SAARLAND UNIVERSITY
COMPUTER SCIENCE

C/C++-Programmierung

Rückblick, C++

Sebastian Hack
Christoph Mallon

(hack|mallon)@cs.uni-sb.de

Fachbereich Informatik
Universität des Saarlandes

Wintersemester 2009/2010

Ausdrücke und Anweisungen

SAARLAND UNIVERSITY
COMPUTER SCIENCE

() [] − > . ! ++ −− + − (*type*) ∗ & sizeof new delete −>∗ .∗ ∗ / % + − << >> < <= > >= == ! = & ^ | && || ? : = + = − = ∗ = / = % = <<= >>= & = ^= | = throw , const_cast dynamic_cast reinterpret_cast static_cast

break case catch continue default do else for goto if return switch try while

Präprozessor

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
#define SIX     1 + 5
#define NINE    8 + 1
#define ANSWER  SIX * NINE
```

Zeiger, Speicherverwaltung

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
X* p = malloc(sizeof(*p) * n);
free(p);
```

Zeichenketten

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
"hello␣world\n"
```

make


```
%make fire?
make: No match.
```

diff, patch

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
%diff -u a b
--- a
+++ b
@@ -1,7 +1,7 @@
-falsch
+richtig
```

Versionsverwaltung

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
%svn ci -m "Make␣everything␣really␣kaputt"

%svn blame main.c
```

Unterschiede zwischen C und C++

- Sprachunterstützung für Objektorientierung
- Strengere Typprüfung
- Referenzen
- Schablonen (Templates)
- Ausnahmen (Exceptions)
- Namensräume
- C++ ist fast Obermenge von C

Strengere Typprüfung


```
int    i;
void* v = &i;
int*   p = v;  // Fehler in C++

// C:   Parameter unspezifiziert
// C++: Keine Parameter
void f();
void g(void); // Weiterhin erlaubt in C++

f(23, "bla"); // OK in C, Fehler in C++
g(23);        // Fehler in C und C++

int i = 23.42; // Allerdings immernoch erlaubt
```

Verbindung von C und C++

C99 §6.10.9:5:

> *The implementation shall not predefine the macro* `__cplusplus`*, nor shall it define it in any standard header.*

C++ §16.8:1:

> *The name* `__cplusplus` *is defined to the value* `199711L` *when compiling a C++ translation unit.*

Beispiel

Häufigste Verwendung: Unterbinden von name mangling um C-Funktionen aufrufen zu können.

```
#ifndef HEADER_H
#define HEADER_H

#ifdef __cplusplus
extern "C" {
#endif

void f(void):

#ifdef __cplusplus
}
#endif

#endif HEADER_H
```

Klassen

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
class X;

class X { X(); int i; };

X::X() : i(42) {}
```

- Erweiterung von von `struct` um Methoden, Vererbung und Sichtbarkeitsbeschränkungen
- Deklaration und Implementierung sowohl von Klasse selbst als auch Methoden trennbar
- Syntaktische Vereinfachung gegenüber C: `typedef class X X;` wird automatisch gemacht (auch für `struct`, `union` und `enum`)

Sichtbarkeit

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
class X
{
  public:
    X();
    void everybody();
  protected:
    void us();
  private:
    void me();
    int i;
};
```

- Besser: Verwendbarkeitsbeschränkungen
- `public`: Jeder darf darauf zugreifen
- `protected`: Nur Klasse und ihre Kinder
- `private`: Nur Klasse selbst
- Betrifft alle folgenden Elemente
  $\longrightarrow$ nicht wie Java nur ein Element

Freunde

§11.4

```
class X
{
  private:
    int i;

  friend void f(X*);
};

void f(X* x) { x->i = 42; }
void g(X* x) { x->i = 23; } // Fehler!
```

- Klasse kann andere Klassen und Funktionen als `friend` deklarieren
- Diese Klasse/Funktion kann dann auf die `protected` und `private` Attribute/Methoden zugreifen

Only friends may touch your privates.

Objektlebenszyklus

- Beschaffung von Speicher
- Konstruktor
  $\longrightarrow$ Etablierung der Objektinvariante
- Benutzung
  $\longrightarrow$ Methoden erhalten Objektinvariante
- Destruktor
  $\longrightarrow$ Resourcenfreigabe
- Speicherfreigabe

Vererbung

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
class X {};
class Y {};
class Z : public X, public Y {}
```

- Klassen können von keiner, einer oder mehreren Klassen erben
- Sichtbarkeit gibt an, wie Attribute/Methoden der Oberklasse und Oberklasse selbst von außen zugegriffen werden können

class vs. struct


```
struct X      { void f() {}          int i; };
class  Y : X { void g() { f(); i = 42; } }; // OK
class  Z : Y { void h() {      i = 23; } }; // Fehler
```

- Fast kein Unterschied
- Attribute von `class` sind normalerweise `private`
- Attribute von `struct` sind normalerweise `public`
- `class` wird normalerweise `private` vererbt
- `struct` wird normalerweise `public` vererbt

Konstruktoren


```
struct X {
  X()       { /* ... */ }
  X(int i) { /* ... */ }
};

struct Y : X { Y(int i) : X(i + 1) { /* ... */ } };
```

- Aufgabe: Etablierung der Objektinvariante
- Jede Klasse muss mindestens einen Konstruktor besitzen.
  $\longrightarrow$ Wenn nicht explizit, werden automatisch mehrere erzeugt.
- Ein Konstruktor jeder Oberklasse muss aufgerufen werden.
  $\longrightarrow$ Wenn nicht explizit, dann wird automatisch ein parameterloser Konstruktor der Oberklasse aufgerufen.

Destruktoren


```
struct X { ~X() { /* ... */ } };
```

- Aufgabe: Freigabe von Resourcen
- Bei Aufruf des Destruktors muss der dynamische Typ des Objekts bekannt sein
- Achtung: Es dürfen (fast) niemals Ausnahmen aus einem Destruktor entkommen
- Nach eigenem Destruktor werden automatisch Destruktoren aller aggregierten Objekte und Destruktoren der Oberklasse(n) aufgerufen.

RAII

- Ressource Acquisition Is Initialisation
  $\longrightarrow$ wenig aussagekräftiger Name
- Idee: Binde Resource/Aktion an Lebenszeit eines Objekts
- Destruktoraufruf geschieht bei lokalen Variablen und aggregierten Objekten automatisch, daher keine explizite Ressourcenfreigabe notwendig
- Beispiel: Datei
  Konstruktor: Datei öffnen
  Destruktor: Datei schließen

Vergleich

Vergleich von Ressourcenverwaltung in

- C
- Java
- C#
- C++

```c
typedef struct Resource { /* ... */ } Resource;

Resource* x = alloc_resource();

/* do stuff */

if (some_error)
  return; // Ressourcenleck!

/* do stuff */

free_resource(x);
```

- Explizite Freigabe auf jedem Pfad notwendig
- Viel Duplikation, sehr fehleranfällig

Java


```
class Resource {
  Resource()  { /* Allokationen */ }
  void free() { /* Freigaben */ }
}

Resource x = new Resource();
try {
  /* do stuff */
  if (some_error)
    return; // finally wird automatisch ausgefuehrt
  /* do stuff */
  // hier ebenfalls
}
finally {
  x.free();
}
```

- Explizite Freigabe nur noch einmal pro Verwendung notwendig
- Dennoch Duplikation, aber schon weniger fehleranfällig
  $\rightarrow$ Typischer Fehler: Schließen von Datei vergessen, geschriebenes geht verloren

```
class Resource : IDisposable {
  Resource()                    { /* Allokationen */ }
  void IDisposable.Dispose() { /* Freigaben */ }
}

using (Resource x = new Resource()) {
  /* do stuff */

  if (some_error)
    return; // x.Dispose() wird automatisch aufgerufen

  /* do stuff */
  // hier ebenfalls
}
```

- Keine explizite Freigabe: Methode Dispose() wird automatisch aufgerufen
- Keine Duplikation
- Funktioniert nicht rekursiv: in Dispose() müssen aggregierte Resourcen manuell freigegeben werden

```
struct Resource {
  Resource()  { /* ... */ }
  ~Resource() { /* ... */ }

  private: SubResource sub;
};

{ Resource x;
  /* do stuff */
  if (some_error)
    return; // x wird automatisch zerstoert
  /* do stuff */
  // hier auch
}
```

- Ebenfalls keine explizite Freigabe: lokales Objekt wird automatisch zerstört
- Funktioniert rekursiv: Aggregiertes Objekt `sub` wird automatisch mitzerstört