Chapter 44: Declarations

Section 44.1: Calling a function from another C file

foo.h

```
#ifndef FOO_DOT_H    /* This is an "include guard" */
#define FOO_DOT_H    /* prevents the file from being included twice. */
                     /* Including a header file twice causes all kinds */
                     /* of interesting problems.*/

/**
 * This is a function declaration.
 * It tells the compiler that the function exists somewhere.
 */
void foo(int id, char *name);

#endif /* FOO_DOT_H */
```

foo.c

```
#include "foo.h"    /* Always include the header file that declares something
                     * in the C file that defines it. This makes sure that the
                     * declaration and definition are always in-sync.  Put this
                     * header first in foo.c to ensure the header is self-contained.
                     */
#include <stdio.h>

/**
 * This is the function definition.
 * It is the actual body of the function which was declared elsewhere.
 */
void foo(int id, char *name)
{
    fprintf(stderr, "foo(%d, \"%s\");\n", id, name);
    /* This will print how foo was called to stderr - standard error.
     * e.g., foo(42, "Hi!") will print `foo(42, "Hi!")`
     */
}
```

main.c

```
#include "foo.h"

int main(void)
{
    foo(42, "bar");
    return 0;
}
```

Compile and Link

First, we *compile* both `foo.c` and `main.c` to *object files*. Here we use the `gcc` compiler, your compiler may have a different name and need other options.

```
$ gcc -Wall -c foo.c
$ gcc -Wall -c main.c
```

Now we link them together to produce our final executable:

```
$ gcc -o testprogram foo.o main.o
```

Section 44.2: Using a Global Variable

Use of global variables is generally discouraged. It makes your program more difficult to understand, and harder to debug. But sometimes using a global variable is acceptable.

global.h

```
#ifndef GLOBAL_DOT_H    /* This is an "include guard" */
#define GLOBAL_DOT_H

/**
 * This tells the compiler that g_myglobal exists somewhere.
 * Without "extern", this would create a new variable named
 * g_myglobal in _every file_ that included it. Don't miss this!
 */
extern int g_myglobal; /* _Declare_ g_myglobal, that is promise it will be _defined_ by
                        * some module. */

#endif /* GLOBAL_DOT_H */
```

global.c

```
#include "global.h" /* Always include the header file that declares something
                     * in the C file that defines it. This makes sure that the
                     * declaration and definition are always in-sync.
                     */

int g_myglobal;     /* _Define_ my_global. As living in global scope it gets initialised to 0
                     * on program start-up. */
```

main.c

```
#include "global.h"

int main(void)
{
    g_myglobal = 42;
    return 0;
}
```

See also How do I use **`extern`** to share variables between source files?

Section 44.3: Introduction

Example of declarations are:

```
int a; /* declaring single identifier of type int */
```

The above declaration declares single identifier named `a` which refers to some object with `int` type.

```
int a1, b1; /* declaring 2 identifiers of type int */
```

The second declaration declares 2 identifiers named `a1` and `b1` which refers to some other objects though with the same `int` type.

Basically, the way this works is like this - first you put some type, then you write a single or multiple expressions separated via comma (`,`) (**which will not be evaluated at this point - and which should otherwise be referred to as declarators in this context**). In writing such expressions, you are allowed to apply only the indirection (`*`), function call (`( )`) or subscript (or array indexing - `[ ]`) operators onto some identifier (you can also not use any operators at all). The identifier used is not required to be visible in the current scope. Some examples:

```
/* 1 */ int /* 2 */ (*z) /* 3 */ , /* 4 */ *x , /* 5 */ **c /* 6 */ ;
```

#	Description
1	The name of integer type.
2	Un-evaluated expression applying indirection to some identifier `z`.
3	We have a comma indicating that one more expression will follow in the same declaration.
4	Un-evaluated expression applying indirection to some other identifier `x`.
5	Un-evaluated expression applying indirection to the value of the expression `(*c)`.
6	End of declaration.

Note that none of the above identifiers were visible prior to this declaration and so the expressions used would not be valid before it.

After each such expression, the identifier used in it is introduced into the current scope. (If the identifier has assigned linkage to it, it may also be re-declared with the same type of linkage so that both identifiers refer to the same object or function)

Additionally, the equal operator sign (`=`) may be used for initialization. If an unevaluated expression (declarator) is followed by `=` inside the declaration - we say that the identifier being introduced is also being initialized. After the `=` sign we can put once again some expression, but this time it'll be evaluated and its value will be used as initial for the object declared.

Examples:

```
int l = 90; /* the same as: */

int l; l = 90; /* if it the declaration of l was in block scope */

int c = 2, b[c]; /* ok, equivalent to: */

int c = 2; int b[c];
```

Later in your code, you are allowed to write the exact same expression from the declaration part of the newly introduced identifier, giving you an object of the type specified at the beginning of the declaration, assuming that you've assigned valid values to all accessed objects in the way. Examples:

```
void f()
{
    int b2; /* you should be able to write later in your code b2
            which will directly refer to the integer object
            that b2 identifies */

    b2 = 2; /* assign a value to b2 */

    printf("%d", b2); /*ok - should print 2*/

    int *b3; /* you should be able to write later in your code *b3 */
```

```
    b3 = &b2; /* assign valid pointer value to b3 */

    printf("%d", *b3); /* ok - should print 2 */

    int **b4; /* you should be able to write later in your code **b4 */

    b4 = &b3;

    printf("%d", **b4); /* ok - should print 2 */

    void (*p)(); /* you should be able to write later in your code (*p)() */

    p = &f; /* assign a valid pointer value */

    (*p)(); /* ok - calls function f by retrieving the
            pointer value inside p -    p
            and dereferencing it -      *p
            resulting in a function
            which is then called -      (*p)() -

            it is not *p() because else first the () operator is
            applied to p and then the resulting void object is
            dereferenced which is not what we want here */
}
```

The declaration of `b3` specifies that you can potentially use `b3` value as a mean to access some integer object.

Of course, in order to apply indirection (`*`) to `b3`, you should also have a proper value stored in it (see pointers for more info). You should also first store some value into an object before trying to retrieve it (you can see more about this problem here). We've done all of this in the above examples.

```
int a3(); /* you should be able to call a3 */
```

This one tells the compiler that you'll attempt to call `a3`. In this case `a3` refers to function instead of an object. One difference between object and function is that functions will always have some sort of linkage. Examples:

```
void f1()
{
    {
        int f2(); /* 1 refers to some function f2 */
    }

    {
        int f2(); /* refers to the exact same function f2 as (1) */
    }
}
```

In the above example, the 2 declarations refer to the same function `f2`, whilst if they were declaring objects then in this context (having 2 different block scopes), they would have be 2 different distinct objects.

```
int (*a3)(); /* you should be able to apply indirection to `a3` and then call it */
```

Now it may seems to be getting complicated, but if you know operators precedence you'll have 0 problems reading the above declaration. The parentheses are needed because the `*` operator has less precedence then the `( )` one.

In the case of using the subscript operator, the resulting expression wouldn't be actually valid after the declaration because the index used in it (the value inside `[` and `]`) will always be 1 above the maximum allowed value for this object/function.

```
int a4[5]; /* here a4 shouldn't be accessed using the index 5 later on */
```

But it should be accessible by all other indexes lower then 5. Examples:

```
a4[0], a4[1]; a4[4];
```

`a4[5]` will result into UB. More information about arrays can be found here.

```
int (*a5)[5](); /* here a4 could be applied indirection
                   indexed up to (but not including) 5
                   and called */
```

Unfortunately for us, although syntactically possible, the declaration of `a5` is forbidden by the current standard.

Section 44.4: Typedef

Typedefs are declarations which have the keyword `typedef` in front and before the type. E.g.:

```
typedef int (*(*t0)())[5];
```

(*you can technically put the typedef after the type too - like this* `int typedef (*(*t0)())[5];` *but this is discouraged*)

The above declarations declares an identifier for a typedef name. You can use it like this afterwards:

```
t0 pf;
```

Which will have the same effect as writing:

```
int (*(*pf)())[5];
```

As you can see the typedef name "saves" the declaration as a type to use later for other declarations. This way you can save some keystrokes. Also as declaration using `typedef` is still a declaration you are not limited only by the above example:

```
t0 (*pf1);
```

Is the same as:

```
int (*(**pf1)())[5];
```

Section 44.5: Using Global Constants

Headers may be used to declare globally used read-only resources, like string tables for example.

Declare those in a separate header which gets included by any file ("*Translation Unit*") which wants to make use of them. It's handy to use the same header to declare a related enumeration to identify all string-resources:

resources.h:

```
#ifndef RESOURCES_H
#define RESOURCES_H

typedef enum { /* Define a type describing the possible valid resource IDs. */
  RESOURCE_UNDEFINED = -1, /* To be used to initialise any EnumResourceID typed variable to be
```

```
                              marked as "not in use", "not in list", "undefined", wtf.
                              Will say un-initialised on application level, not on language level.
Initialised uninitialised, so to say ;-)
                              Its like NULL for pointers ;-)*/
  RESOURCE_UNKNOWN = 0,     /* To be used if the application uses some resource ID,
                              for which we do not have a table entry defined, a fall back in
                              case we _need_ to display something, but do not find anything
                              appropriate. */

  /* The following identify the resources we have defined: */
  RESOURCE_OK,
  RESOURCE_CANCEL,
  RESOURCE_ABORT,
  /* Insert more here. */

  RESOURCE_MAX /* The maximum number of resources defined. */
} EnumResourceID;


extern const char * const resources[RESOURCE_MAX]; /* Declare, promise to anybody who includes
                                      this, that at linkage-time this symbol will be around.
                                      The 1st const guarantees the strings will not change,
                                      the 2nd const guarantees the string-table entries
                                      will never suddenly point somewhere else as set during
                                      initialisation. */
#endif
```

To actually define the resources created a related .c-file, that is another translation unit holding the actual instances of the what had been declared in the related header (.h) file:

resources.c:

```
#include "resources.h" /* To make sure clashes between declaration and definition are
                          recognised by the compiler include the declaring header into
                          the implementing, defining translation unit (.c file).

/* Define the resources. Keep the promise made in resources.h. */
const char * const resources[RESOURCE_MAX] = {
  "<unknown>",
  "OK",
  "Cancel",
  "Abort"
};
```

A program using this could look like this:

main.c:

```
#include <stdlib.h> /* for EXIT_SUCCESS */
#include <stdio.h>

#include "resources.h"


int main(void)
{
  EnumResourceID resource_id = RESOURCE_UNDEFINED;

  while ((++resource_id) < RESOURCE_MAX)
  {
```

```
    printf("resource ID: %d, resource: '%s'\n", resource_id, resources[resource_id]);
  }

  return EXIT_SUCCESS;
}
```

Compile the three file above using GCC, and link them to become the program file `main` for example using this:

```
gcc -Wall -Wextra -pedantic -Wconversion -g  main.c resources.c -o main
```

(use these `-Wall -Wextra -pedantic -Wconversion` to make the compiler really picky, so you don't miss anything before posting the code to SO, will say the world, or even worth deploying it into production)

Run the program created:

```
$ ./main
```

And get:

```
resource ID: 0, resource: ''
resource ID: 1, resource: 'OK'
resource ID: 2, resource: 'Cancel'
resource ID: 3, resource: 'Abort'
```

Section 44.6: Using the right-left or spiral rule to decipher C declaration

The "right-left" rule is a completely regular rule for deciphering C declarations. It can also be useful in creating them.

Read the symbols as you encounter them in the declaration...

```
*   as "pointer to"          - always on the left side
[]  as "array of"            - always on the right side
()  as "function returning"  - always on the right side
```

How to apply the rule

STEP 1

Find the identifier. This is your starting point. Then say to yourself, "identifier is." You've started your declaration.

STEP 2

Look at the symbols on the right of the identifier. If, say, you find `()` there, then you know that this is the declaration for a function. So you would then have *"identifier is function returning"*. Or if you found a `[]` there, you would say *"identifier is array of"*. Continue right until you run out of symbols OR hit a right parenthesis `)`. (If you hit a left parenthesis `(`, that's the beginning of a `()` symbol, even if there is stuff in between the parentheses. More on that below.)

STEP 3

Look at the symbols to the left of the identifier. If it is not one of our symbols above (say, something like "int"), just say it. Otherwise, translate it into English using that table above. Keep going left until you run out of symbols OR hit

a left parenthesis (.

Now repeat steps 2 and 3 until you've formed your declaration.

Here are some examples:

```
int *p[];
```

First, find identifier:

```
int *p[];
     ^
```

"p is"

Now, move right until out of symbols or right parenthesis hit.

```
int *p[];
      ^^
```

"p is array of"

Can't move right anymore (out of symbols), so move left and find:

```
int *p[];
    ^
```

"p is array of pointer to"

Keep going left and find:

```
int *p[];
^^^
```

"p is array of pointer to int".

(or *"p is an array where each element is of type pointer to int"*)

Another example:

```
int *(*func())();
```

Find the identifier.

```
int *(*func())();
       ^^^^
```

"func is"

Move right.

```
int *(*func())();
           ^^
```

"func is function returning"

Can't move right anymore because of the right parenthesis, so move left.

```
int *(*func())();
      ^
```

"func is function returning pointer to"

Can't move left anymore because of the left parenthesis, so keep going right.

```
int *(*func())();
              ^^
```

"func is function returning pointer to function returning"

Can't move right anymore because we're out of symbols, so go left.

```
int *(*func())();
    ^
```

"func is function returning pointer to function returning pointer to"

And finally, keep going left, because there's nothing left on the right.

```
int *(*func())();
^^^
```

"func is function returning pointer to function returning pointer to int".

As you can see, this rule can be quite useful. You can also use it to sanity check yourself while you are creating declarations, and to give you a hint about where to put the next symbol and whether parentheses are required.

Some declarations look much more complicated than they are due to array sizes and argument lists in prototype form. If you see `[3]`, that's read as *"array (size 3) of..."*. If you see `(char *,int)` that's read as *"function expecting (char *,int) and returning..."*.

Here's a fun one:

```
int (*(*fun_one)(char *,double))[9][20];
```

I won't go through each of the steps to decipher this one.

*"fun_one is pointer to function expecting (char *,double) and returning pointer to array (size 9) of array (size 20) of int."*

As you can see, it's not as complicated if you get rid of the array sizes and argument lists:

```
int (*(*fun_one)())[][];
```

You can decipher it that way, and then put in the array sizes and argument lists later.

Some final words:

It is quite possible to make illegal declarations using this rule, so some knowledge of what's legal in C is necessary. For instance, if the above had been:

```
int *((*fun_one)())[][];
```

it would have read *"fun_one is pointer to function returning array of array of pointer to int"*. Since a function cannot return an array, but only a pointer to an array, that declaration is illegal.

Illegal combinations include:

```
[]() - cannot have an array of functions
()() - cannot have a function that returns a function
()[] - cannot have a function that returns an array
```

In all the above cases, you would need a set of parentheses to bind a `*` symbol on the left between these `()` and `[]` right-side symbols in order for the declaration to be legal.

Here are some more examples:

Legal

```
int i;              an int
int *p;             an int pointer (ptr to an int)
int a[];            an array of ints
int f();            a function returning an int
int **pp;           a pointer to an int pointer (ptr to a ptr to an int)
int (*pa)[];        a pointer to an array of ints
int (*pf)();        a pointer to a function returning an int
int *ap[];          an array of int pointers (array of ptrs to ints)
int aa[][];         an array of arrays of ints
int *fp();          a function returning an int pointer
int ***ppp;         a pointer to a pointer to an int pointer
int (**ppa)[];      a pointer to a pointer to an array of ints
int (**ppf)();      a pointer to a pointer to a function returning an int
int *(*pap)[];      a pointer to an array of int pointers
int (*paa)[][];     a pointer to an array of arrays of ints
int *(*pfp)();      a pointer to a function returning an int pointer
int **app[];        an array of pointers to int pointers
int (*apa[])[];     an array of pointers to arrays of ints
int (*apf[])();     an array of pointers to functions returning an int
int *aap[][];       an array of arrays of int pointers
int aaa[][][];      an array of arrays of arrays of int
int **fpp();        a function returning a pointer to an int pointer
int (*fpa())[];     a function returning a pointer to an array of ints
int (*fpf())();     a function returning a pointer to a function returning an int
```

Illegal

```
int af[]();         an array of functions returning an int
int fa()[];         a function returning an array of ints
int ff()();         a function returning a function returning an int
int (*pfa)()[];     a pointer to a function returning an array of ints
int aaf[][]();      an array of arrays of functions returning an int
int (*paf)[]();     a pointer to a an array of functions returning an int
int (*pff)()();     a pointer to a function returning a function returning an int
int *afp[]();       an array of functions returning int pointers
int afa[]()[];      an array of functions returning an array of ints
int aff[]()();      an array of functions returning functions returning an int
int *fap()[];       a function returning an array of int pointers
int faa()[][];      a function returning an array of arrays of ints
int faf()[]();      a function returning an array of functions returning an int
int *ffp()();       a function returning a function returning an int pointer
```

Source: http://ieng9.ucsd.edu/~cs30x/rt_lt.rule.html