Chapter 17: Initialization

Section 17.1: Initialization of Variables in C

In the absence of explicit initialization, external and `static` variables are guaranteed to be initialized to zero; automatic variables (including `register` variables) have *indeterminate*1 (i.e., garbage) initial values.

Scalar variables may be initialized when they are defined by following the name with an equals sign and an expression:

```
int x = 1;
char squota = '\'';
long day = 1000L * 60L * 60L * 24L; /* milliseconds/day */
```

For external and `static` variables, the initializer must be a *constant expression*2; the initialization is done once, conceptually before the program begins execution.

For automatic and `register` variables, the initializer is not restricted to being a constant: it may be any expression involving previously defined values, even function calls.

For example, see the code snippet below

```
int binsearch(int x, int v[], int n)
{
   int low = 0;
   int high = n - 1;
   int mid;
   ...
}
```

instead of

```
   int low, high, mid;

   low = 0;
   high = n - 1;
```

In effect, initialization of automatic variables are just shorthand for assignment statements. Which form to prefer is largely a matter of taste. We generally use explicit assignments, because initializers in declarations are harder to see and further away from the point of use. On the other hand, variables should only be declared when they're about to be used whenever possible.

Initializing an array:

An array may be initialized by following its declaration with a list of initializers enclosed in braces and separated by commas.

For example, to initialize an array days with the number of days in each month:

```
int days_of_month[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
```

(Note that January is encoded as month zero in this structure.)

When the size of the array is omitted, the compiler will compute the length by counting the initializers, of which there are 12 in this case.

If there are fewer initializers for an array than the specified size, the others will be zero for all types of variables.

It is an error to have too many initializers. There is no standard way to specify repetition of an initializer — but GCC has an extension to do so.

Version < C99

In C89/C90 or earlier versions of C, there was no way to initialize an element in the middle of an array without supplying all the preceding values as well.

Version ≥ C99

With C99 and above, designated initializers allow you to initialize arbitrary elements of an array, leaving any uninitialized values as zeros.

Initializing Character arrays:

Character arrays are a special case of initialization; a string may be used instead of the braces and commas notation:

```
char chr_array[] = "hello";
```

is a shorthand for the longer but equivalent:

```
char chr_array[] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

In this case, the array size is six (five characters plus the terminating `'\0'`).

1 What happens to a declared, uninitialized variable in C? Does it have a value?

2 Note that a *constant expression* is defined as something that can be evaluated at compile-time. So, `int global_var = f();` is invalid. Another common misconception is thinking of a `const` qualified variable as a *constant expression*. In C, `const` means "read-only", not "compile time constant". So, global definitions like `const int SIZE = 10; int global_arr[SIZE];` and `const int SIZE = 10; int global_var = SIZE;` are not legal in C.

Section 17.2: Using designated initializers

Version ≥ C99

C99 introduced the concept of *designated initializers.* These allow you to specify which elements of an array, structure or union are to be initialized by the values following.

Designated initializers for array elements

For a simple type like plain `int`:

```
int array[] = { [4] = 29, [5] = 31, [17] = 101, [18] = 103, [19] = 107, [20] = 109 };
```

The term in square brackets, which can be any constant integer expression, specifies which element of the array is to be initialized by the value of the term after the `=` sign. Unspecified elements are default initialized, which means zeros are defined. The example shows the designated initializers in order; they do not have to be in order. The example shows gaps; those are legitimate. The example doesn't show two different initializations for the same element; that too is allowed (ISO/IEC 9899:2011, §6.7.9 Initialization, ¶19 *The initialization shall occur in initializer list order, each initializer provided for a particular subobject overriding any previously listed initializer for the same subobject*).

In this example, the size of the array is not defined explicitly, so the maximum index specified in the designated initializers dictates the size of the array — which would be 21 elements in the example. If the size was defined, initializing an entry beyond the end of the array would be an error, as usual.

Designated initializers for structures

You can specify which elements of a structure are initialized by using the `.element` notation:

```
struct Date
{
    int year;
    int month;
    int day;
};

struct Date us_independence_day = { .day = 4, .month = 7, .year = 1776 };
```

If elements are not listed, they are default initialized (zeroed).

Designated initializer for unions

You can specify which element of a union is initialize with a designated initializer.

`Version = C89`

Prior to the C standard, there was no way to initialize a `union`. The C89/C90 standard allows you to initialize the first member of a `union` — so the choice of which member is listed first matters.

```
struct discriminated_union
{
    enum { DU_INT, DU_DOUBLE } discriminant;
    union
    {
        int     du_int;
        double  du_double;
    } du;
};

struct discriminated_union du1 = { .discriminant = DU_INT, .du = { .du_int = 1 } };
struct discriminated_union du2 = { .discriminant = DU_DOUBLE, .du = { .du_double = 3.14159 } };
```

`Version ≥ C11`

Note that C11 allows you to use anonymous union members inside a structure, so that you don't need the `du` name in the previous example:

```
struct discriminated_union
{
    enum { DU_INT, DU_DOUBLE } discriminant;
    union
    {
        int     du_int;
        double  du_double;
    };
};

struct discriminated_union du1 = { .discriminant = DU_INT, .du_int = 1 };
struct discriminated_union du2 = { .discriminant = DU_DOUBLE, .du_double = 3.14159 };
```

Designated initializers for arrays of structures, etc

These constructs can be combined for arrays of structures containing elements that are arrays, etc. Using full sets of braces ensures that the notation is unambiguous.

```
typedef struct Date Date;  // See earlier in this example

struct date_range
{
    Date    dr_from;
    Date    dr_to;
    char    dr_what[80];
};

struct date_range ranges[] =
{
    [3] = { .dr_from = { .year = 1066, .month = 10, .day = 14 },
            .dr_to   = { .year = 1066, .month = 12, .day = 25 },
            .dr_what = "Battle of Hastings to Coronation of William the Conqueror",
          },
    [2] = { .dr_from = { .month = 7, .day =  4, .year = 1776 },
            .dr_to   = { .month = 5, .day = 14, .year = 1787 },
            .dr_what = "US Declaration of Independence to Constitutional Convention",
          }
 };
```

Specifying ranges in array initializers

GCC provides an extension that allows you to specify a range of elements in an array that should be given the same initializer:

```
int array[] = { [3 ... 7] = 29, 19 = 107 };
```

The triple dots need to be separate from the numbers lest one of the dots be interpreted as part of a floating point number (*maximimal munch* rule).

Section 17.3: Initializing structures and arrays of structures

Structures and arrays of structures can be initialized by a series of values enclosed in braces, one value per member of the structure.

```
struct Date
{
    int year;
    int month;
    int day;
};

struct Date us_independence_day = { 1776, 7, 4 };

struct Date uk_battles[] =
{
    { 1066, 10, 14 },    // Battle of Hastings
    { 1815,  6, 18 },    // Battle of Waterloo
    { 1805, 10, 21 },    // Battle of Trafalgar
};
```

Note that the array initialization could be written without the interior braces, and in times past (before 1990, say) often would have been written without them:

```
struct Date uk_battles[] =
{
    1066, 10, 14,    // Battle of Hastings
    1815,  6, 18,    // Battle of Waterloo
    1805, 10, 21,    // Battle of Trafalgar
};
```

Although this works, it is not good modern style — you should not attempt to use this notation in new code and should fix the compiler warnings it usually yields.

See also designated initializers.