C++ - Eine kurze Übersicht

Daniel Wasserrab

Lehrstuhl für Programmierparadigmen
Universität Karlsruhe

8. Mai 2008

C++

- Entwickelt Anfang der 80er von Bjarne Stroustrup
- Beeinflusst von SIMULA-67
- 1985: Cfront, erster kommerzieller Compiler
- Standardisiert von der ISO 1998
- dazwischen zahllose Erweiterungen

Eigenschaften

- Konzipiert als “C mit Klassen”
- Multiparadigmen-Sprache:
  - Prozedurale Programmierung
  - Modulare Programmierung
  - Strukturierte Programmierung
  - Objektorientierte Programmierung
  - Generische Programmierung
- Maschinennahe Programmierung möglich

Pointer

- Java Referenz == C++ Pointer
- `A*` ist ein Pointer auf `A`
- Zugriffe auf Members mit `->`
- Auch Pointer auf Basisdatentypen und Pointer
- Löschen eines Objektes mit `delete`

```
class A {
public: void f() { }
};
...
A* a=new A();
a->f();
delete a;
a->f(); // undefiniert
```

“echte” Objekte

- Objekt wird bei Deklaration angelegt
- Objekt wird am Ende des Blockes gelöscht
- Zugriffe auf Members mit .
- Speicher für Objekt auf dem Stack

```
class A {
public: A(int a) { }
        void f() { }
};
...
{
  A a(3); // Konstruktor
  a.f();
}
a.f(); // Compiler-Fehler
```

Pointer und Objekte

- &a erzeugt einen Pointer auf a
- *a liefert den Inhalt von a

```
class A {
public: void f() { }
};
...
A* ap;
{
  A a;
  ap=&a;
  ap->f(); a.f();
  (*ap).f();
}
ap->f(); // undefiniert
```

Referenzen

- Alias für ein Objekt
- `A&` ist eine Referenz auf `A`
- Referenzen müssen initialisiert werden!
- Bei Parametern: Call-by-reference

```
class A { public: int x; };
A a1; a1.x=5; A a2; a2.x=6;
A& p; // Compiler-Fehler

A& r=a1; cout << r.x << endl; // 5

r=a2; // Objekt-Kopie!
cout << r.x << " " << a1.x << endl; // 6 6

r.x=7;
cout << a1.x << " " << a2.x << endl; // 7 6
```

Destruktoren

- Automatischer Aufruf vor Zerstörung des Objektes
- geeignet zum Resourcen-Management

```
class A {
public: A() { cout << "hello, "; }
        ~A() { cout << "!"; }
        who(char* who) { cout << who }
};
...
{
  A a;
  a.who("Harry");
}
```

$\Rightarrow$ hello, Harry!

Speicherverwaltung

- Keine eingebaute Garbage-Collection!
- Manuelle Speicherverwaltung: `new` und `delete`
- Der Programmierer hat dafür zu sorgen, dass nach dem Löschen keine Pointer auf das Objekt mehr verwendet werden!
- aber: die Art der Speicherverwaltung ist änderbar
  - Auto-Pointer
  - Reference-Counting
  - Garbage-Collection als Library erhältlich

Mehrfachvererbung

- Eine Klasse kann beliebig viele Oberklassen haben

```
class A {
public:
    void f();
    void g();
};

class B {
public:
    void g();
    void h();
};

class C : public A, public B {
};
...
C c;
c.f(); // ruft A::f() auf
c.h(); // ruft B::h() auf
c.g(); // Compilerfehler
```

Methoden-Deklarationen in Klassen

- `void f() ...`
- `void f();`
  Forward-Deklaration, der Rumpf ist woanders definiert
- `static void f() ...`
- `virtual void f() ...`
  Virtuelle Methode, kann in Unterklasse überschrieben werden
- `virtual void f()= 0;`
  Abstrakte Methode, muss in Unterklasse überschrieben werden
- `void f()const ...`
  Methode kann keine Instanzvariablen ändern

Zugriffsrechte

- Zugriffsrechte werden ggf. für mehrere Members gesetzt
- Innerhalb von Klassen: `private` ist default
- `protected` erweitert nur auf Unterklassen
- Mit `friend` können Ausnahmen gemacht werden
- bei Vererbung: Sichtbarkeit der Vererbung

```
class A {
public: void f() { ... }
};
class B : private A {
public: void g() { f(); ... }
};
...
B b;
b.g();
b.f(); // Compiler-Fehler
A* a = &b; // Compiler-Fehler
```

Operator Overloading

- Für eigene Klassen können die Standard-Operatoren definiert werden:
  - `bool operator ==(A a1, A a2);`
  - `bool operator ==(A a, B b);`
  - `C operator *(A a, B b);`
  - `ostream& <<(int, ostream &);`
- erhöht richtig eingesetzt die Lesbarkeit:

```
m["Alter"]["Harry"]=43;
```

- Sogar Cast- und Zugriffs-Operatoren können überladen werden:

```
class B {
public:
    A* operator ->() { ... }
};
```

Standard-Bibliothek

- Ursprünglich nur C-Bibliothek
- Heute: STL (Container) + Streams (I/O)
- Keine Klassenbibliothek
- Weitere Bibliotheken in der Normung
- Prinzipiell jede C Bibliothek einbindbar
- Für system-nahe Programmierung C-Bibliothek weiter notwendig

Preprocessor

Jeder Quellcode wird vor dem Compilieren durch den "Preprocessor" geschickt:

- `#include <filename.hpp>`
  Fügt den Text aus der Datei filename ein.
- `#define MACRO value`
  Ersetzt jedes Vorkommen von `MACRO` durch `value`
- `#define MAX(m,n)((m)>(n)?(m):(n))`
  Macros können Argumente haben
- `#if BEDINGUNG`
  ...
  `#endif`
  Bedingte Kompilation: Text dazwischen wird nicht eingesetzt, wenn `BEDINGUNG` falsch ist

"Hello World" nach Präprozessor: 30000 LOC.

Headerdateien

- bestimmte Inhalte einer Datei in Headerdatei deklariert:
  - Makros und symbolische Konstanten
  - Signatur von von außen aufrufbaren Methoden
  - evtl. globale Konstanten
- Konvention: Endung .hpp, .hh oder .h (.h auch für C-Header)
- Einbinden in Datei mittels `#include <filename.hpp>` oder `#include "filename.hpp"`
- Standardbibliothek-Header ohne Endung: `#include <stdio>`
- Header-Dateien verwandt mit Java Interfaces
- Verhindern von Mehrfacheinfügungen eines Headers:

```
/* This is file foo.hpp */
#ifndef _FOO_HPP_
#define _FOO_HPP_
... Definitionen von foo.hpp ...
#endif
```

Namespaces

- In großen Projekten ($> 10^6$ LOC) sind Namenskonflikte fast unvermeidlich
- Typisch: Libraries von unterschiedlichen Herstellern
- Namenskonventionen helfen nur bedingt
- zu lange Bezeichnernamen machen Code unübersichtlich
- Java missbraucht Klassen (e.g. `java.lang.Math`)

Namespaces (Syntax)

```
#include <iostream>
  // alles in namespace "std"

... std::cout << "..." << std::endl; ...

using namespace std;
// importiert alles aus "std" in den
// aktuellen namespace

... cout << "..." << endl; ...

namespace bar {
   using std::cout;
   // importiert ein Element in bar
}

... bar::cout << "..." << std::endl; ...
```

Namespaces (Anwendung)

```
tree.hpp:                       baeume.hpp:

class Baum { ... };             class Baum { ... };
class Knoten { ... };           class Eiche { ... };
class Blatt { ... };            class Buche { ... };

namespace Datenstrukturen {
#include "tree.hpp"
}
namespace Wald {
#include "baeume.hpp"
}
...
   Wald::Baum b;
   Datenstrukturen::Baum t;
...
```

Templates

- gleiche Grundidee wie Generics
- keine Angabe von Typschranken nötig oder möglich
- Anstelle von Klassen auch Werte und Basisdatentypen als Parameter möglich
- Default Parameter

```
template<int rows, int cols=rows,
         class content=double>
class Matrix { ... };
...
Matrix<3,4,int> m;
Matrix<3,4> n;
Matrix<3> o;
```

Kein super()

- Oberklassenkonstruktoren: spezielle Syntax
- Methodenaufrufe: explizite Auswahl der Methode

```
class C : public A, public B {
  C(int a, int b)
    :A(a),
     B(b)
  {}
  int f() {
    return A::f() + B::f();
  }
};
```

Unterschiede zu Java

- Kein Reflection
- Kein dynamisches Laden
- `assert`
  - Keine Exception, sondern Programmabbruch
  - Können nur beim Compilieren entfernt werden
- Exceptions
  - Alle Objekte, Integers, Zeichenketten, ... können geworfen werden
  - Spezifikation ist optional
  - Bei erfolgter Spezifikation:
    - Programmabbruch, wenn falsche Exception geworfen wird
    - Alternativ: Statt dessen wird `std::bad_exception` geworfen

Compiler und Linker

- Compiler erzeugt aus Quelltext Objekt-File
- Objekt-File ist i.A. plattform- und Compiler-abhängig
- Linker erzeugt Executable aus Objekt-File(s)
- Objekt-File(s) werden in Libraries zusammengefasst
- Aus dem Objekt-File ist kein Quelltext regenerierbar
- Bei Änderung einer Klasse Neukompilation des gesamten benutzten Codes notwendig
- Global optimierende Compiler:
  Objekt-File nur geparster Source-Code

Zusammenfassung

- Multiparadigmen-Sprache
- Maschinennahe Programmierung möglich
- Kompliziert
- Viele Möglichkeiten
- Viele Möglichkeiten, sich in den Fuß zu schießen
- Sehr relevant in der Industrie