SAARLAND UNIVERSITY
COMPUTER SCIENCE

C/C++-Programmierung

Aufzählungen, Zeichenketten

Sebastian Hack
Christoph Mallon

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

Fachbereich Informatik
Universität des Saarlandes

Wintersemester 2009/2010

Aufzählungen

enumerations, C: §6.7.2.2, C++: §7.2

- Symbolische Namen für ganzzahlige Konstanten
- Jede Aufzählung ist ein eigener Typ
- Dahinterliegender Typ ist implementierungsabhängig
  → Muss ein Ganzzahltyp sein
- Typname wieder nur mit Präfix `enum` verwendbar
- Unterschied bei Typ von Aufzählungswerten
  - C: `int`
  - C++: Aufzählungstyp
- Aufzählungen können implizit in normale Zahlen umgewandelt werden
- Zahlen können implizit in Aufzählungen umgewandelt werden (nicht C++)

Beispiel: enum

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
enum Meme {
  FIRST_POST,              /* Erster Eintrag ohne
                            * expliziter Wert: 0 */
  EPIC_FAIL     =   -1,
  FACEPALM      =   -1, // Duplikate moeglich
  LEET          = 1337,
  NINETHOUSAND = 9000,
  OVER9000                 // voriger Eintrag + 1: 9001
};
```

Beispiel: enum als Bitflags

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
typedef enum AccessPermissions
{
  USER_READ   = 0400,
  USER_WRITE  = 0200,
  USER_EXEC   = 0100,
  USER_RXW    = USER_READ | USER_WRITE | USER_EXEC,
  GROUP_READ  = 0040,
  GROUP_WRITE = 0020,
  GROUP_EXEC  = 0010,
  GROUP_RWX   = 0070,
  OTHER_READ  = 0004,
  OTHER_WRITE = 0002,
  OTHER_EXEC  = 0001,
  OTHER_RWX   = 0007, // Komma hier seit C99 erlaubt
} Perm;

Perm x = USER_READ | USER_WRITE; // OK in C, nicht C++
```

Beispiel: Arbeiten mit Bitflags

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
x |=  USER_EXEC;    // Bit setzen     (nicht C++)
x ^=  GROUP_WRITE;  // Bit kippen     (nicht C++)
x &= ~OTHER_RWX;    // Bits loeschen  (nicht C++)

if (x & USER_READ) // Bit testen
  printf("Besitzer darf lesen");
```

Zeichenketten

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- Zeichenkettenverarbeitung kein Sprachelement, wird mittels Bibliotheksfunktionen realisiert
- Per Konvention sind Zeichenketten NUL-terminiert (`'\0'`)
  $\longrightarrow$ Beschränkung: Zeichenketten können kein NUL enthalten
  $\longrightarrow$ Manchmal ASCIIZ genannt
- Nur etwas syntaktischer Zucker für Zeichenkettenliterale

Zeichenkettenliterale

String Literals

```
"Hallo\n"
```

- Sind einfach Reihungen von `char`
- NUL wird am Ende automatisch ergänzt
- Fast äquivalent zu

```
char eindeutiger_name[] =
  { 'H', 'a', 'l', 'l', 'o', '\n', '\0' };
```

- Elementtyp ist `char`, aber Inhalt darf nicht verändert werden
  → Sonst verhalten undefiniert
  → In C++ Elementtyp `char const` und veraltete Umwandlung nach `char*` erlaubt
- Gleiche Zeichenkettenliterale können (aber müssen nicht) vom Übersetzer zusammengelegt werden

Zeichenkettenliterale und Reihungen/Zeiger

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
char  a[] = { 'H', 'a', 'l', 'l', 'o', '\n', '\0' };
char  a[] = { "Hallo\n" };
char  a[] = "Hallo\n"; // {} kann weggelassen werden
// Andere Bedeutung!
char* p   = "Hallo\n";
```

- Syntaktischer Zucker: Reihungen von `char` können mit Zeichenkettenliteralen initialisiert werden

Zeichenkettenverarbeitung

§7.21.2.3, §7.21.3.1


```
char*  strcpy(char* dst, char const* src);
char*  strcat(char* dst, char const* src);
size_t strlen(char const*);
```

- Header `<string.h>`
- Zeichketten müssen immer NUL-terminiert sein
- Generell: Ziel und Quelle dürfen sich nicht überlappen, sonst Verhalten undefiniert
- Bei unvorsichtiger Benutzung Pufferüberläufe möglich
  $\longrightarrow$ z.B. Benutzereingaben
- Alle Namen beginnend mit `str` gefolgt von einem Kleinbuchstaben sind reserviert

strncpy(): Sichere Alternative?

§7.21.2.4


```
char* strncpy(char* dst, char const* src, size_t n);
```

- Kopiert nicht mehr als `n` Zeichen nach `dst`
- Wenn `src` kürzer, wird Rest mit NUL-Zeichen gefüllt
- Achtung: NUL-Terminierung ist nicht garantiert!

strncat(): Sichere Alternative?

§7.21.3.2

```
char* strncat(char* dst, char const* src, size_t n);
```

- Hängt nicht mehr als `n` weitere Zeichen an bestehende Zeichenkette `dst` an
- Wenn `src` kürzer, wird Rest mit NUL-Zeichen gefüllt
- Achtung: Bedeutung von `n` abstrus – man muss wissen wie lang die Zeichenkette an dst bereits ist!

```
char buf[128];
strncpy(buf, "Hallo␣", sizeof(buf));
if (buf[sizeof(buf) - 1] != '\0')
abort(); // Buffer too small
// !!!
strncat(buf, "Welt!",  sizeof(buf) - strlen(buf));
if (buf[sizeof(buf) - 1] != '\0')
abort(); // Buffer too small
```

Sichere Alternative: strl*()

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
size_t strlcpy(char* dst, char const* src, size_t n);
size_t strlcat(char* dst, char const* src, size_t n);
```

- `n` ist Größe des Zielpuffers
- Es wird nie eine Zeichenkette von mehr als `n - 1` Zeichen erzeugt
- Wird immer NUL-terminiert
- Gibt Länge der Zeichenkette zurück, die versucht wurde zu erzeugen
  $\longrightarrow$ Einfach prüfbar, ob Puffer zu klein ist
- Nachteil: Leider nicht normiert

strcmp(): Lexikographischer Vergleich

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
int strcmp(const char *s1, const char *s2);
```

- Gibt Zahl kleiner/gleich/größer 0 zurück, wenn `s1` kleiner/gleich/größer `s2`
- Vergleich abhängig von gewählter Lokalisierung
  → Normalfall `"C"`
  → Heutzutage effektiv ASCII-betisch

Variadische Funktionen

§6.7.5.3:9, §7.15

```c
#include <stdarg.h>

void f(int x, ...) // Ellipse (Auslassung)
{
  va_list ap;
  va_start(ap, x);
  /* ... */
  TYPE a = va_arg(ap, TYPE);
  /* ... */
  va_end(ap);
}
```

- `va_start`: zweites Argument muss letzter formaler Parameter sein
- `va_arg`: Holt nächstes variadisches Argument
  - Angegebener Typ muss mit dem Typ des übergebenen Arguments übereinstimmen
  - Bei Ganzzahlentypen: mindestens `int`
  - Bei Gleitkommazahlentypen: mindestens `double`

*printf()


```
int printf(char const* fmt, ...);
int fprintf(FILE*, char const* fmt, ...);
int sprintf(char* buf, char const* fmt, ...);
int snprintf(char* buf, size_t n, char const* fmt, ...);
```

- Funktionsfamilie zur Formatierung von Text
- `printf()`: Ausgabe auf Bildschirm
- `fprintf()`: Ausgabe in Datei
- `sprintf()`: Ausgabe in Zeichenkette + NUL
- `snprintf()`: Mit Größenangabe des Puffers, es werden nicht mehr als $n - 1$ Zeichen + NUL geschrieben
- Rückgabewert: Je Anzahl der geschriebenen Zeichen
  - Bei `s*printf()` ohne NUL
  - Bei `snprintf()` wenn Puffer groß genug wäre

v*printf()


```
int vprintf(char const* fmt, va_list);
int vfprintf(FILE*, char const* fmt, va_list);
int vsprintf(char* buf, char const* fmt, va_list);
int vsnprintf(char* buf, size_t n, char const* fmt,
              va_list);
```

- Letzter Parameter bereits vorbereitete Argumentenliste statt Ellipse

```
void dbg_printf(char const* fmt, ...)
{
  printf("DEBUG: ");
  va_list ap;
  va_start(ap, fmt);
  vprintf(fmt, ap);
  va_end(ap);
}
```

wchar_t


```c
#include <stddef.h>
#include <stdio.h>
#include <wchar.h> // wchar_t-Funktionen

wprintf(L"Hallo, %lcelt\n", L'W');
```

- Vorangestelltes `L`
- Zeichenvorrat umfasst mindestens den von `char`
- `wchar_t` definiert in `<stddef.h>` für C, eingebauter Typ in C++
- Üblicherweise UCS-2 (16 Bit) oder UCS-4 (32 Bit)
- Verarbeitungsfunktionen haben `w` (`wprintf()`, `swprintf()`) oder `wcs` (`wcscpy()`, `wcslen()`) im Namen
- Achtung: `wchar_t`- und `char`-Funktionen auf selbem Ausgabestrom nicht mischen! (§7.19.2:5)

Verkettung von Zeichenkettenliteralen

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
// Aequivalent
"Hallo, Welt!"
"Hallo," " Welt!"

L"hello, world"
L"hello, wo" L"rld"
// Auch gemischt (ab C99)
L"hell" "o, world"
"hello" L", world"
```

- Ergebnis ist `wchar_t[]`, wenn mindestens ein Teil `wchar_t[]` ist

```
printf("Version:"
#ifndef NDEBUG
  " (DEBUG)"
#endif
  "3.14159");
```