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

Java

- Main-Methode ist eine Klassenmethode, also Teil einer Klasse
- Klassen sind Teil eines Packages (evtl. unbenanntes Package)
- Klassenvariablen geho?ren zum Namensraum einer Klasse

C++

- Main-Funktion ist global (Teil des globalen Namensraums)
- Klassen, Methoden, Variablen sind Teil eines Namensraums (benannt oder global)
- uneingeschränkte Sichtbarkeit: aus allen Programmteilen ko?nnen sie verwendet werden (Sichtbarkeit u?ber 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 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];

2.5 Initialisierungslisten

Listing 1: Initialisierungsliste

```
1 #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
  };
6 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.

Zeigervariablen bzw. Zeiger zeigen auf gültige Speicheradressen (zB. dynamische Objekte, aufs erste Element von Arrays oder auf statische Variablen und Objekte). Sie können aber auch auf ungültige Speicheradressen zeigen. Sie haben einen Typ ¨Zeiger auf ...¨. Soll eine Zeigervariable auf eine Instanz der Klasse C zeigen, so muss der Typ der Zeigervariablen zur Klasse C zuweisungskompatibel sein

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;
```

4 Klassen

4.1 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 3: 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 4: class-Klasse

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

4.2 Instanzen

Listing 5: Erzeugen von Instanzen

```
Point pnt1;                    // auf Stack
Point *pnt2 = new Point();     // auf Heap
Person pers1;                  // auf Stack
Person *pers2 = new Person(); // auf Heap
Person& refP = pers1;
```

Listing 6: Zugriff auf Instanzvariablen und Instanzmethoden

```
pnt1.m_x = 3;
pnt1.setY(pnt1.m_x);
pnt2->m_x = 4;
pnt2->setY(pnt2->m_x);
char *name = pers1.getName();
name = pers2->getName();
name = refP.getName();
```