

Verifikation von C-Programmen

Vorlesung 1 vom 23.10.2014: Der C-Standard: Typen und Deklarationen

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

Der C-Standard

C: Meilensteine

- ▶ 1965– 69: BCPL, B; Unix, PDP-7, PDP-11
- ▶ 1972: Early C; Unix
- ▶ 1976– 79: K& R C
- ▶ 1983– 89: ANSI C
- ▶ 1990– : ISO C
 - ▶ 1985: C++

Geschichte des Standards

- ▶ **1978**: Kernighan & Ritchie: *The C Programming Language*
- ▶ **1980**: zunehmende Verbreitung, neue Architekturen (80x86), neue Dialekte
- ▶ **1983**: Gründung C Arbeitsgruppe (ANSI)
- ▶ **1989** (Dez): Verabschiedung des Standards
- ▶ **1990**: ISO übernimmt Standard (kleine Änderungen)
- ▶ **1999**: Erste Überarbeitung des Standards (ISO IEC 9899: 1999)
- ▶ **2011**: Zweite Überarbeitung des Standards (ISO IEC 9899: 2011)

Nomenklatur

- ▶ “Implementation”: Compiler und Laufzeitumgebung
- ▶ “Implementation-defined”: Unspezifiziert, Compiler bestimmt, dokumentiert
 - ▶ Bsp: MSB bei signed shift right
- ▶ “Unspecified”: Verhalten mehrdeutig (aber definiert)
 - ▶ Bsp: Reihenfolge der Auswertung der Argumente einer Funktion
- ▶ “Undefined”: undefiniertes Verhalten
 - ▶ Bsp: Integer overflow
- ▶ “Constraint”: Einschränkung (syntaktisch/semantisch) der Gültigkeit
- ▶ Mehr: §3, “Terms, Definitions, Symbols”

Gliederung

- ▶ §3: Terms, Definitions, Symbols 4 S.
- ▶ §4: Conformance 3 S.
- ▶ §5: Environment 20 S.
 - ▶ Übersetzungsumgebung, Laufzeitumgebung
- ▶ §6: Language 165 S.
 - ▶ Die Sprache — Lexikalik, Syntax, Semantik
 - ▶ Präprozessor
 - ▶ Future language directions
- ▶ §7: Library 238 S.
- ▶ Anhänge
 - ▶ Language syntax summary; Library summary; Sequence Points; Identifiers; Implementation limits; Arithmetic; Warnings; Portability 112 S.

Eine Sprachkritik

- ▶ **Philosophie:** The Programmer is always right.
 - ▶ Wenig Laufzeitprüfungen
 - ▶ Kürze vor Klarheit
 - ▶ Geschwindigkeit ist (fast) alles
- ▶ **Schlechte** Sprachfeatures:
 - ▶ Fall-through bei `switch`, String concatenation, Sichtbarkeit
- ▶ **Verwirrende** Sprachfeatures:
 - ▶ Überladene und mehrfach benutzte Symbole (`*`, `()`¹), Operatorpräzedenzen

¹7 Bedeutungen.

Varianten von C

- ▶ “K&R”: ursprüngliche Version

- ▶ Keine Funktionsprototypen

```
f(x, y)
int x, y;
{
    return x+ y;
}
```

- ▶ ANSI-C: erste standardisierte Version
 - ▶ Funktionsprototypen, weniger Typkonversionen
- ▶ C99, C+11: konservative Erweiterungen

Typen und Deklarationen

Verschiedene Typen im Standard

- ▶ **Object** types, **incomplete** types, **function** types (§6.2.5)
 - ▶ **Basic** types: standard/extended signed/unsigned integer types, floating types (§6.2.5)
 - ▶ **Derived** types: structure, union, array, function types
- ▶ **Qualified Type** (§6.2.5)
 - ▶ `float const * qualified` (qualified pointer to type)
 - ▶ `const float * not qualified` (pointer to qualified type)
- ▶ **Compatible Types** (§6.2.7)
- ▶ **Assignable Types** (§6.5.16, §6.5.2.2)

Namensräume in C

- ▶ Labels;
- ▶ *tags* für Strukturen, Aufzählungen, Unionen;
- ▶ Felder von **Strukturen** (je eines pro Struktur/Union);
- ▶ Alles andere: **Funktionen**, **Variablen**, **Typen**, ...
- ▶ Legal: `struct foo {int foo; } foo;`
 - ▶ Was ist `sizeof(foo);`?

Deklarationen in C

- ▶ Sprachphilosophie: Declaration resembles use
- ▶ Deklarationen:
 - ▶ *declarator* — was deklariert wird
 - ▶ *declaration* — wie es deklariert wird

Der *declarator*

Anzahl	Name	Syntax
Kein oder mehr	<i>pointer</i>	<i>*</i> <i>type-qualifier *</i>
Ein	<i>direct-declarator</i>	<i>identifier</i> <i>identifier [expression]</i> <i>identifier (parameter-type-list)</i>
Höchstens ein	<i>initializer</i>	<i>= expression</i>

Die declaration

Anzahl	Name	Syntax
Ein oder mehr	<i>type-specifier</i>	<code>void, char, short, int, long, double, float, signed, unsigned, struct-or-union-spec, enum-spec, typedef-name</code>
Beliebig	<i>storage-class</i>	<code>extern, static, register, auto, typedef</code>
Beliebig	<i>type-qualifier</i>	<code>const, volatile</code>
Genau ein	<i>declarator</i>	s.o.
Beliebig	<i>declarator-list</i>	<code>, declarator</code>
Genau ein		<code>;</code>

Restriktionen

Illegal:

- ▶ Funktion gibt Funktion zurück: `foo () ()`
- ▶ Funktion gibt Feld zurück: `foo () []`
- ▶ Felder von Funktionen: `foo [] ()`

Aber **erlaubt**:

- ▶ Funktion gibt **Zeiger** auf Funktion zurück: `int (* fun) () ;`
- ▶ Funktion gibt **Zeiger** auf Feld zurück: `int (*foo ()) [] ;`
- ▶ Felder von **Zeigern** auf Funktionen: `int (*foo []) () ;`

Strukturen

- ▶ Syntax: `struct identifierOpt { struct-declaration* }`
- ▶ Einzelne Felder (*struct-declaration*):
 - ▶ Beliebige viele *type-specifier* oder *type-qualifier*, dann
 - ▶ *declarator*, oder
 - ▶ *declarator*_{Opt} : *expression*
- ▶ Strukturen sind **first-class objects**
 - ▶ Können zugewiesen und übergeben werden.
- ▶ **union**: syntaktisch wie `struct` (andere Semantik!)

Präzedenzen für Deklarationen

- A Name zuerst (von links gelesen)
- B In abnehmender Rangfolge:
 - B.1 Klammern
 - B.2 Postfix-Operatoren:
 - () für Funktion
 - [] für Felder
 - B.3 Präfix-Operator:
 - * für Zeiger-auf
- C *type-qualifier* beziehen sich auf *type-specifier* rechts daneben (wenn vorhanden), ansonsten auf den Zeiger * links davor

Beispiele

▶ `char* const *(*next)();`

Beispiele

- ▶ `char* const *(*next)();`
- ▶ `char *(*c [10])(int ** p);`

Beispiele

- ▶ `char* const *(*next)();`
- ▶ `char *(*c [10])(int ** p);`
- ▶ `void (*signal (int sig , void (*func)(int)))(int);`

Beispiele

- ▶ `char* const *(*next)();`
- ▶ `char *(*c [10])(int ** p);`
- ▶ `void (*signal (int sig , void (*func)(int)))(int);` Lesbarer als
`typedef void (*sig_handler_t)(int);`
`sig_handler_t signal(int signum , sig_handler_t handler`

Typdefinitionen mit typedef

- ▶ typedef definiert Typsynonyme
- ▶ typedef ändert eine Deklaration von
 - ▶ x ist eine Variable vom Typ T zu
 - ▶ x ist ein Typsynonym für T
- ▶ Braucht nicht am **Anfang** zu stehen (aber empfohlen)
- ▶ Nützliche Tipps:
 - ▶ Kein typedef für structs
 - ▶ typedef für lesbarere Typen
 - ▶ typedef für Portabilität (e.g. uint32_t in <stdint.h>)

Zusammenfassung

- ▶ **Typen** in C: Object, incomplete, function; qualified; compatible
- ▶ **Deklarationen** in C:
 - ▶ *declarator, declaration*
 - ▶ Präzedenzregeln: Postfix vor Präfix, rechts nach links

Nächste Woche

- ▶ Programmauswertung (“dynamische” Semantik)
- ▶ Wie wird folgendes Programm **nach dem Standard** ausgewertet:

```
int x;  
int a[10];  
int *y;
```

```
x= 0;  
x= x+1;  
y= &x;  
*y= 5;  
x= a[3];  
y= &a[3];  
*y= 5;
```