Dekarationen

Syntax für primitive Datentypen:

Typ Bezeichner [`=` *Ausdruck*] [`,` *Bezeichner* [`=` *Ausdruck*] `,` …] `;`

Beispiel:

```
1 int foo = 4, bar = 6, baz;
```

Spezielle Deklaratoren existieren für Pointer, Arrays, und Funktionen:

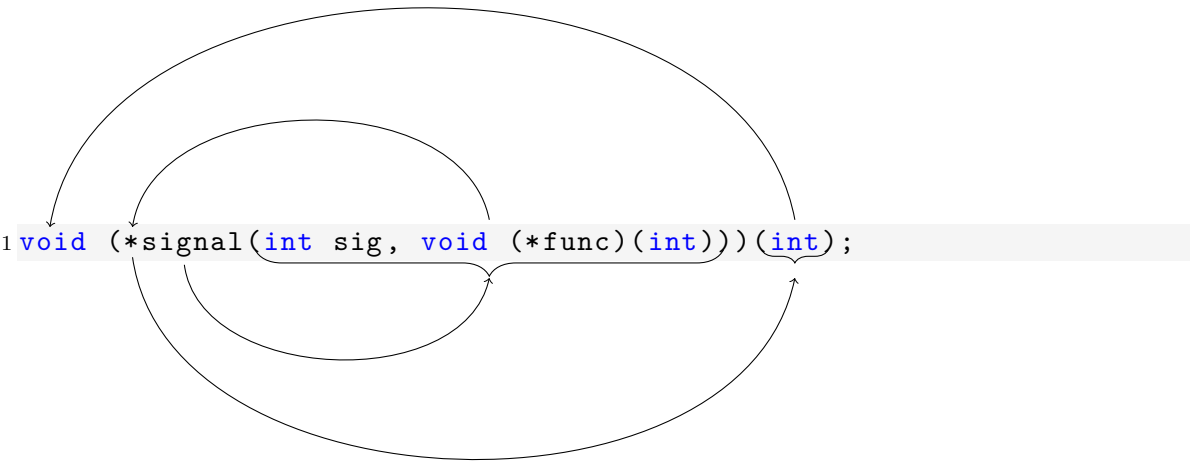
Typ Deklarator [`=` *Ausdruck*] [`,` *Deklarator* [`=` *Ausdruck*] `,` …] `;`

Beispiel:

```
1 char *foo = "Hallo"; /* Deklarator: *foo */
2 char bar[] = "Welt"; /* Deklarator: bar[] */
3 int *lipsum, *baz(void); /* Deklaratoren *lipsum, *baz(void) */
```

Intuititiv sieht ein Deklarator aus wie ein Ausdruck, den man auf die Variable anwenden kann, um den zugrundeliegenden Typen zu erhalten. Bspw. deklariert `int *foo;` die Variable `foo` als Pointer auf `int`, denn `*foo` ist ein `int`.

Deklaratoren müssen von Innen nach außen gelesen werden: `int *bar()` deklariert `bar` als Funktion, die einen Zeiger auf einen `int` zurückgibt, denn `*bar()` ist ein `int`. Hierbei folgen die Rangfolge der Deklaratoren der Operatorenrangfolge.



`signal()` ist also eine Funktion, die als Argumente …

1. einen `int sig` und
2. einen Pointer `func` auf eine Funktion, die
   a) ein `int` nimmt und
   b) `void` zurückibt

… bekommt. Außerdem gibt sie einen Pointer auf eine Funktion zurück, die

1. einen `int` nimmt und
2. `void` zurückgibt.

Datentypen

Primitive Datentypen Alle ganzzahligen Datentypen sind einfach so (außer `char`) oder mit `signed` vorzeichenbehaftet. Sie können mit `unsigned` vorzeichenlos gemacht werden

- `char` Mindestens ein Byte, damit kleinster Typ. Häufig für Buchstaben verwendet, speziell in ASCII-Codierung.
- `_Bool` Boolean. So groß wie ein `char`, aber nur 0 oder 1.
- `int` Ganzzahl. Mindestens 16 Bit, üblicherweise 32 Bit.
- `float` Gleitkommazahl einfacher Präzision. Eigentlich immer 32 Bit.
- `double` Gleitkommazahl doppelter Präzision Eigentlich immer 64 Bit.

Es gibt noch Qualifikatoren, mit denen man `int` und z. T. auch `double`-Typen in der größe verändern kann:

- `short int` kurze Ganzzahl. Mindestens und eigentlich immer 16 Bit.
- `long int` lange Ganzzahl. Mindestens 32 Bit, üblicherweise 32 oder 64 Bit.
- `long long int` sehr lange Ganzzahl. Mindestens und eigentlich immer 64 Bit.
- `long double` Gleitkommazahl hoher Präzision. Mindestens 64 Bit, auf x86 80 Bit.

Wichtige abgeleitete Datentypen

Aus `<stddef.h>`:

- `size_t` Vorzeichenloser Ganzzahltyp, der groß genug für die Größe jedes Objektes ist. Idealer Typ für Schleifenzähler o. ä.
- `ptrdiff_t` Vorzeichenbehafteter Ganzzahltyp, der groß genug ist, die Differenz zwischen zwei Zeigern zu halten. Gelegentlich hilfreich für Zeigertricks.

Aus `<stdint.h>`:

- `int`N`_t` Vorzeichenbehafteter Ganzzahltyp mit genau N Bits, $N \in \{8, 16, 32, 64\}$.
- `uint`N`_t` Vorzeichenloser Ganzzahltyp mit genau N Bits, $N \in \{8, 16, 32, 64\}$.
- `intmax_t` Vorzeichenbehafteter Ganzzahltyp maximaler Größe.
- `uintmax_t` Vorzeichenloser Ganzzahltyp maximaler Größe.

Funktionsdefinitionen Funktionen können nur auf höchster Ebene definiert werden. Beim Funktionsaufruf werden alle Argumente in die Parameter der Funktion kopiert, hierbei werden Arrays werden als Pointer übergeben. Soll in ein Argument geschrieben werden, so muss ein Zeiger auf dieses übergeben werden werden. Alle Variablen einer Funktion werden beim Funktionsende zerstört! Syntax:

```
Rückgabetyp Funktionsname ( [ Argument , … ] )
{
  …
}
```

Soll die Funktion keine *Argumente nehmen, schreibt man den speziellen Parameter* `void`

Aufzählungen Aufzählungen (Enumerationen) definieren Mengen an Konstanten. Die Konstanten haben den Typ `int`.

`enum` [*Bezeichner*] `{` *Bezeichner* [`=` *Ausdruck*] [`,` …] `;`

Wird optional ein Bezeichner nach dem Schlüsselwort `enum` angegeben, dann wird ein Enumerationstyp mit dem angegebenen Namen erzeugt. Das Wort `enum` ist Teil des Namens. Wird für eine Konstante in der Enumeration kein Wert angegeben, dann wird der Standardwert genommen. Für die erste Konstante ist dieser Null, für folgende Konstanten der Wert der vorherigen Konstante um 1 erhöht.

Zeiger Zeiger sind Typen, die auf Objekte zeigen. Ein Zeiger wird mit einem *Zeigerdeklarator* definiert:

Typ `*` *Deklarator* [`=` *Ausdruck*] `;`

Das Token „`*`“ weißt darauf hin, dass die zu deklarierende Variable ein Zeiger ist. Zeiger können mit dem Präfixoperator `*` *dereferenziert* werden. Man erhält das Objekt, auf das der Zeiger zeigt. Auf dieses kann auch zugewiesen werden (es ist ein lvalue). Mit dem Präfixoperator `&` erhält man die Adresse eines Objektes, also einen Zeiger auf das Objekt.

Der spezielle Zeiger `NULL` aus `<stddef.h>` zeigt nirgendwo hin. Man kann Zeiger auf `NULL` setzen, um sie explizit nicht auf ein Objekt zeigen zu lassen. Das ist für viele Datenstrukturen hilfreich.

Arithmetik auf Zeigern ist möglich: Man kann Ganzzahlen addieren und subtrahieren. Dies fasst den Zeiger als Zeiger auf ein Array auf und verschiebt diesen um die gegebene Zahl von Elementen. Man kann die Differenz zweier Zeiger bestimmen, sofern sie beide auf Elemente des gleichen Arrays zeigen. Das Ergebnis, vom Typ `ptrdiff_t`, ist der Unterschied der Indizes der beiden Elemente.

Arrays Arrays sind Sammlungen typgleicher Elemente. Man deklariert sie mit dem Array-Deklarator:

Typ Deklarator `[` [*Ausdruck*] `]` [`=` `{` *Ausdruck* [`,` …] `}`] `;`

In den eckigen Klammern steht die Länge des Arrays. Lässt man diese weg, dann wird ein Array unbekannter Länge deklariert. In einer Definition hat das Array dann so viele Elemente, wie initialisiert worden sind.

Man kann mit dem Index-Operator `[]` auf Array-Elemente zugreifen. Arrays verhalten sich ein bischen wie Zeiger auf das erste Element des Arrays. Strings sind Arrays von `char` Elementen, deren letztes Element eine Null ist.