  // Übergabe als Referenz
  std::cout << f.magic() << std::endl;
}
print(FooBar()); // gibt 23 nicht 42 aus!
```


Einführung C++ – Vererbung

- Unterschied zu Java: Methoden „liegen“ bei C++ statisch im Speicher
- D.h. `f.magic();` ruft statisch `magic`-Methode in Klasse `Foo` auf, weil `f` eine Referenz vom Typ Foo ist

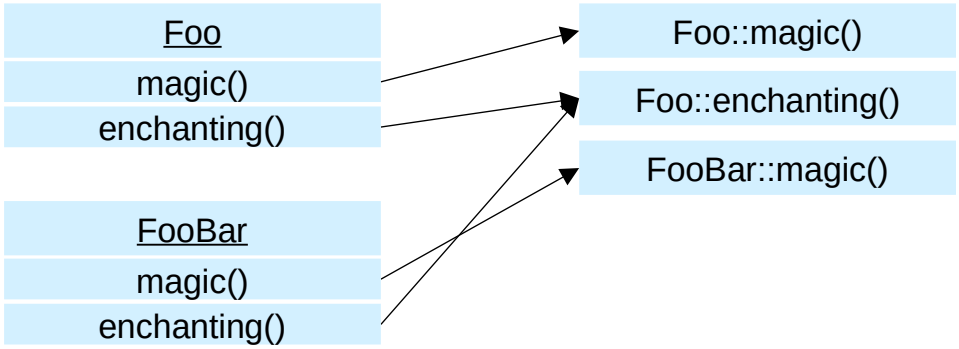


- Vermeidet Mehrfachimplementierungen, realisiert aber keine einheitliche Schnittstelle!
- Nach Überschreiben einer Methode wollen wir meist, dass genutzte Methode nicht vom Referenztyp abhängt, sondern vom Objekttyp

TELEMATIK Rechnernetze

Einführung C++ – Vererbung

- Java-ähnliches Verhalten kostet Zeit & Speicher → muss explizit aktiviert werden
- Idee zu jedem Objekt speichern wir Zeiger auf zu nutzende Methoden
- Tabelle wird `vtable` bezeichnet



- Markierung von Methoden, für die ein Zeiger vorgehalten wird, mit Schlüsselwort `virtual`
- Für `enchanting` eigentlich nicht benötigt!

Einführung C++ – Vererbung mit virtual

- Funktionierendes Beispiel:

```
class Foo {
public:
  virtual int magic() const { return 23; }
};
class FooBar : public Foo {
public:
  int magic() const override { return 42; }
};
int r(const Foo& f) { return f.magic(); }
int main() {
  return r(FooBar()); // yeah gibt 42 zurück!
}
```

- Aufruf von virtuellen und nicht-virtuellen Methoden transparent
- `virtual`-Markierung genügt in Oberklasse, alle abgeleiteten Methoden ebenfalls „virtuell“
- `override`-Markierung optional, aber hätte vor fehlendem `virtual` gewarnt!

TELEMATIK Rechnernetze

Einführung C++ – Vererbung mit virtual

- Nicht umsonst:

```
class Foo {
public:
  int magic()             const { return 23; }
};
class VFoo {
public:
  virtual int magic()    const { return 23; }
};
int main() {
  std::cout << sizeof(Foo) << std::endl;
  std::cout << sizeof(VFoo) << std::endl;
  return 0;
}
```

- Ausgabe:

```
1    // Adressen unterschiedlicher Objekte sollen unterschiedlich sein (kein 0)
8    // 8 Byte pro Funktionszeiger, die pro Objekt (nicht Klasse!) anfallen
```


Überblick – Einführung in C++

- Vergleich mit Java
- Speichermanagement
- Vererbung
- **Mehrfachvererbung**
- Operator-Overloading
- Templates
- Container
- Shared Pointer

Einführung C++ – Mehrfachvererbung

- C++ unterstützt keine Interfaces
- Aber C++ unterstützt Mehrfachvererbung!
  → Pro Interface eine Basisklasse mit abstrakten Methoden erstellen

```
class FooInterface { // sowas wie ein interface?!
public:
  virtual int magic() const = 0;  // abstrakte Methode
                                  // (da Zeiger auf 0 gesetzt)
};
class Bar {
public:
  int enchanting() const { return 0xbeef; }
};

class FooX : public FooInterface, public Bar {
public:
  int magic() const override { return 42; }
};
```


Einführung C++ – Mehrfachvererbung

- Mächtigeres Konstrukt, aber „With great power there must also come great responsibility”
- Was geschieht wenn mehrere Basisklassen gleiche Methoden überschreiben?

```
class Foo {
public:
  virtual int magic() const { return 23; }
};
class Bar {
public:
  virtual int magic() const { return 0xbeef; }
};
class FooBar : public Foo, public Bar {};
```

- Explizite Instanziierung nötig:

```
FooBar().magic();       // Fehler weil doppeldeutig...
FooBar().Foo::magic();  // gibt 23 zurück!
FooBar().Bar::magic();  // gibt 0xbeef zurück!
```


Einführung C++ – Mehrfachvererbung

- Gute Praxis: Explizites Überschreiben

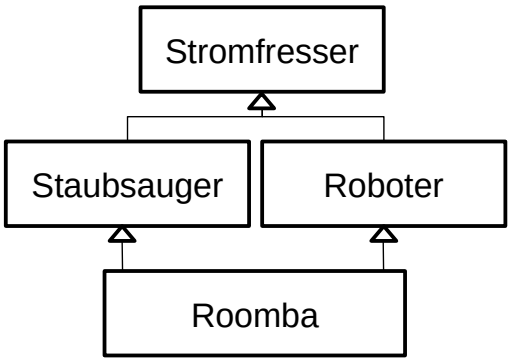
```
class NiceFooBar : public Foo, public Bar {
  // erlaube NiceFooBar().magic()
  int magic() const override { return Bar::magic(); }
};
```

- Wegen Mehrfachvererbung: kein `super::`
- Stattdessen immer `NameDerBasisKlasse::`

- Alles gelöst? Leider nein

Einführung C++ – Diamond Problem

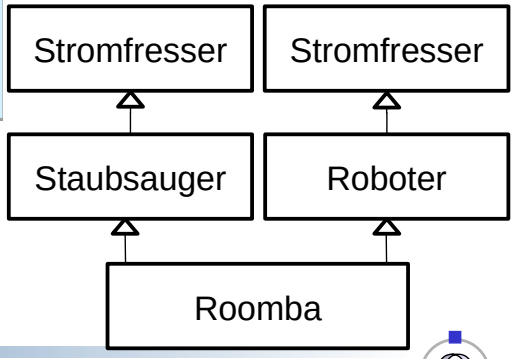
```
class Stromfresser {
public:
  Stromfresser() {
    std::cerr << "Mjam" << std::endl;
  }
};
class Roboter : public Stromfresser {};
class Staubsauger : public Stromfresser {};
class Roomba : public Staubsauger,
               public Roboter {};

int main() {
  Roomba q;
  return 0;
}
```



- Ausgabe?

```
Mjam
Mjam
```

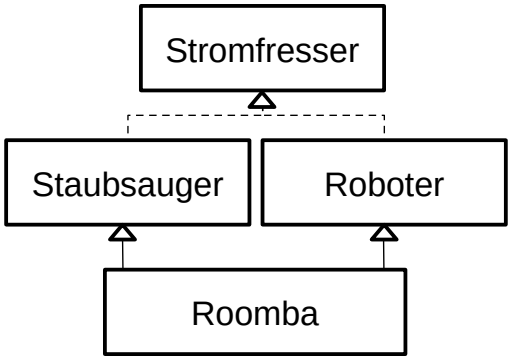
- Warum?



TELEMATIK Rechnernetze

Einführung C++ – Diamond Problem

- Staubsauger und Roboter instanziieren Basisklasse Stromfresser getrennt…
- Je nach Szenario, aber semantisches Problem!
- Auflösung zur Compile-Zeit unmöglich
- Auflösung zur Laufzeit über Zeiger möglich, kann aber zeitaufwändiger sein → nicht Standard!
- Markieren der Ableitung als `virtual` behebt das Problem…




```
class Roboter : virtual public Stromfresser {};
class Staubsauger : virtual public Stromfresser {};
```

- Eine mögliche Lösung des Problems… aber nicht immer im vorhinein klar wann Ableitung virtuell sein muss

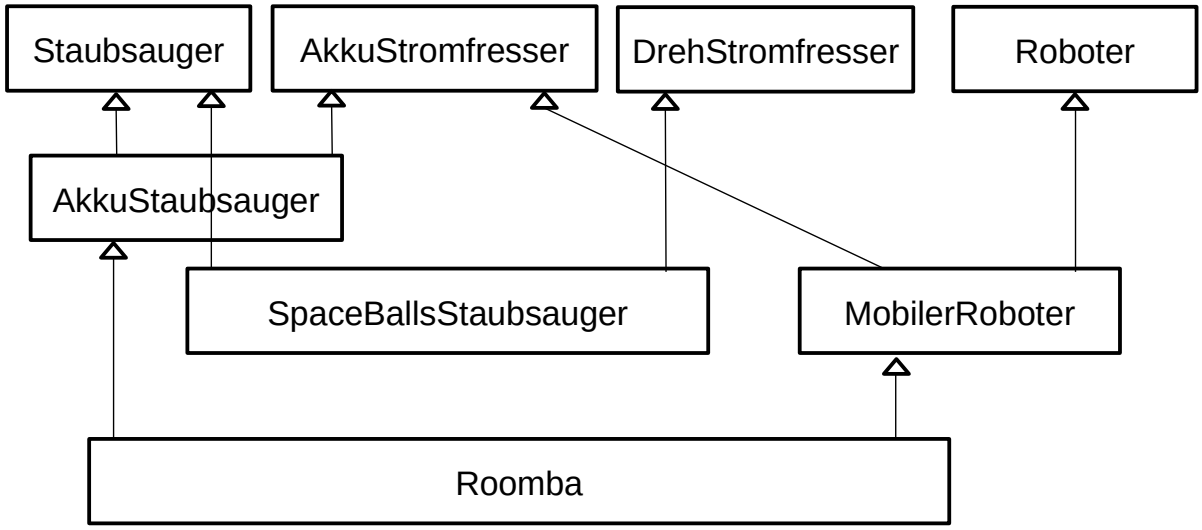
Komposition statt Vererbung

- Vererbungshierarchien werden trotzdem häufig als zu unflexibel angesehen
- Ein möglicher Ausweg:
  - Klassen flexiblen aus anderen Objekten zusammensetzen
  - Einzelobjekte modellieren Aspekte des Verhaltens des Gesamtobjekts
  - Werden beim Anlegen des Gesamtobjekts übergeben
- Engl.: Prefer composition over inheritance

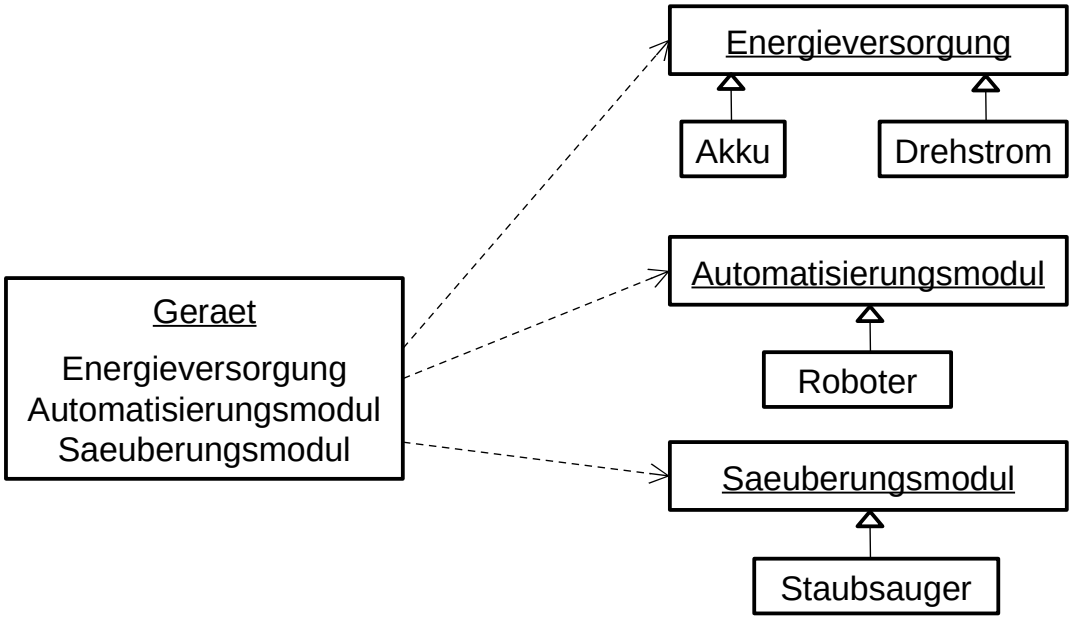
- Insbesondere bei Script-Sprachen beliebt, aber z.B. auch golang
- Aber relativ universell anwendbar

Komposition statt Vererbung – Motivierendes Beispiel

Wir möchten die Stromfresser gerne flexibler machen. Es soll batterie- und kabelgebundene Geräte geben. Nur mit Vererbung wird das unübersichtlich!



Komposition statt Vererbung – Auflösung



Ansatz: Jedes Gerät besteht aus mehreren Komponenten, für welche es ggf. Varianten gibt, die wiederum per Vererbung modelliert werden.

Komposition statt Vererbung – Auflösung (II)

```
class Energieversorgung {
public:
    void laden() = 0;
};
class Akku : public Energieversorgung {
public:
    void laden() { /* doSomething */ }
};
class Drehstrom : public Energieversorgung {
public:
    void laden() { /* ignore */ }
};

class Automatisierungsmodul {
public:
    void steuere() = 0;
};
class Roboter : public Automatisierungsmodul{
public:
    void steuere() { /* call HAL */ }
};
class DumbDevice : public Automatisierungsmodul {
public:
    void steuere() { /* do nothing */ }
};
```

Komposition statt Vererbung – Auflösung (III)

```
class Saeuberungsmodul {
public:
    void sauberMachen() = 0;
};
class Staubsauger : public Saeuberungsmodul {
public:
    void sauberMachen() { std::cerr << "Krach"; }
}

class Geraet {
protected:
    Energieversorgung*      _e;
    Automatisierungsmodul*  _a;
    Saeuberungsmodul*       _s;
    Geraet(Energieversorgung* e, Automatisierungsmodul* a,
           Saeuberungsmodul* s) : _e(e), _a(a), _s(s) {}

public:
    void laden()        { _e->laden(); }
    void steuere()      { _a->steuere(); }
    void sauberMachen() { _s->sauberMachen(); }
};
```


Komposition statt Vererbung – Auflösung (IV)

- Wie komme ich jetzt zu meinem Roomba?

```
class Roomba : public Geraet {
public:
    Roomba() : Geraet(new Akku(), new Roboter(),
                      new Staubsauger()) {}
};
```

- Zusammengefasst:
  - Viel Schreibarbeit, nicht viel übersichtlicher
  - Zusätzlicher Overhead durch Zeigerdereferenzierungen
  - Möglicherweise erweiterbarer:

```
class WallE : public Geraet {
public:
    WallE() : Geraet(new SolarStrom(), new Roboter(),
                     new SchaufelHaende()) {}
};
```

  - Individuelles Abwägen nötig!

Überblick – Einführung in C++

- Vergleich mit Java
- Speichermanagement
- Vererbung
- Mehrfachvererbung
- **Operator-Overloading**
- Templates
- Container
- Shared Pointer

Überladen von Operatoren

- In Java: Unterschied zwischen “==“ und `equals()` bei String-Vergleich
- In C++: “==“-Operator für String-Vergleich
- Generisches Konzept um lesbareren Code zu erzeugen!
  - Die meisten Operatoren dürfen für eigene Klassen überladen/überschrieben werden….
  - Vereinfacht Nutzung und Lesen teils dramatisch
    - `c = a + b` statt `c = a.plus(b)` für komplexe Zahlen?!
    - `std::cout << MagicString() << std::endl`?!
  - Erlaubt Implementierung von Datentypen, die sich wie primitive Datentypen “anfühlen”
- Umsetzung:
  - Hinzufügen einer Methode mit Namen `operator`*x* wobei für *x* unter anderem zulässig: `+ - * / % ^ & | ~ ! = < > += -= *= /= %= ^= &= |= << >> >>= <<= == != <= >= <=> && || ++ -- , ->* -> () []`
  - Impl. aber auch über Funktionen außerhalb von Klassen möglich

Überladen von Operatoren – Arithmetisches Beispiel

- Klasse um gebrochene Zahlen zu verarbeiten:

```
class Bruch {
public:
    using Val = int; // “Makro” um internen Typ zentral festzulegen
    Bruch(Val n, Val d) : n(n / ggT(n, d)), d(d / ggT(n, d)) {}
    Bruch& operator*=(const Bruch& r) {
        Val nn = n * r.n / ggT(n * r.n, d * r.d);
        d = d * r.d / ggT(n * r.n, d * r.d);
        n = nn;
        return *this;
    }
private:
    Val ggT(Val a, Val b) { return b == 0 ? a : ggT(b, a % b); }
    Val n, d;
};
```

- Leider nur einen Operator. Welche weiteren sind sinnvoll?

Überladen von Operatoren – Speicherbeispiel

- = erlaubt Kopieren → muss für sicheren Speicher impl. werden!

```
class MagicString { // Remember me?
  std::string* s;
public:
  MagicString() : s(new std::string("wiz!")) {}
  MagicString(const MagicString& m) : s(new std::string(*m.s)) {}
  std::string* magicStr() { return s; }
  // Neu: = operator erlaubt Zuweisungen
  MagicString& operator=(const MagicString& other) {
    if(this != &other) {
      // ACHTUNG beide Werte werden dereferenziert...
      // ruft operator= in std::string auf → String wird kopiert
      *s = *other.s;
    }
    return *this;
  }
  ~MagicString() { delete s; }
};
```


Überladen von Operatoren: Fazit

- Vereinfacht Nutzung komplexer Datentypen teilweise sehr stark

- Aber: Erfordert Disziplin beim Schreiben von Code
  - Oft erwartet: Freiheit von Exceptions
    - Wer gibt Speicher frei, wenn eine Zuweisung fehlgeschlagen ist?

  - Semantik der Operatoren muss selbsterklärend sein
    - Ist der Operator auf einem multiplikativen Ring + oder * ?
    - Was ist, wenn zwei ungleiche Objekte jeweils kleiner als das andere sind?
    - Ist * bei Vektoren das Skalar- oder das Kreuzprodukt (oder etwas ganz anderes)?

Überblick – Einführung in C++

- Vergleich mit Java
- Speichermanagement
- Vererbung
- Mehrfachvererbung
- Operator-Overloading
- **Templates**
- Container
- Shared Pointer

Templates

- Generische Datentypen werden in C++ mit Templates realsiert
- Häufig ähnlich eingesetzt wie Generics, aber können neben Typen auch Konstanten enthalten
- Zur Compile-Zeit aufgelöst → Deklaration & Implementierung in Header-Dateien
- Einfaches Beispiel (mit Typen, ähnl. zu Generics, primitive Typen ok!):

```
template<typename T> // typename keyword → deklariert T als Typ
T max(T a, T b) {
  return (a > b ? a : b);
}
```

```
int i = 10;
int j = 2;
int k = max<int>(j, i); // explizit
int l = max(j, i); // automat. Typinferenz durch Parametertypen
```


Templates – Beispiel mit Konstanten

- Berechnung Fibonacci

```
using u64 = unsigned long long; // Schreibarbeit sparen...
template<u64 i>
u64 fib() { return fib<i - 1>() + fib<i - 2>(); }
template<> // Spezialisierung des Templates für 1
u64 fib<1>() { return 1; }
template<> // Spezialisierung des Templates für 0
u64 fib<0>() { return 1; }
```

- Ausgabe?

```
std::cout << fib<3>();
```

- Wie lange dauert die Programmausführung?

```
std::cout << fib<100>();
```

- Was passiert wenn Basisfälle vergessen werden?

Templates – Beispiel mit Klassen

- Ein sehr einfacher, generischer Stack:

```
template<typename T>
class Stack {
  T* data;
  int num;
public:
  Stack() : data(new T[100]), num(0) {}

  void push(const T& d) {
    if(num >= 99)
      throw std::runtime_error("Stack overflow");
    data[num++] = d;
  }
  ...
};
```


Templates – SFINAE

- Ein wichtiges Grundkonzept von Templates: *Substitution failure is not an error* (SFINAE) ➞ es wird solange nach passenden Templates (in lexikogr. Reihenfolge) gesucht bis Parameter passen (sonst Fehler!)

```
template<typename T> T quadrieren(T* ptr) {
    return (*ptr) * (*ptr);
}
template<typename T> T quadrieren(T i)  {
    return i * i;
}

...

int i = 0;
std::cout << quadrieren(i) << std::endl;
std::cout << quadrieren(&i) << std::endl;
```

- Sehr häufig verwendetes Konstrukt & mächtiger als es scheint, aber schwer zu beherrschen

Templates – SFINAE

- Aufruf von `quadrieren` auch mit unserer Klasse `Bruch`:

```
template<typename T> T quadrieren(T* ptr) {
    return (*ptr) * (*ptr);
}
template<typename T> T quadrieren(T i)  {
    return i * i; // Bruch hat kein *-Operator, nur *=-Operator
}
template<typename T> T quadrieren(T i)  { // Fehler: Doppelte Def.
    T b(i);
    b *= i;
    return b;
}
```

- Funktioniert nicht!
- Auswege:
  - Wir könnten eine Spezialisierung machen (siehe Fibonacci), aber das ist nicht generalisierbar
  - Wir können alternativ versuchen, durch SFINAE zu verhindern, dass Funktionen doppelt definiert sind

TELEMATIK Rechnernetze

Templates – SFINAE

- Trick: Einführen eines Pseudoparameters, der nicht benutzt wird

```
template<typename T>
T quadrieren(T i, typename T::Val pseudoParam = 0)  {
    T b(i); b *= i; return b;
}
```

→ verhindert dass diese Version durch für T = int aufgerufen wird, weil int keinen Subtyp Val besitzt

- Verhindert aber nicht, dass

```
template<typename T> T quadrieren(T i)  {
    return i * i;
}
```

für T = Bruch instanziiert wird

- Kann man das auch durch Pseudoparameter lösen?
  Aber: wie erkennt man `int`s?!

Templates – SFINAE

- Trick: Einführen eines Hilfstemplates (sogenannter trait)

```
template<typename T> struct arithmetic {};
template<> struct arithmetic<int> { using Cond = void*; };
```

  → wenn arithmetic<T>::Cond definiert ist, muss T = int sein
- Überladen für weitere Typen wie double, unsigned int, ...
- Definition einer Funktion, die nur für int instanziiert werden kann:

```
template<typename T>
T quadrieren(T i, typename arithmetic<T>::Cond = nullptr)  {
    return i * i;
}
```

- Kompliziert? Ja, aber
  - Vordefinierte Hilfstemplates: `std::enable_if`, `std::is_arithmetic`, ...
  - In C++20 gibt es dafür Concepts, aber diese sind (noch) nicht verbreitet

Überblick – Einführung in C++

- Vergleich mit Java
- Speichermanagement
- Vererbung
- Mehrfachvererbung
- Operator-Overloading
- Templates
- **Container**
- Shared Pointer

Container

- Templates werden an vielen Stellen der C++ Standard-Bibliothek verwendet
- Container implementieren alle gängigen Datenstrukturen
- Prominente Beispiele:

```
template<typename T> class vector; // dynamisches Array
template<typename T> class list;   // doppelt verkette Liste
template<typename T> class set; // geordnete Menge basiert auf Baum
template<typename K, typename V> class map; // Assoziatives Array,
                                            // geordnet

// wie oben aber basierend auf Hash-Datenstruktur
template<typename T> class unordered_set;
template<typename K, typename V> class  unordered_map;
```

Alle Templates sind stark vereinfacht dargestellt, weitere Parameter haben Standardwerte, die z.B. Speicherverhalten regeln

Container – Enumerieren

- Je nach Struktur unterschiedlicher Zugriff
- Oft über Iteratoren vom Typ `Container::iterator`, bspw. `vector<int>::iterator`

```
std::vector<int> v{ 1, 2, 3 }; // Initialisierung über Liste

// “normale” for-Schleife, Beachte: Überladene Operatoren ++ und *
for(std::vector<int>::iterator i = v.begin(); i != v.end(); ++i) {
    std::cout << *i << std::endl;
}

// auto erlaubt Typinferenz → Code lesbarer, aber fehleranfälliger
for(auto i = v.begin(); i != v.end(); ++i) {
    std::cout << *i << std::endl;
}

// range loop (nutzt intern Iteratoren),
// komplexe Datentypen nur mit Ref. “&” sonst werden Kopie erzeugt!
for(int i : v) { // hier ohne “&”, da nur int in v gespeichert
    std::cout << i << std::endl;
}
```


Container – Einfügen

- Unterschiedliche Operationen je nach Container-Typ

- `std::vector<T>::push_back()` fügt neues Element am Ende ein
  - Allokiert ggf. neuen Speicher

  → Exisitierende Pointer können dadurch invalidiert werden!!!

- `std::list<T>` zusätzlich push_front() fügt Element am Anfang ein

- `std::set, std::map, …`
  - `insert()` fügt Element ein, falls es nicht existiert
    - Optional mit Hinweis wo ungefähr eingefügt werden soll
  - `operator[]` erlaubt Zugriff aber auch Überschreiben alter Elemente
  - `emplace()` Einfügen, ohne Kopien zu erzeugen (nicht behandelt)

Container – Löschen

- Unterschiedliche Operationen je nach Container-Typ
- Allgemein: `erase(Container::iterator)`
  - Vorsicht ggf. werden Iterator/Zeiger auf Objekte dadurch ungültig!

- `std::vector<T>::resize()` löscht implizit letzte Elemente bei Verkleinerung
- `std::vector<T>::pop_back()`entfernt letztes Element
- `std::list<T>` hat zusätzlich `pop_front()`
- `std::set, std::map, …` löschen nur mit `erase()`

Überblick – Einführung in C++

- Vergleich mit Java
- Speichermanagement
- Vererbung
- Mehrfachvererbung
- Operator-Overloading
- Templates
- Container
- **Shared Pointer**

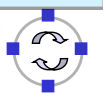
Joining Things – Shared Pointer

- Synonym: Smart Pointer
- Ziel: Sichereres Verwenden von Speicher
- Idee: kleine, schlanke Zeiger-Objekte, die Referenzzähler + Zeiger auf komplexere Objekte enthalten, wird letztes Zeiger-Objekt gelöscht, wird auch das komplexe Objekt gelöscht
- Realisierung mit RAII, Templates, Operator-Überladung
- Beispiel, wie shared_ptr sich verhalten sollten

```
using stringP = shared_ptr<std::string>;
stringP hello() { // gibt kopie der referenz zurück
    return stringP(new std::string("Hello!"));
}
int main() {
    stringP x = hello();
    stringP y(x); // Erstellen einer weiteren Referenz
    std::cout << y->length();
    return 0; // Original-String wird gelöscht wenn letzte Ref. weg
}
```


Joining Things – Shared Pointer

```
template<class T> class shared_ptr { // Vereinfacht!
    T*      p; // Zeiger auf eigentliches Objekt
    int*    r; // Referenzzähler
public:
    // neue Referenz auf Objekt erzeugen
    shared_ptr(T* t) : p(t), r(new int) { *r = 1; }
    // Referenz durch andere Referenz erzeugen
    shared_ptr(const shared_ptr<T>& sp) : p(sp.p), r(sp.r) { ++(*r); }
    T* operator->() const { // benutzen wie einen richtigen Zeiger
        return p;
    }
    ~shared_ptr() {
        if(--(*r) == 0) { // Objekt loeschen, wenn letzte Referenz weg
            delete r;
            delete p;
        }
    }
}; // TODO operator= implementieren!
```

Zusammenfassung

- C++ erlaubt sehr detaillierte Kontrolle über Speicher- und Laufzeitverhalten
  - Es ist relativ einfach, schwierig zu findende Fehler einzubauen
- Die Sprache ist durch Operator-Overloading, Mehrfachvererbung und Templates sehr mächtig
  - Erlaubt hohen Grad an Wiederverwendung
  - Anzahl an Code-Zeilen kann reduziert werden
  - Code kann völlig unlesbar werden! Viele Features sollten nur eingesetzt werden wenn sich dadurch ein wirklicher Vorteil ergibt!

“Where there is great power there is great responsibility!”

Viele weiterführende Bereiche

- Löschen von Default-Kopieroperationen & boost::noncopyable
- constexpr
- Lambda-Funktionen & std::bind
- Rvalue-Referenzen & std::move
- Concepts
- Variadic Templates
- Überschreiben von new und delete
- Module
- Standard Library & Boost:
  - Multi-index Container
  - Atomics, Threads, Futures, Co-Routinen etc.
  - std::tie
  - Ranges

Weiterführende Themen

- Einige Best Practices im Netz:
  - https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md
  - https://docs.microsoft.com/en-us/cpp/cpp/welcome-back-to-cpp-modern-cpp?view=vs-2019

- Vorlesung „Softwaretechnik” wird einige OO-betreffende Themen weiter vertiefen