Einführung in die Programmiersprache C++

Thomas Wiemann
Institut für Informatik
AG Wissensbasierte Systeme

Letzte Vorlesung

- `unions`
- `const`
- `extern`
- `static`

Gliederung

gdb (1)

- GNU-Debugger
- Umgebung zum Auffinden von Programmierfehlern
- Kann in ein Programm zur Laufzeit reinschauen
- `gcc` bzw. `g++` mit `-g` aufrufen
- Aufruf: `gdb` (bzw. `gdb --args programm p1 p1` ...)
- Kommandos:
  - `run`
  - `backtrace`
  - `break filename:linenum`
  - `quit`

gdb (2)

- Beispiel:

```
(gdb) run
Starting program: main
```

- Segfault:

```
Program received signal SIGSEGV, Segmentation fault.
Node<int>::next (this=0x0) at main.cc:28
28     Node<T>* next () const { return next_; }
(gdb)
```

- Backtrace:

```
(gdb) backtrace
#0  Node<int>::next (this=0x0) at main.cc:28
#1  0x2a16c in LinkedList<int>::remove (this=0x40160,
    item_to_remove=@0xffbef014) at main.cc:77
#2  0x1ad10 in main (argc=1, argv=0xffbef0a4) at main.cc:
111
(gdb)
```

Linus Torvalds über C++

> C++ is a horrible language. It's made more horrible by the fact that a lot of substandard programmers use it, to the point where it's much much easier to generate total and utter crap with it. Quite frankly, even if the choice of C were to do ***nothing*** but keep the C++ programmers out, that in itself would be a huge reason to use C

http://thread.gmane.org/gmane.comp.version-control.git/57643/focus=57918

Die Programmiersprache C++

- Ursprung
  - [vgl. Folie 10]
  - Dennis Ritchie arbeitete bei Bell Labs an UNIX
    - 1972 Programmiersprache C
    - Unix wurde zu 95% in C geschrieben
  - 1979 erste Version von „C with Classes“
    - Sprachkonzepte aus C
    - Klassenkonzepte aus simula67
  - 1983 Namensänderung in C++
- Seitdem viele Erweiterungen des Klassenkonzepts, z.B.
  - Überladen von Operatoren
  - Referenzen
  - Konstanten (const)
  - Schablonen (templates)
  - Ausnahmenbehandlung (exceptions)
  - Eindeutige Namensräume

Die Philosophie von C++

- Nahe am zu lösenden Problem arbeiten
  - Möglichkeit mit eleganten Abstraktionen zu programmieren
  - Starker Fokus auf Modularisierung
  - Verbesserungen gegenüber dem C-type-System
- Nahe an der Hardware arbeiten
  - Der Fokus von C bezüglich Performanz wird übernommen
  - Es gibt weiterhin die Möglichkeit der Low-level Datenmanipulation
- C++ besteht im Wesentlichen aus zwei Komponenten
  - C++ Sprachkern
    - Syntax, Datentypen, Kontroll-Fluss, Variablen,…
    - Funktionen, Klassen, Templates
  - C++ Standard Bibliothek
    - Zusammenstellung nützlicher Funktionen, die mit „C++ core“ geschrieben wurden
    - Generische Strings, Streams, Exceptions

C++ Hello World

- Erzeuge eine Datei hello.cc, z.B. mit emacs:

```
#include <iostream>
using namespace std;
int main()
{
   cout<< "Hello, world!" << endl;
   return 0;
}
```

- Übersetze das Programm:

```
% g++ hello.cc -o hello
% hello
Hello, world!
%
```

- Juhu!!
- C++-Dateien enden auf `.cc` oder `.cpp` oder `.cxx`
- Wir benutzen zunächst `g++`, aber es gibt auch viele andere C++-Compiler

Ein- und Ausgabe auf die Konsole (1)

- C benutzt `printf()`, `scanf()`, etc.

```
printf("Hello, world!\n");
```

- Definiert in der C-Header-Datei `stdio.h`
- C++ führt Datenströme (IO-Streams) ein

```
cout <<  "Hello, world!" << endl;
```

- Definiert in der Header-Datei `iostream`
- `cin` liest Eingaben von stdin
- `cout` macht Ausgaben auf stdout
- `cerr` zur Fehlerausgabe auf stderr
- << ist ein „überladener Operator“
  - Der Compiler findet selbst heraus, ob man „shift nach links“ oder Ausgaben in einen Datenstrom mein
  - Er wird von allen Standarddatentypen und C++-Strings unterstützt
  - `endl` bedeutet Zeilenumbruch

Ein- und Ausgabe auf die Konsole (2)

- Beispiel:

```
string name = "series";
int n = 15;
double sum = 35.2;
cout << "name = " << name << endl
     << "n = " << n << endl
     << "sum = " << sum << endl;
```

- Der Operator >> ist überladen für die Eingabe
- >> wird von allen Standarddatentypen und C++-Strings unterstützt

Ein- und Ausgabe auf die Konsole (3)

- Beispiel:

```
float x, y;
cout << "Enter x and y coordinates: ";
cin  >> x >> y;
```

- Eingaben werden durch „Whitespaces“ getrennt:

```
Enter x and y coordinates: 3.2    -5.6
Enter x and y coordinates: 4
35
```

- Ab jetzt gilt: **Nicht mehr `printf()` und `scanf()` benutzen**
- C- und C++-Funktionen verwenden die selbe Hardware, daher nicht mixen!
- Immer `endl` benutzen, da es in der Ausgabeklasse definiert ist

Namespaces

- Namespaces gruppieren Dinge, die zusammengehören
- Gedacht, um Namenskollisionen zu vermeiden
- Es kann Funktionen / Methoden / Klassen mit der gleichen Signatur und verschiedenen Namespaces geben
- Der gesamte Code der C++-Standardbibliothek befinden sich im Namespace std
- Entweder man schreibt `namespace::name` überall...

```
std::cout << "Hello, world!" << std::endl;
```

- Oder man erklärt dem Compiler, dass das Programm den Namespace `std` benutzt

```
using namespace std;
...
cout << "Hello, world!" << endl;
```

- `namespace::name` nennt man qualifizierter Name

C++ Funktionsargumente (1)

- Funktionsargumente werden in C++ via call-by-value übergeben
  - Eine Kopie jedes Arguments wird erstellt
  - Die Funktion arbeitet mit der Kopie und verändert das Original nicht
- Beispiel

```
void outputPoint(Point p)
{
  cout << "(" << p.getX()
       << "," << p.getY() << ")";
}


...
Point loc(35,-117);
outputPoint(loc);          // loc is copied
```

- Das Kopieren von vielen / großen Objekten ist aufwändig

Referenzen (1)

- C++ führt Referenzen ein
- Eine Referenz ist so was ähnliches wie ein Zeiger
- Die Verwendung von Referenzen ist analog zur Verwendung des Originals
- Beispiel:

```
int i = 5;
int &ref = i;
ref++;
cout << "Now i = " << i << endl;
```

- `ref` ist vom Type `int&` (Referenz zu einem `int`)
- Identisch: Benutzung von `ref` und `i`
- Vorteil gegenüber Pointern: Einfachere Schreibweise

Referenzen (2) / Funktionsargumente (2)

- Weitere Eigenschaften von Referenzen:
  - Der Referent kann sich wechseln – analog zu Pointern
  - Referenzen auf einfache Datentypen und Objekte
- Die Verwendung von Objektreferenzen als Funktionsparameter ist viel schneller!
- C++ Referenzen müssen immer irgendetwas referenzieren
- Beispiel: Rotation eines Punktes um 90°

Mit Pointer:

```
void rotate90(Point *p);
```

`NULL` kann übergeben werden!

Mit Referenz:

```
void rotate90(Point &p);
```

Es ist nicht möglich nichts zu übergeben!

- In C++ wird 0 statt NULL benutzt

Referenzen (3) / Funktionsargumente (3)

- Referenzen können zu Seiteneffekten führen:

```
void rotate90(Point &p) {
  double x = p.getX();
  double y = p.getY();
  p.setX(y);
  p.setY(-x);
}
...
Point f(5, 2);
rotate90(f);
```

- f wird verändert
- Will man nun effiziente Funktionsaufrufe, muss man auf diese Effekte achten!
- Bei der Verwendung von Pointern wird man durch das Schreiben von * an die Seiteneffekte erinnert

Instanzen / Pointer / Referenzen

- Variablendeklarationen

```
int i;
double d = 53.217;
```

- Pointer werden mit * bezeichnet

```
int *pInt;          // A pointer to an integer
double *pDbl = &d; // A pointer to double d
```

- Referenzen werden mit & bezeichnet

```
int &intRef = i;    // A reference to an integer
```

- Diese Symbole haben verschiedene Bedeutungen

```
int *pInt = &i;     // Here & means address-of
int j = *pInt;      // Here * means dereference
```

C++ Leerzeichen

- Die folgenden Kommandos sind alle äquivalent:

```
int *p;          // Space before *
int* p;          // Space after *
int * p;         // Space before and after
```

- Gleiches gilt für Referenzen

```
Point &p;        // Space before &
Point& p;        // Space after &
Point & p;       // Space before and after &
```

- Leerzeichen am besten vor * und & setzen
- Beispiel:

```
int* p, q;
```

- Von welchem Typ ist `q`?
- q ist ein `int` und kein `int*`
- * ist mit der Variable assoziiert und nicht mit dem Namen des Typs

C++ Der Datentyp `bool`

- Neuer Datentyp in C++: `bool`
- Schlüsselwörter: `true`, `false`
- C++ Vergleichsoperatoren (==, <, >, !=, …) geben `bool` zurück
- `bool` kann nach `int` konvertiert werden
- `true` ➠ 1, `false` ➠ 0
- ints und Pointer können nach bool konvertiert werden
- nonzero ➠ `true`, zero ➠ `false`
- Beispiel:

```
FILE *pFile= fopen("data.txt", "r");
if (!pFile) { /* Complain */}
```

- Kein C++!

```
ifstream in("data.txt");
if (!in.good()) {
 // Complain
}
```

Gliederung

Klassen und Objekte (1)

- Objekte sind eine Paarung von zwei Dingen
  - Zustand: Eine Ansammlung von zusammengehörenden Daten
  - Verhalten: Auf die Daten bezogene Funktionalität
- Objekte = Daten + Code
- Eine Klasse ist eine „Blaupause“ für Objekte
  - Eine Klasse definiert den Zustand und das Verhalten für Objekte
  - Definiert einen neuen Typ in der Sprache
- Eine Klasse besteht aus Mitgliedern (Members)
  - Member-Variablen: Speichern den Zustand einer Klasse
  - Member-Funktionen: Definieren das Verhalten einer Klasse
    - Der Code der Funktion ist in der Implementierung der Funktion
    - Involvieren in der Regel die Member Variablen

Klassen und Objekte (2)

- Man kann mehrere Objekte einer Klasse haben
  - Jedes Objekt hat seine separaten Member-Daten
- Ein Objekt ist eine Instanz einer Klasse
  - D.h. die Bezeichnungen Objekt und Instanz sind analog verwendbar
- Eine Klasse ist **kein** Objekt
- Konstruktoren initialisieren neue Instanzen einer Klasse
  - Können Argumente haben
  - Haben keinen Rückgabewert
  - Der Standard-Konstruktor hat keine Argumente
- Destruktoren
  - Löschen eine Instanz einer Klasse
  - Haben keine Argumente und keine Rückgabewerte
  - Jede Klasse hat genau einen Destruktor

Klassen und Objekte (3)

- Accessors erlaubt es, den internen Zustand einer Klasse abzurufen
  - Bieten also die Kontrolle, wie und wann Daten ausgelesen werden
- Mutators erlauben es, den internen Zustand von Klassen zu verändern
  - Bieten also die Kontrolle, wie und wann Daten verändert werden
- Der Designer / Entwickler einer Klasse
  - ...entscheidet, wie die Klasse implementiert wird
  - ...entscheidet, welche Funktionalität nach außen sichtbar ist
- Der Benutzer / Klient einer Klasse
  - ...denkt nicht über die Implementation der Klasse nach
  - ...soll die Funktionalität nur benutzen

Abstraktion

- Versteckt die Implementationsdetails vor dem Benutzer
  - Benutzer sollen sich nicht um diese Details kümmern!
  - Benutzer sollen sich auf das konzentrieren, was sie wirklich tun wollen
- Zustandsübergänge sind kontrolliert
  - Benutzer kann nicht eigenständig Zustände ändern
  - Member-Funktionen garantieren valide / erlaubte Zustände
- Implementation von Klassen kann sich ändern
  - Funktionalität kann erweitert werden
  - Interne Representation kann sich ändern
- … solange, wie das extern sichtbare Verhalten gleich bleibt

Klassendeklaration

- Die Klassendeklaration definiert den sichtbaren und den versteckten Teil einer Klasse
- Drei Zugriffsdefinitionen in C++:
  - `public` jeder kann darauf zugreifen
  - `private` nur die Klasse selbst darf darauf zugreifen
  - `protected` ... bekommen wir später
- Der Standardzugriff ist `private`
- Anderer Programmcode kann es `public` deklarierte Funktionen / Members benutzen
- C++ macht einen Unterschied zwischen der Deklaration einer Klasse und deren Implementierung
- Die Deklaration ist in einer Header-Datei, Endung `.h` oder `.hh`
- Die Implementation ist in den `.cc` bzw. `.cpp` Dateien

C++-Beispielklasse (1)

- Der Benutzer fügt die Header-Datei in seinen Code ein
- Die Implementierung findet der Linker
- Beispielklasse für die Repräsentation eines Punktes (`Point.hh`):

```
// A 2D point class!
class Point {
  double x_coord, y_coord;    // Data-members
(private)
public:
  Point();                    // Constructors
  Point(double x, double y);
  ~Point();                   // Destructor
  double getX();              // Accessors
  double getY();
  void setX(double x);        // Mutators
  void setY(double y);
};
```

C++-Beispielklasse (2)

- Implementierung der Klasse in `Point.cc`

```
#include "Point.hh"

// Default (no-arg) constructor
Point::Point() {
  x_coord = 0;
  y_coord = 0;
}

// Two-argument constructor -sets point to (x, y)
Point::Point(double x, double y) {
  x_coord = x;
  y_coord = y;
}

// Cleans up a Point instance.
Point::~Point() {
  // no dynamic resources, so doesn't do anything
}
```

C++-Beispielklasse (3)

```
// Returns X-coordinate of a Point
double Point::getX() {
  return x_coord;
}

// Returns Y-coordinate of a Point
double Point::getY() {
  return y_coord;
}

// Sets X-coordinate of a Point
void Point::setX(double x) {
  x_coord = x;
}

// Sets Y-coordinate of a Point
void Point::setY(double y) {
  y_coord = y;
}
```

C++-Beispielklasse (4)

- Jetzt können wir einen neuen Typ benutzen!

```
#include "Point.hh"
...
Point p1;              // Calls default constructor
Point p2(3, 5);        // Calls 2-arg constructor
cout << "P2 = (" << p2.getX()
     << "," << p2.getY() << ")" << endl;
p1.setX(210);
p1.setY(154);
```

- Die private-Variablen sind nicht erreichbar

```
p1.x_coord = 452; // Compiler reports an error.
```

- Keine Klammen um den leeren Kontruktor benutzen

```
Point p1();          // This declares a function!
```

C++-Beispielklasse (5)

- In der Klasse `Point` macht der Destruktor nichts

```
// Cleans up a Point instance.
Point::~Point() {
// no dynamic resources, so doesn't do anything
}
```

- Point allokiert keine Resourcen dynamisch
- Statische Resourcen werden wie in C automatisch freigegeben
- In diesen Fall ist es nicht notwendig einen Destruktor zu definieren
- Aber: Falls die Klasse dynamisch Speicher allokiert, muss ein Destruktor geschrieben werden