C++

Jan Fässler

3. Semester (HS 2012)

If you use this documentation for a exam, you should offer a beer to the authors!

Inhaltsverzeichnis

1 Einleitung

1.1 C++ verglichen mit Java

1.1.1 Gemeinsamkeiten

- typisierte, objektorientierte Sprache
- sehr ähnliche Syntax (Java-Syntax wurde an C++ angelehnt)
- ähnliche Grundtypen, Operatoren und Klassenkonzept

1.1.2 Unterschiede

- C++-Programm muss nicht objektorientiert sein
- plattformabhängiger Maschinencode anstatt Bytecode für die VM
- C++-Programme ko?nnen aufs unterliegende System zugreifen
- Flexibleres Speichermanagement
- Flexiblerer Polymorphismus
- Effizienz vor Sicherheit
- Unterscheidung zwischen Referenzen und Zeigern
- keine strikt geschachtelten Namensräume
- Trennung zwischen Schnittstelle und Implementierung

1.2 C++-Dateien

***.c**
: Dateien, die mit dem C-Compiler kompiliert werden

***.cpp**
: Dateien, die mit dem C++-Compiler kompiliert werden

Header-Datei (*.h)
: enthält oft mehrfach benötigte Definitionen und wird nicht direkt kompiliert, sondern in eine oder mehrere cpp-Dateien importiert

***.hpp**
: ursprünglich als reine C++-Header-Dateien gedacht, werden aber selten verwendet

1.3 Programmerzeugung

1.3.1 Präprozessor

- Programmcode darf Makros enthalten
- Makros werden unmittelbar vor der Kompilation evaluiert
- Bsp. Substitution von Konstanten, bedingte Kompilation
- Verwendung:

#define
: definiert ein Symbol/Makro mit oder ohne Parameter

#undef
: löscht eine Definition eiens Symbols/Makros, bzw ist danach nicht mehr definiert

#ifdef und #endif
: bedingte Kompilation: die Kompilation eines Textblocks ist abha?ngig von der Definition eines Symbols

1.3.2 Compiler

- Syntaxüberprüfung des Quellcodes
- Erzeugung von Objektdateien (Maschinencode mit unaufgelösten Verknüpfungen zu anderen Objektdateien)
- Der C++-Compiler ist ein One-Pass-Compiler. Das bedeuted bevor ein Bezeichner (Variable, Klasse usw.) verwendet werden darf, muss er deklariert bzw. definiert werden. Die Deklaration bzw. Definition eines Bezeichners muss vor seiner Benutzung kompiliert werden.

1.3.3 Linker (Binder)

- Erzeugung von Bibliotheken oder ausführbaren Programmen aus einzelnen Objektdateien
- Verknüpfungen zwischen Objektdateien werden aufgelöst
- Optimierungen (z.B. Entfernung nicht verwendeter Prozeduren) sind möglich

2 Variablen und Methoden

2.1 Global

- Main-Funktion ist global (Teil des globalen Namensraums)
- Klassen, Methoden, Variablen sind Teil eines Namensraums (benannt oder global)
- uneingeschränkte Sichtbarkeit: aus allen Programmteilen können sie verwendet werden (Sichtbarkeit über die Objektdateigrenze hinweg)
- Verwendung: wenn immer möglich vermeiden, da das Information Hiding Prinzip stark unterwandert wird

2.2 Modular

Sichtbarkeitsbereich
... ist beschränkt auf die Objektdatei, jedoch über Methoden- und Klassengrenzen hinweg

Einsatzgebiet
bei nicht-objektorientierter Programmierung als Ersatz von Klassenvariablen und -methoden (in OO: Einsatz vermeiden)

2.3 Einfache Datentypen

Speicherbedarf der einfachen Datentypen ist Compiler spezifisch. Alle ganzzahligen Datentypen (inkl. char) gibt es vorzeichenlos (unsigned) und vorzeichenbehaftet (signed) in der Zweierkomplementdarstellung.

2.4 Neue Zeichentypen

bisher

8-Bit String const char *s = ”abcd”;

16-Bit String const wchar_t *s = L”abcd”;

neu

UTF8 String const char *s = u8”abcd”;

UTF16 String const char16_t *s = u”abcd”;

UTF32 String const char32_t *s = U”abcd”;

Unicode-Codepoints

16 Bit Unicode-Codepoints \u1234 (4-stelliger Hex-Code)

32 Bit Unicode-Codepoints \u123456 (6-stelliger Hex-Code)

2.5 Schlüsselwörter

typedef

Es dient der Festlegung eigener Typenbezeichner.
Bsp.:
typedef int INT32;
typedef unsigned long long int UINT64;

using (C++11)

Dieses kann auch fu?r eigene Typenbezeichner verwendet werden.
Bsp.:
using INT32 = int;
using UINT64 = unsigned long long int;

auto (C++11)

Bei Variablendefinitionen, wo aus dem Initialisierungswert der Variable der Typ der Variable fu?r den Compiler automatisch ersichtlich ist, kann das Schlu?sselwort auto anstatt des konkreten Typs hingeschrieben werden.
Bsp.:
auto x = 7;
double f();
auto g = f();

decltype (C++11)

decltype(x) ist eine Funktion, welche den Deklarationstyp des Ausdruckes x zuru?ckgibt.
Bsp.:
decltype(8) y = 8;
decltype(g) h = 5.5;

const

Im Gegensatz zu Java wo dieses Wort reserviert aber nicht verwendet wird, hat es in C++ vielfältiger Einsatz mit unterschiedlicher Semantik. Die hier verwendete Semantik: nach Initialisierung nur noch lesender Zugriff.
Beispiele:
const unsigned int SIZE = 1000;
const auto LENGTH = 500;
const char GRADES[] = 'A', 'B', 'C', 'D', 'E', 'F' ;
const char NOTEN[] = 1, 2, 3, 4, 5, 6 ;
double const PI = 3.141596;
auto const PID2 = PI/2;

constexpr (C++11)

Verallgemeinerung des Schlüsselwort const fu?r konstante Ausdrücke, welche auch Funktionsaufrufe und als Spezialfall auch Konstruktoren enthalten dürfen und stellt statische Initialisierung zur Kompilationszeit sicher.
Beispiel:
constexpr int getFive() return 2 + 3;
int array[getFive() + 7];

2.6 Initialisierungslisten

Listing 1: Initialisierungsliste

```
#include <initializer_list> struct Tuple {
int value[];
Tuple(initializer_list<int> v);
Tuple(int a, int b, int c); Tuple(initializer_list<int>v,size_tcap); //#3
};
Tuple t1{1, 2, 3};    // Konstruktor #1 wird verwendet
Tuple t2{2, 4, 6, 8}; // Konstruktor #1 wird verwendet
Tuple t3(4, 5, 6};    // Konstruktor #2 wird verwendet
```

Die Initialisierungslisten sind ein neuer C++ Typ. Wenn die Initialisierungsliste der einzige Parameter ist, kann wie oben gezeigt vorgegangen werden. Wenn noch weitere Parameter vorhanden sind, dann müssen die geschweiften Klammern verschachtelt werden Tuple:
t4 = {{1, 2, 3, 4}, 4};

3 Zeiger & Referenzen

3.1 Zeiger und Adressoperator

Ein Zeiger zeigt auf eine Speicherstelle des (virtuellen) Adressraums. Zeigen können im Quellcode Typinformationen mitführen. Von jeder Variable und jedem Objekt kann mit dem Adressoperator & zur Laufzeit die Adresse (Speicherstelle) abgefragt werden.

Listing 2: Zeigerbeispiele

```
typedef unsigned int * PUInt32;
char text[] = "test";
unsigned int i = 2;
char c = text[i + 1];
char *p = text, *q = text + 1, *r = &text[i], *s = &c, *t = nullptr, *u = 0;
PUInt32 x = &i;
void *y = x;
```

3.2 Zeiger und Konstanten

int *p = &x
nichts ist konstant

const int *p = &x
nur Ziel ist konstant: p ist ein Zeiger auf einen konstanten Integer

int *const p = &x
nur Zeiger ist konstant: p ist ein konstanter Zeiger auf einen Integer

const int *const p = &x
Ziel und Zeiger sind beide konstant

3.3 Referenzen

- eine Referenz ist ein Alias für eine andere Variable eine Referenz wird durch ein & gekennzeichnet
- eine Referenz kann nicht uninitialisiert sein
- eine Neuinitialisierung ist nicht möglich
- hinter der Kulisse ist eine Referenz nichts Anderes als ein Zeiger

Listing 3: Beispiele

```
int k = 2;
int& ref = k;      // ref ist ein Alias fu?r k
ref = 3;           // die Variable k hat nun den Wert 3
int *pk = &k;
int*& ref2 = pk;   // ref2 ist ein Alias fu?r den Zeiger pk
*ref2 = 4;         // die Variable k hat nun den Wert 4
```

3.4 Zugriff

Listing 4: Zugriff auf Instanzvariablen und Instanzmethoden

```
Person *pers = new Person(); // Pointer
Person& refP = pers;         // Referenz
name = pers->getName();
name = refP.getName();
```

3.5 Smart Pointers

3.5.1 Grundlegendes

Prinzip

- spezielle Zeigerobjekte verwalten Heap-Adressen
- mittels Referenzzähler wird festgehalten, wie viele Zeigerobjekte auf das gleiche Objekt auf dem Heap zeigen
- im Destruktor des Zeigerobjektes wird der Referenzzähler u?berprüft und das Objekt auf dem Heap automatisch gelöscht, wenn keine weiteren Zeigerobjekte mehr auf das gleiche Objekt zeigen

Ziel

- der Umgang mit den Zeigerobjekten muss annähernd so einfach sein, wie der Umgang mit gewöhnlichen Zeigern, d.h. der Benutzer soll nichts mit dem Referenzzähler zu tun haben

3.5.2 Zeigerobjekte

std::unique_ptr$< T >$

- Zeigerobjekt ist der Besitzer des Objektes, auf welches verwiesen wird
- pro Objekt existiert ho?chstens ein einziger unique_ptr
- das Objekt wird beim Aufruf des Destruktors des Zeigerobjekts zerstört

std::shared_ptr$< T >$

- Zeigerobjekt beinhaltet einen Referenzzähler
- mehrere Zeigerobjekte ko?nnen auf das gleiche Objekt zeigen
- das Objekt wird beim Aufruf des Destruktors des Zeigerobjekts nur dann zerstört, wenn keine weiteren Zeigerobjekte aufs gleiche Objekt zeigen

std::weak_ptr$< T >$

- zum Aufbrechen von zyklischen Abhängigkeiten

4 Arrays

4.1 C-Arrays

Die Länge des Arrays wird nicht abgespeichert in C++. Die Länge ist nur im Sichtbarkeitsbereich der Definition des Arrays bekannt. Sehr grosse Arrays sollen auf dem Heap (dynamisch) angelegt werden.

statische Erzeugung

- Die Länge ist konstant und zur Kompilationszeit bekannt
- Array kann auf dem Stack angelegt werden
- Beispiel: char text[100];

statische Erzeugung

- Array wird zur Laufzeit auf dem Heap angelegt
- Beispiel:

```
1 const int len = 100; // len kann, aber muss nicht konstant sein
  char *text = new char[len];
  delete[] text;
```

4.1.1 C-Array als Funktionsparameter

Funktion liegt ausserhalb des Sichtbarkeitsbereichs der Definition des Arrays.

- Länge des Arrays ist nicht bekannt und sollte als Parameter mitgegeben werden
- sizeof kann nur die Anzahl Bytes des Zeigers ermitteln
- 2 gleichwertige Schreibweisen:
  - void print(char *s) { ... }
  - void print(char s[]) { ... }

4.2 C-Strings

Ein C-String ist ein ein eindimensionales Character-Array. Das Ende der gu?ltigen Zeichenkette ist durch ein 0-Character gekennzeichnet.
Der Unterschied zu anderen Arrays:

vereinfachte Initialisierung erlaubt

- char s[] = ¨Das ist ein Test.¨; // String-Schreibweise anstatt Initialisierungsliste
- s zeigt auf eine Kopie des konstanten Strings ¨Das ist ein Test.¨
- sizeof(s) gibt den Speicherbedarf der Kopie zurück (im Sichtbereich der Def.)

implizite Konstante

- const char *t = ¨Das ist ein Test.¨;
- t zeigt direkt auf den konstanten String der Länge 18!
- sizeof(t) gibt die Anzahl Bytes des Zeigers t zurück

4.3 Mehrdimensionale C-Arrays

4.3.1 Grundsätze

- mehrdimensionale C-Arrays werden im Hintergrund als eindimensionale Arrays abgespeichert
- die Länge der ersten Dimensionen ist nur im Sichtbarkeitsbereich der Definition des Arrays bekannt
- die Längen der weiteren Dimensionen gehen mit in den Typ ein
- sehr grosse Arrays sollen auf dem Heap (dynamisch) angelegt werden

4.3.2 Statische Erzeugung

- Anzahl Dimensionen ist fix und zur Kompilationszeit bekannt
- Längen der Dimensionen sind konstant und zur Kompilationszeit bekannt
- Syntax: Type Variable [dim1][dim2] .. [dimN];
- Beispiele:
  int matrix[2][3]; // matrix ist nicht initialisiert
  int matrix2[2][3] = {{ 1, 2, 3 }, { 4, 5, 6 }};

4.3.3 Dynamische Erzeugung

- Anzahl Dimensionen ist fix und zur Kompilationszeit bekannt
- Längen der Dimensionen sind konstant und mit Ausnahme der ersten zur Kompilationszeit bekannt
- Array erzeugen mit new und löschen mit delete[]
- Syntax: Type (* const Variable)[dim2] .. [dimN] = new Type [dim1][dim2] .. [dimN];
- Beispiele:
  int (* const pMatrix)[7] = new int[5][7];
  int (* const pMatrix)[7] = new int[5][7];
  delete[] pMatrix; delete[] pFloats;

4.3.4 nD-Array als Funktionsparameter

- Die Länge der ersten Dimension muss als Parameter zusa?tzlich übergeben werden
- alle Dimensionen bis auf die erste gehen in den Typ mit ein
- Interpretation als Folge von Arrays möglich
- Beispiel:

```
void print(int m[][dim2], int dim1) { ... }
```

4.4 C++ Arrays

- Array mit fester Grösse
- generische Klasse aus der STL
- kapselt ein C-Array fixer Länge und bietet ein Set nützlicher Array- Methoden
- Beispiele:

Listing 5: C++ Array

```
#include <array>
string s[] = { "ab", "cd", "ef", "gh" };
const int slen = sizeof(s)/sizeof(string);

array<string, slen> a;

for(int i=0; i < slen; i++) {
  cout << "a[" << i << "] = " << a[i] << endl;
  a[i] = s[i]; // copy by value (deep copy)
  cout << "a[" << i << "] = " << a[i] << endl;
}
```

4.5 C++ Vektoren

Ein Vektor ist eine generische Klasse aus der STL, sie entspricht der ArrayList aus Java. Beispiele:

Listing 6: C++ Vekotr

```
#include <vector>
string s[] = { "ab", "cd", "ef", "gh" };
const int slen = sizeof(s)/sizeof(string);

vector<string> v1;
vector<string*> v2;
vector<shared_ptr<string>> v3;   // C++11

for(const string &t: s) {    // C++11: neue for-Schleife wie in Java
// for(const auto& t: s) {   // wäre auch möglich
  v1.emplace_back(t);      // copy by value (deep copy)
  v2.push_back(new string(t));
  v3.push_back(make_shared<string>(t));
}
// Iteration im alten Stiel
for(vector<string*>::iterator i = v2.begin(); i != v2.end(); ++i) {
  delete *i;
}
// Iteration im neuen Stil mit Lambda-Ausdruck (C++11)
cout << "v3 = ";
for_each(begin(v3), end(v3), [&](const shared_ptr<string>& s) {
  cout << s << " (" << *s << "), ";
});
cout << endl;
// Ausgabe: v3 = 003AF70C (ab), 003AF7CC (cd), 003AF764 (ef), 003AF864 (gh),
```

5 Klassen

5.1 Deklaration

In C++ kann man Klassen auf zwei Arten erzeugen:

struct
: in C: Verbund (Record) von verschiedenen Datenfeldern
in C++: öffentliche Klasse (alle Members sind public per Default)

Listing 7: struct-Klasse

```
struct Point {
  int m_x, m_y;
  void setY(int y) { m_y = y; }
};
```

class
: nur in C++: alle Members sind private per Default

Listing 8: class-Klasse

```
1 class Person {
    char m_name[20];
    int m_alter;
  public:
    char * getName() { return m_name; }
6 };
```

5.2 Instanztypen

5.2.1 statisch

- Syntax: Punkt p;
- Compiler hat aus der Definition der Klasse Punkt berechnet, wie viel Speicher eine Instanz der Klasse Punkt benötigt
- auf dem Stack wird entsprechend Platz reserviert, so dass alle vier Attribute des Punktes abgespeichert werden können
- die Variable p repräsentiert die Punktinstanz auf dem Stack

5.2.2 dynamisch

- Syntax: Punkt *p; p = new Punkt();
- Ein neues Objekt wird erzeugt und auf dem Heap angelegt
- Ein Zeiger auf das neue Objekt wird zurückgegeben und in der Zeigervariablen p abgespeichert

5.3 Vorgabeparameter

- Parameter in Methoden du?rfen mit Standardwerten belegt werden
- für Default-Parameter müssen beim Methodenaufruf keine Werte angegeben werden (es dürfen aber)
- in der Parameterliste einer Methode müssen zuerst alle Parameter ohne Default-Wert und dann alle Parameter mit Default-Wert aufgelistet werden
- Beispiel:

Listing 9: Vorgabeparameter

```
Punkt(double x, double y, double z, Color color = 0){ ... }
```

5.4 Klassenvariablen und -methoden

5.4.1 Klassenvariablen

- werden pro Klasse und nicht pro Instanz angelegt
- alle Instanzen einer Klasse haben Zugriff auf die gemeinsamen Klassenvariablen dieser Klasse
- Modifikator static vor dem Typ der Variable

5.4.2 Klassenmethoden

- können ohne Instanz einer Klasse aufgerufen werden
- werden über den Klassennamen aufgerufen
- dürfen nur auf Klassenvariablen zugreifen
- Modifikator static vor der Methoden-Deklaration

5.5 Default-Methoden

5.5.1 Konstruktor

- Konstruktoren heissen gleich wie die Klasse und initialisieren die Attribute eines Objekts
- können nur bei der Erzeugung von Objekten mit gleichzeitiger Initialisierung
- wenn kein Konstruktor definiert ist, dann stellt der Compiler einen vordefinierten Standard-Konstruktor bereit. Dieser hat keine U?bergabeparameter.
- Beispiele:

Listing 10: Konstruktor

```
class Punkt {
  Punkt(double x, double y, double z, int color) {
    m_x = x;
    m_y = y;
    m_z = z; m_color = color;
  }
}
// Verwendung von Initialisierungslisten:
class Punkt {
  Punkt(double x, double y, double z, int color) : m_x(x), m_y(y), m_z(z),
      m_color(color) { ... } }
// Anwendung:
Punkt p2(1, 2, 3, 5);
Punkt *p1 = new Punkt(4, 5, 6, 8);
```

5.5.2 Typkonvertierungs-Konstruktor

- wird zur impliziten Konvertierung herangezogen
- enthält u?blicherweise nur ein Argument (wird mit nur einem Argument aufgerufen)
- soll ein Konstruktor mit einem Argument nicht als impliziter Typkonvertierungs- Konstruktor missbraucht werden können, so muss vor dem Konstruktor das Schlüsselwort explicit geschrieben werden
- Beispiel:

Listing 11: Typkonvertierungs-Konstruktor

```
class Punkt {
  Punkt(double d[4]) : m_x(d[0]), m_y(d[1]) , m_z(d[2]), m_color(static_cast
      <int>(d[3])) { ... }
}
double array[4] = { 4.4, 3.3, 2.2, 5.0 };
Punkt p = array;
```

5.5.3 Kopierkonstruktor

- swenn Sie keinen eigenen Kopierkonstruktor und keine Verschiebeoperation definieren, dann stellt der Compiler einen vordefinierten Kopierkonstruktor fu?r eine flache Kopie bereit
- wird ein eigener Kopierkonstruktor angeboten, so sollte auch der Zuweisungsoperator angeboten werden
- Beispiel:

Listing 12: Typkonvertierungs-Konstruktor

```
class Punkt {
  Punkt(const Punkt& p)
    : m_x(p.m_x), m_y(p.m_y), m_z(p.m_z), m_color(p.m_color) { ... }
}
```

5.5.4 Destruktor

- wird automatisch aufgerufen, kurz bevor ein Objekt seine Gültigkeit verliert (unmittelbar vor der Zerstörung)
- trägt den gleichen Namen wie die Klasse, mit einem Tilde davor
- wenn Sie keinen eigenen Destruktor definieren, dann stellt der Compiler einen vordefinierten Standard-Destruktor bereit

Listing 13: Destruktor

```
class Punkt {
  ~Punkt() {
    cout << "Objekt ist nicht länger gültig" << endl;
  }
}
```

5.6 Resource Allocation is Initialization

- beim Erzeugen eines Objekts muss das Objekt initialisiert werden (im Konstruktor)
- beim ordentlichen Verlassen des Konstruktors immer ein gültiges Objekt zurücklassen
- im Fehlerfall sollte der Konstruktor mit einer Exception beendet werden, das bedeutet, dass bereits angeforderte Ressourcen wieder freigegeben werden müssen
- wird ein Objekt infolge einer Exception nicht vollständig initialisiert, so müssen die einzelnen Teile des Objektes sich selbsta?ndig abbauen

Listing 14: RAII-Lösungsansatz

```
#include <memory> // unique_ptr in C++11
struct StereoImage {
  std::unique_ptr<Image> left, right;
  StereoImage() : left(new Image), right(new Image) {}
  ~StereoImage() { delete left; delete right; }
};
```

5.7 Verschiebeoperationen (C++11)

- werden zum Verschieben eines Objektes verwendet (Move-Semantik)
- verwenden einen Parameter: Rechtswert-Referenz auf Objekt
- wenn Sie keinen eigenen Verschiebekonstruktor und keine Kopieroperation definieren, dann stellt der Compiler einen vordefinierten Verschiebekonstruktor bereit
- wird ein eigener Verschiebekonstruktor angeboten, so sollte auch der Verschiebeoperator angeboten werden

Listing 15: Verschiebeoperationen

```
class Vector {
  Punkt *m_array;
  int m_size;
public:
  // benötigt eigenen Standardkonstruktor und Destruktor
  Vector(Vector&& v) : m_array(v.m_array), m_size(v.m_size) {
    v.m_array = nullptr;
    v.m_size = 0;
  }
  Vector& operator=(Vector&& v) {
    m_size = v.m_size;
    v.m_size = 0;
    delete[] m_array;
    m_array = v.m_array;
    v.m_array = nullptr;
    return *this;
  }
};
Vector createVector() {
    Vector v;
    v.add(Punkt(1,2,3));
    v.add(Punkt(4,5,6));
    Vector v2;
    v2 = move(v);                 // ruft den Verschiebeoperator auf
    // v enthält keine Punkt-Objekte mehr
    return move(v2);
}
int main() {
    Vector v1(createVector());  // ruft den Verschiebekonstruktor auf
    Vector v2 = createVector(); // ruft den Verschiebekonstruktor auf
    Vector v3;                  // ruft den Standardkonstruktor auf
    v3 = createVector();        // ruft den Verschiebeoperator auf
};
```