SAARLAND UNIVERSITY
COMPUTER SCIENCE

C/C++ Programmierung

Verbunde, Reihungen, Zeiger

Sebastian Hack
Christoph Mallon

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

Fachbereich Informatik
Universität des Saarlandes

Wintersemester 2009/2010

Typen

§6.2.5

- Primitive Typen
  - Ganzzahlen (`int`, ...)
  - Gleitkommazahlen (`float`, ...)
  - Boole (`_Bool`)
  - Aufzählungen (`enum`, später)
- Abgeleitete Typen
  - Funktionen (`void f(int)`)
  - Verbunde (`struct`)
  - Varianten (`union`)
  - Reihungen (arrays, `int a[10]`)
  - Zeiger (`int* p`)

Verbunde

§6.7.2.1, struct

```
struct XY;    // Deklaration der struct-Namen

struct XY {   // Definition
  int x;
  int y;
} a, b;       // Direkt erzeugen von Objekten moeglich

struct XY c; // Weiteres Objekt
```

- Ansammlung von Daten
  → Name in Pascal: Datensatz (record)
- Zugriff auf Felder mittels .-Operator

```
a.x = b.x = 0;
```

- Verbundnamen sind in eigenem Namensraum
  → Zusammen mit `union` und `enum` (später)
- Verbundnamen nur mit Präfix `struct` verwendbar
- Zwischen Einträgen kann sich Füllung (padding) befinden
  → Auch nach letztem Eintrag

Verbunde: Initialisierung

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
struct XY a = { 0, 1 };              // a.x = 0, a.y = 1
struct XY b = { 1 };                 // b.x = 1, b.y = 0
struct XY c = { .y = 1, .x = 0 }; // C99
struct XY d = { .x = 1 };            // d.y = 0
```

- Werte werden in Reihenfolge der Felddefinitionen verwendet
- Nicht explizit initialisierte Felder sind 0
- C99: Initialisierung mittels Designatoren

Varianten

§6.7.2.1, union

- Syntax wie bei `struct`
- Alle Einträge werden übereinandergelegt
- Gesamtgröße ist Größe der größten Variante
- Achtung: Programmierer muss selbst darauf achten, welcher Eintrag gültig ist
  $\longrightarrow$ Lesen von nicht zuletzt beschriebener Variante undefiniert
- Mit herkömmlicher Syntax nur Initialisierung von erster Variante möglich
  $\longrightarrow$ C99: Mit Designatorsyntax beliebiger Eintrag initialisierbar

Beispiel: Varianten

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
union U {
  unsigned i;
  float    f;
};

union U a = { 23 };          // i ist initialisiert
union U b = { .i = 23 };     // C99, s.o.
union U c = { .f = 42.f }; // C99

unsigned x = c.i; // Undefiniert!
```

- Wird häufig missbraucht, um an die Bitdarstellung von Gleitkommazahlen zu gelangen

Varianten: Gemeinsamer Anfang

§6.5.2.3:5

```
struct SDL_KeyboardEvent {
  Uint8       type;
  Uint8       which;
  Uint8       state;
  SDL_keysym  keysym;
};
struct SDL_ResizeEvent {
  Uint8 type;
  int    w;
  int    h;
};
union SDL_Event {
  struct SDL_KeyboardEvent key;
  struct SDL_ResizeEvent   resize;
  /* Weitere Varianten ... */
};
```

- Auf gemeinsamen Anfang darf mittels anderer Variante zugegriffen werden
- Gemeinsamer Anfang: Selbe Liste von Typen

Typalias

§6.7.7, typedef


```
int          x; // Variable namens x vom Typ int
typedef int  x; // Typalias namens x vom Typ int
```

- Erzeugt Alias für anderen Typen, kein neuer Typ
- Syntax sieht aus wie Deklaration mit Schlüsselwort `typedef`
- Häufiger Einsatz: Verbundnamen (`struct` tags) in einfache Typnamen verwandeln

```
typedef struct XY XY;
struct XY a; // Bisher: Mit struct-Namen
XY        b; // Jetzt:  Mit Typalias
// Oder in einem Rutsch
typedef struct XY { int x; int y; } XY;
// Anonymer Verbund nur mit Alias
typedef struct { int x; int y } XY;
```

Reihungen

§6.7.5.2, arrays

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
int arr[10];
```

- Reihungen in C sind Objekte, keine Referenzen
  → z.B. Java: `int[] r;` ist Refernz auf Reihung
- Größe steht zur Übersetzungszeit fest
- Zugriff auf Elemente mittels `[]`-Operator
  → Symmetrie von Deklaration und Benutzung
- Indizes beginnen bei 0
- Achtung: Keine Prüfung der Reihungsgrenzen bei Zugriff
  → Bei Verletzung Verhalten undefiniert

```
arr[0]  = 42;      // Setze erstes Element
x       = arr[9];  // Lies letzes Element
arr[10] = 23;      // Grenzen verletzt, undefiniert!
```

Reihung: Initialisierung

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
int a[10] = { 1, 2, 3 }; // Rest automatisch 0
int b[]   = { 1, 2, 3 }; // Automatische Laenge 3
// Designatoren, C99
int c[10] = { [7] = 42, [3] = 23 }; // Rest 0
int d[]   = { [7] = 42 };           // Laenge 8, Rest 0
int e[]   = { 1, [5] = 2, 3 }; // Gemischt: 0, 5, 6
```

- Ohne explizite Längenangabe wird Länge automatisch berechnet
- Nicht explizit initialisierte Elemente sind 0
- C99: Initialisierung mittels Designatoren

Größe von Reihungen

- `sizeof` liefert Größe einer Reihung in Byte
- Daraus Anzahl der Elemente berechenbar:

```
sizeof(arr) / sizeof(arr[0])
```

  → Größe der Reihung geteilt durch Größe eines Elements

- Funktioniert nur mit Reihungen bekannter Größe
  → Nicht bei dynamischen Reihungen, keine Zeiger

```
// sizeof(int) sei 4
int arr[10];
sizeof(arr)                    // 40
sizeof(arr) / sizeof(arr[0]) // 10
sizeof(arr) / sizeof(int)     // fehleranfaellig
sizeof(arr) / 4                // plattformabhaengig
10                             // igitt!

// Praktisches Makro zur Bestimmung der Reihungslaenge
#define lengthof(x) (sizeof(x) / sizeof((x)[0]))
```

Mehrdimensionale Reihungen

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
int arr[2][4];
```

- Reihung mit zwei Zeilen und vier Spalten
- Andere Sichtweise:
  - Reihung von zwei Objekten
  - Elemente der Reihung sind Reihung von vier Objekten vom Typ `int`

```
typedef int int4[4];
int4 arr[2];
```

- `arr[1]` greift auf zweite Zeile zu, Ergebnistyp ist `int[4]`
- `arr[1][3]` greift auf vierte Spalte in der zweiten Zeile zu, Typ `int`

Zeiger

§6.7.5.1

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- Adressoperator: `&x` bestimmt Adresse von Objekt `x`
- Dereferenzierungsoperator: `*p` greift auf durch den Zeiger `p` referenziertes Objekt zu
- Rechnen mit Zeigern möglich: `+`, `-`, `++`, `--`
  $\longrightarrow$ Elementsweise, nicht byteweise
- Spezielle Bedeutung: `void*` ist Zeiger auf beliebiges
  $\longrightarrow$ Keine Arithmetik mit `void*`!

Zeigerarithmetik

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
int a[10];
int* p = &a[0];
int* q;
p += 3;           // Zeigt auf viertes Objekt
++p;              // Zeigt auf fuenftes Objekt
q = p - 2;        // Zeigt auf drittes Objekt
p - q;            // Abstand: 2 Objekte
p = &a[0] + 10; // Legal, nicht dereferenzieren!
p - q;            // Abstand: 8 Objekte
```

- Typ von Differenz zweier Zeiger ist `ptrdiff_t`, ein vorzeichenbehafteter Ganzzahltyp
- Berechnung der Adresse genau ein Objekt hinter das Ende einer Reihung ist erlaubt
  $\longrightarrow$ Adresse darf nicht dereferenziert werden!
- Keine Arithmetik mit `void*` und Funktionszeigern

```
// Setze alle Eintraege der Reihung auf 0
int arr[10];
int* end = &arr[0] + 10;
for (int* i = arr; i != end; ++i) *i = 0;
```

void*

- Zeiger auf beliebiges
- Jeder (nicht-funktionszeiger) implizit in `void*` umwandelbar
- `void*` implizit in jeden (nicht-funktionszeiger) umwandelbar
  $\longrightarrow$ nicht in C++, explizite Typumwandlung (cast) notwendig

[] und Zeiger

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
p[i]      = 23;
*(p + i)  = 23;
```

- `p[i]` ist äquivalent zu `*(p + i)`
- Kuriosum: Kommutativität

  `p[5]` $\iff$ `*(p + 5)` $\iff$ `*(5 + p)` $\iff$ `5[p]`

Zeiger und Verbunde/Varianten

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
struct XY* p;
// Typ von  p: struct XY*
// Typ von *p: struct XY
(*p).x = 23;
```

Syntaktischer Zucker: `a->b` äquivalent zu `(*a).b`

```
p->x = 23;
```

Funktionszeiger


```
int f(int);              // Funktionsdeklaration
int (*pf)(int) = &f; // Zeiger auf Funktion

(*pf)(23);               // Aufruf mittels Funktionszeiger
```

- Klammern sind bedeutungstragend

```
int *g(int); /* Funktion mit Ergebnistyp Zeiger
              * auf int */
int* g(int); // Formatierung egal
```

- `&` bei Funktionen redundant
  $\longrightarrow$ Funktionsdesignatoren degenerieren automatisch zu Funktionszeigern
- `(*)` bei Verwendung ebenso redundant

```
int (*pf)(int) = f; // & redundant
pf(23);             // (*) redundant
```

Zeiger und Funktionen


```
void f(int*);
/* ... */
int i;
f(&i);
```

- Parameterübergabe immer per Wertübergabe (call-by-value)
  $\rightarrow$ C++ hat Referenzübergabe (call-by-reference) (später)
- Referenzübergabe (call-by-reference) mit Zeigern simulierbar:
- Zeiger selbst wird per Wertübergabe übergeben

Reihungen und Zeiger

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- Reihungen und Zeiger sind verschieden
- Reihung: Objekt, das aus einer fortlaufenden Reihe von einzelnen Objekten besteht
- Zeiger: Objekt, das auf ein anderes Objekt zeigt
- Reihungen degenerieren bei Verwendung (`[]`, `*`, `+`, ...) automatisch zu Zeiger auf erstes Element
  $\longrightarrow$ Ausnahmen: `&` und `sizeof`

```
int arr[10];
int* p = &arr[0]; // Zeiger auf erstes Element
int* p = arr;     // Syntaktischer Zucker, s.o.
```

Reihungen und Funktionen

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```c
void f(int*, size_t n);
/* ... */
int a[10];
f(a, sizeof(a) / sizeof(*a));
```

- Reihungen weder als Parameter noch als Ergebnistyp
  $\longrightarrow$ Übergabe nur per Zeiger
- Achtung: Keine Information über die Länge
  $\longrightarrow$ Muss selbst verwaltet werden

Reihungen und Funktionen

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- Syntaktischer Zucker: Wenn Parametertyp ein Reihung ist, dann wird es wie ein Zeiger behandelt
- Gilt nur, wenn es ein Reihungstyp ist
- Weitere Dimensionen bleiben erhalten

```
void f(int a[]);    // Syntaktischer Zucker
void f(int* a);
void f(int a[10]); // Zahl ist bedeutungslos

void g(int a[][10]);  // Nur erste Reihung!
void g(int (*a)[10]); // Aequivalent
```

Beispiel: Reihungen und Zeiger

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
int  arr[10]; // Typ: int[10]
int* p;       // Typ: int*
p = arr;      // int[10] degeneriert zu int*
p = &arr;     // Fehler!
int (*r)[10] = &arr;
++p; // Zeigt auf zweites Element der Reihung
++r; // Zeigt hinter die Reihung
```

- `arr` degeneriert bei Verwendung automatisch zu Zeiger auf erstes Element
  $\longrightarrow$ Typ: Zeiger auf Element `int*`
- `&arr` liefert Zeiger auf ganze Reihung
  $\longrightarrow$ Typ: Zeiger auf Reihung `int (*)[10]`

Deklarationen

§6.7

```
declaration:
  declaration-specifier+ declarator-list

declaration-specifier:
  type-specifier
  type-qualifier              // spaeter
  storage-class-specifier     // spaeter

type-specifier: one of
  void char short int long float unsigned /* ... */

declarator-list:
  declarator
  declarator-list, declarator
```

- Nur Typname und dessen Qualifizierungen (wie `const`, später) gilt für alle Deklarationen
  → type specifier (§6.7.2), type qualifier (§6.7.3)
- Alles andere (`*`, `[]`, `()`) gilt nur für den jeweiligen Deklator
  → declarator (§6.7.5)

Beispiel: Mehrere Deklaratoren

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```c
int*a,b,c[10],f(void);
// Gleiche Bedeutung wie:
int* a;
int  b;
int  c[10];
int  d(void);
```

- `int` ist gemeinsame Deklarationsspezifizierungen (declaration specifier)
- Zeiger, Reihungen, Funktionen gilt nur jeweils für einen Deklarator (declarator)
  $\longrightarrow$ Haupsächlich verwirrend bei Zeigern
  $\longrightarrow$ Gemischte Deklarationen vermeiden

Beispiele: Deklarationen


```
int i;              // ein int
int(i);             // s.o. mit redundanten Klammern
int*p;              // Zeiger auf int
int(*p);            // s.o. mit redundanten Klammern
int(*pf)(void);     // Zeiger auf Funktion
int*f(void);        // Funktion, die int* zurueckgibt
int*(f)(void);      // s.o. mit redundanten Klammern
int(*pa)[10];       // Zeiger auf Reihung mit 10 Elementen
int*a[10];          // Reihung von 10 Zeigern auf int
int*(a)[10];        // s.o. mit redundaten Klammern
```

- Deklarationsspezifizierungen (declaration specifiers) hier überall nur `int`
- Klammern, um Bindung/Reihenfolge von Zeigern/Reihungen/Funktionen zu ändern

Beispiel: Komplizierter Typ[1]

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
char (*(*x())[])()
```

- Vorgehen: Von innen nach außen
- `()` und `[]` binden stärker als `*` (und `&`, C++, später)
  $\longrightarrow$ Wie bei den Operatorpräzedenzen

```
char (*(*x())[])()  // ()  Funktion mit Ergebnistyp ...
char (*(*.)[])()    // (*) Zeiger auf ...
char (*.[])()       // []  Reihung von ...
char (*.)()         // (*) Zeiger auf ...
char .()            // ()  Funktion mit Ergebnistyp ...
char                //     char
```

[1] Aus der „Ode to C“

Beispiel: Komplizierter Typ[1]

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
char (*(*x())[])()
```

- Vorgehen: Von innen nach außen
- `()` und `[]` binden stärker als `*` (und `&`, C++, später)
  $\longrightarrow$ Wie bei den Operatorpräzedenzen

```
char (*(*x())[])() // ()  Funktion mit Ergebnistyp ...
char (*(*.)[])()   // (*) Zeiger auf ...
char (*.[])()      // []  Reihung von ...
char (*.)()        // (*) Zeiger auf ...
char .()           // ()  Funktion mit Ergebnistyp ...
char               //     char
```

Fazit: `typedef` verwenden

[1] Aus der „Ode to C“

Beispiel: Komplizierter Typ mit Typaliasen

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
typedef           char                    fct_char();
typedef           fct_char*           ptr_fct_char;
typedef      ptr_fct_char        arr_ptr_fct_char[];
typedef arr_ptr_fct_char* ptr_arr_ptr_fct_char;
ptr_arr_ptr_fct_char x();
```