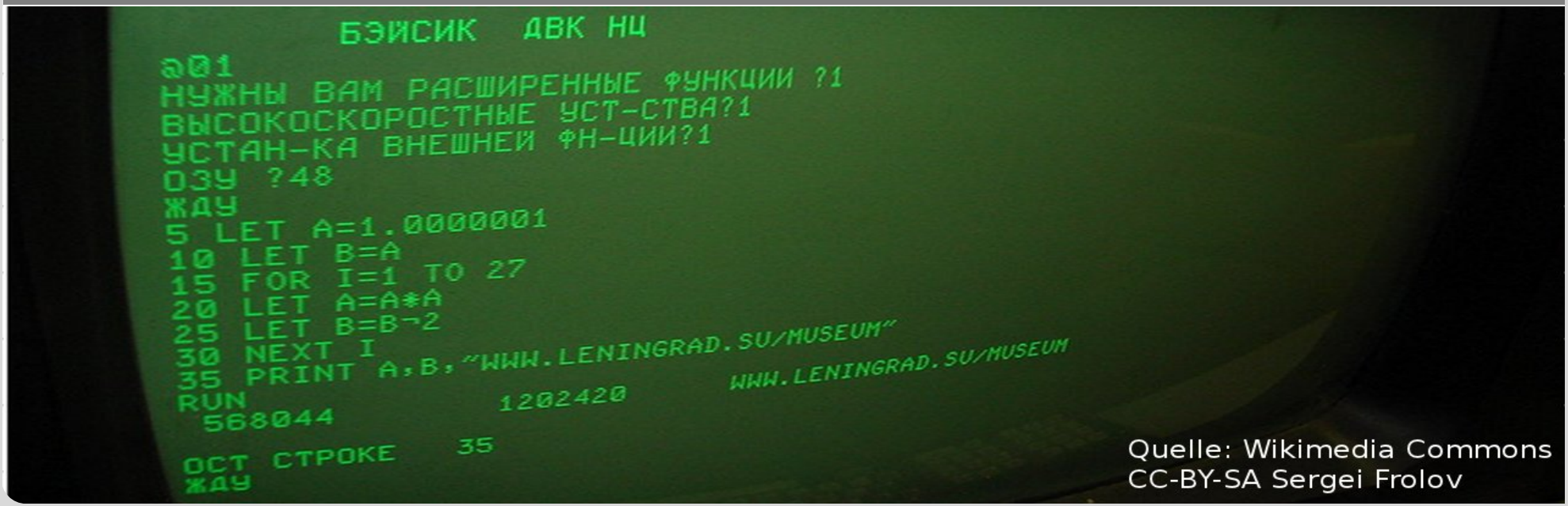
Karlsruher Institut für Technologie

C++ Workshop

5. Block, 01. Juni 2012

Markus Jung, Robert Schneider | 2. Juni 2012



Quelle: Wikimedia Commons
CC-BY-SA Sergei Frolov

Gliederung

1. Unions und enums
   - enum
   - union

Definition

Enumeration declarations (Standard, 7.2)

- Eine `enum` ist eine Aufzählung (enumeration) von benannten Konstanten, den *enumerators*.
- Eine `enum` ist ein eigenständiger Typ.
- Die Namen der Konstanten liegen auf *der selben Ebene* wie die `enum` selbst – sie sind nicht geschachtelt in die enum!

Definition

Enumeration declarations (Standard, 7.2)

- Eine `enum` ist eine Aufzählung (enumeration) von benannten Konstanten, den *enumerators*.
- Eine `enum` ist ein eigenständiger Typ.
- Die Namen der Konstanten liegen auf *der selben Ebene* wie die `enum` selbst – sie sind nicht geschachtelt in die enum!

Syntax

```
1  enum MyEnum { MY_CONST_0, A, B, C };
2
3  MyEnum myVar;
4
5  myVar = A;
6  if(B == myVar) { /* ... */ }
```

Wert eines enumerators

enums werden oft als Ersatz für Ganzzahl-Konstanten (integer numbers) verwendet

ABER: Das ist so nicht ganz korrekt und kann zu Fehlern führen!

Wert eines enumerators

enums werden oft als Ersatz für Ganzzahl-Konstanten (integer numbers) verwendet

ABER: Das ist so nicht ganz korrekt und kann zu Fehlern führen!

Wert eines enumerators

Normalfall:

- der erste enumerator hat den Wert 0
- der enumerator n+1 hat einen um 1 größeren Wert als der des enumerators n

Wert eines enumerators

enums werden oft als Ersatz für Ganzzahl-Konstanten (integer numbers) verwendet

ABER: Das ist so nicht ganz korrekt und kann zu Fehlern führen!

Wert eines enumerators

Normalfall:

- der erste enumerator hat den Wert 0
- der enumerator n+1 hat einen um 1 größeren Wert als der des enumerators n

Manuelles eingreifen: Explizites zuweisen eines Wertes an einen enumerator

```
1  enum MySecEnum {
2          W = 1, X,
3          Y = 0, Z
4  };
```

Integral promotion

Underlying type (Standard, 7.2:5)

- Der *zugrunde liegende Typ* einer enum ist ein Ganzzahl-Typ, der jeden enumerator-Werte dieser enum repräsentieren kann.
- Der underlying type soll nicht größer sein als `int`, es sei denn, `int` reicht nicht aus.

Integral promotion

Underlying type (Standard, 7.2:5)

- Der *zugrunde liegende Typ* einer enum ist ein Ganzzahl-Typ, der jeden enumerator-Werte dieser enum repräsentieren kann.
- Der underlying type soll nicht größer sein als `int`, es sei denn, `int` reicht nicht aus.

Integral promotion of enumerators (Standard, 4.5, 7.2:8)

- Der Wert enumerators wird wo nötig zu einem Ganzzahl-Typ umgewandelt.
- Der Ganzzahl-Typ ist der erste aus folgender Liste, der alle Werte des underlying type repräsentieren kann: `int`, `unsigned int`, `long`, `unsigned long`

Integral promotion: Beispiel

Beispiel

```
1  short s = A;                          // FEHLER!
2  int i = A;                            // OK
3  long l = A;                           // OK, int -> long conversion
4  if(42 == A) { /* ... */ }             // OK, 42 is int
```

Integral promotion: Beispiel

Beispiel

```
1  short s = A;                          // FEHLER!
2  int i = A;                            // OK
3  long l = A;                           // OK, int -> long conversion
4  if(42 == A) { /* ... */ }             // OK, 42 is int
```

Typsicherheit

```
1  enum MyEnum { MY_CONST_0, A, B, C };
2  enum MySecEnum {
3          W = 1, X,
4          Y = 0, Z
5  };
```

```
1  void myFunc(MyEnum);
2  void myFunc(MySecEnum);
3  void myFunc(int);
4
5  myFunc(A);
6  myFunc(X);
7  myFunc((MyEnum)42);
```

Umgekehrter Weg: conversion

Umgekehrter Weg: Conversion (Standard, 7.2:9)

- Eine Ganzzahl und ein enumerator einer anderen enum kann *explizit* zu einem enumerator *konvertiert* werden.

Umgekehrter Weg: conversion

Umgekehrter Weg: Conversion (Standard, 7.2:9)

- Eine Ganzzahl und ein enumerator einer anderen enum kann *explizit* zu einem enumerator *konvertiert* werden.
- Der entstehende enumerator hat denselben Wert wie das, was konvertiert wurde – wenn es einen enumerator mit solchem Wert in der enum gibt.

Umgekehrter Weg: conversion

Umgekehrter Weg: Conversion (Standard, 7.2:9)

- Eine Ganzzahl und ein enumerator einer anderen enum kann *explizit* zu einem enumerator *konvertiert* werden.
- Der entstehende enumerator hat denselben Wert wie das, was konvertiert wurde – wenn es einen enumerator mit solchem Wert in der enum gibt.
- Ansonsten ist das Resultat undefiniert!

Umgekehrter Weg: conversion

Umgekehrter Weg: Conversion (Standard, 7.2:9)

- Eine Ganzzahl und ein enumerator einer anderen enum kann *explizit* zu einem enumerator *konvertiert* werden.
- Der entstehende enumerator hat denselben Wert wie das, was konvertiert wurde – wenn es einen enumerator mit solchem Wert in der enum gibt.
- Ansonsten ist das Resultat undefiniert!

Beispiel

```
1 MyEnum foo;
2 foo = (MyEnum)2;     // OK, 2 == B
3 foo = (MyEnum)42;    // foo now has an unspecified value!
4 foo = (MyEnum)Y;     // OK, Y == MY_CONST_0
```

Hinweise

- Der underlying type ist *nicht* immer `int`!
- Der underlying type ist abhängig vom Compiler und der target platform!
- enums sind eigene Typen, aber sehr Typ-unsicher aufgrund der integral promotion
- enumerator-Namen werden nicht in die enum geschachtelt

Was ist eine union?

- Bekannt: `class` und `struct`
  - Fassen Daten (Member) und Methoden zusammen
  - Das ganze ist (mehr als) die Summe seiner Teile

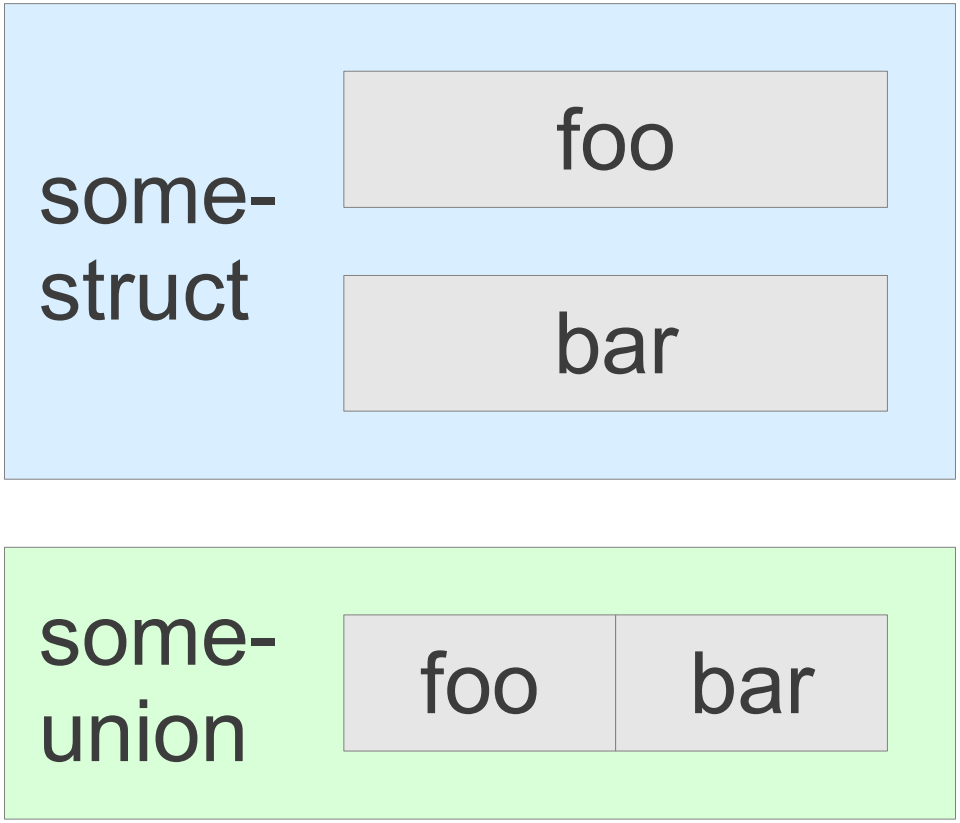
Was ist eine union?

- Bekannt: `class` und `struct`
  - Fassen Daten (Member) und Methoden zusammen
  - Das ganze ist (mehr als) die Summe seiner Teile
- Neu: `union`
  - NICHT die Summe aller Teile, KEIN Aggregat
  - Alle deklarierten Member landen an der gleichen Stelle im Speicher
  - Ein Union kann Daten verschiedenen Typs aufnehmen
  - Aber zu jedem Zeitpunkt nur von einem Typ
  - Nutzung wie bei `class` und `struct`: `myUnion.foo = 42`
  - Möglich: Methoden, `private`, `protected`, `public`, Konstruktoren und Destruktoren
  - Verboten: Jede Form von Vererbung

Unions: Vergleich mit Structs

```
struct somestruct {
        int foo;
        long bar;
};

union someunion {
        int foo;
        long bar;
};
```



Unions: Motivation

- Ursprung: Plain old C, Speicherung verschiedener Typen in einer Variablen
  - „Polymorphismus für Arme“

Unions: Motivation

- Ursprung: Plain old C, Speicherung verschiedener Typen in einer Variablen
  - „Polymorphismus für Arme“
- In C++: Auch nur ein Container für verschiedene Typen ...

Unions: Motivation

- Ursprung: Plain old C, Speicherung verschiedener Typen in einer Variablen
  - „Polymorphismus für Arme“
- In C++: Auch nur ein Container für verschiedene Typen ...
- Type Punning: `foo_t` rein, `bar_t` raus
  - Wird NICHT vom Standard gedeckt!
  - Compilerabhängig

Anonyme Unions

- Unions die selbst nicht in Erscheinung treten/transparent sind

```
1  void foo() {
2          union /*no-name*/
3          {
4                  int a;
5                  long b;
6          };
7
8          a = 23;
9          b = 21 * 2;
10 }
```

```
1  void bar() {
2          union MyUnion
3          {
4                  int a;
5                  long b;
6          };
7          MyUnion wuppdi;
8
9          wuppdi.a = 23;
10         wuppdi.b = 21 * 2;
11 }
```

Unions: Anwendungsbeispiel Variant

```
struct variant {
        enum TYPE {
                T_BOOL,
                T_INT,
                T_DOUBLE
        };

        TYPE m_type;

        union {
                bool b;
                int i;
                double d;
        };
};
```

2 Praxis

Praxis!

- Aufgabe 1: Ein einfaches Schachspiel

`https://github.com/kit-cpp-workshop/workshop-ss12-05`

Aufgabenbeschreibungen und Hinweise: Siehe `README.md`