If the types don't match, C converts the initializer using the same rules as for assignment:

conversion during assignment ➤7.4

```
int j = 5.5;      /* converted to 5 */
```

The initializer for a pointer variable must be a pointer expression of the same type as the variable or of type `void *`:

```
int *p = &i;
```

The initializer for an array, structure, or union is usually a series of values enclosed in braces:

```
int a[5] = {1, 2, 3, 4, 5};
```

C99

designated initializers ➤8.1, 16.1

In C99, brace-enclosed initializers can have other forms, thanks to designated initializers.

To complete our coverage of declarations, let's take a look at some additional rules that govern initializers:

- An initializer for a variable with static storage duration must be constant:

```
#define FIRST 1
#define LAST 100

static int i = LAST - FIRST + 1;
```

  Since `LAST` and `FIRST` are macros, the compiler can compute the initial value of `i` (100 – 1 + 1 = 100). If `LAST` and `FIRST` had been variables, the initializer would be illegal.

- If a variable has automatic storage duration, its initializer need not be constant:

```
int f(int n)
{
  int last = n - 1;
  ...
}
```

- A brace-enclosed initializer for an array, structure, or union must contain only constant expressions, never variables or function calls:

```
#define N 2

int powers[5] = {1, N, N * N, N * N * N, N * N * N * N};
```

  C99

  Since `N` is a constant, the initializer for `powers` is legal; if `N` were a variable, the program wouldn't compile. In C99, this restriction applies only if the variable has static storage duration.

- The initializer for an automatic structure or union can be another structure or union: