Chapter 39: Identifier Scope

Section 39.1: Function Prototype Scope

```
#include <stdio.h>

/* The parameter name, apple, has function prototype scope.  These names
   are not significant outside the prototype itself.  This is demonstrated
   below. */

int test_function(int apple);

int main(void)
{
    int orange = 5;

    orange = test_function(orange);
    printf("%d\r\n", orange); //orange = 6

    return 0;
}

int test_function(int fruit)
{
    fruit += 1;
    return fruit;
}
```

Note that you get puzzling error messages if you introduce a type name in a prototype:

```
int function(struct whatever *arg);

struct whatever
{
    int a;
    // ...
};

int function(struct whatever *arg)
{
    return arg->a;
}
```

With GCC 6.3.0, this code (source file `dc11.c`) produces:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -c dc11.c
dc11.c:1:25: error: ‘struct whatever’ declared inside parameter list will not be visible outside of
this definition or declaration [-Werror]
     int function(struct whatever *arg);
                         ^~~~~~~~
dc11.c:9:9: error: conflicting types for ‘function’
     int function(struct whatever *arg)
         ^~~~~~~~
dc11.c:1:9: note: previous declaration of ‘function’ was here
     int function(struct whatever *arg);
         ^~~~~~~~
cc1: all warnings being treated as errors
$
```

Place the structure definition before the function declaration, or add `struct whatever;` as a line before the function declaration, and there is no problem. You should not introduce new type names in a function prototype because there's no way to use that type, and hence no way to define or use that function.

Section 39.2: Block Scope

An identifier has block scope if its corresponding declaration appears inside a block (parameter declaration in function definition apply). The scope ends at the end of the corresponding block.

No different entities with the same identifier can have the same scope, but scopes may overlap. In case of overlapping scopes the only visible one is the one declared in the innermost scope.

```
#include <stdio.h>

void test(int bar)                    // bar has scope test function block
{
    int foo = 5;                      // foo has scope test function block
    {
        int bar = 10;                 // bar has scope inner block, this overlaps with previous
test:bar declaration, and it hides test:bar
        printf("%d %d\n", foo, bar); // 5 10
    }                                 // end of scope for inner bar
    printf("%d %d\n", foo, bar);      // 5 5, here bar is test:bar
}                                     // end of scope for test:foo and test:bar

int main(void)
{
    int foo = 3;          // foo has scope main function block

    printf("%d\n", foo); // 3
    test(5);
    printf("%d\n", foo); // 3
    return 0;
}                         // end of scope for main:foo
```

Section 39.3: File Scope

```
#include <stdio.h>

/* The identifier, foo, is declared outside all blocks.
   It can be used anywhere after the declaration until the end of
   the translation unit. */
static int foo;

void test_function(void)
{
    foo += 2;
}

int main(void)
{
    foo = 1;

    test_function();
    printf("%d\r\n", foo); //foo = 3;

    return 0;
}
```

Section 39.4: Function scope

Function scope is the special scope for **labels**. This is due to their unusual property. A **label** is visible through the entire function it is defined and one can jump (using instruction `goto`*`label`*) to it from any point in the same function. While not useful, the following example illustrate the point:

```
#include <stdio.h>

int main(int argc,char *argv[]) {
    int a = 0;
    goto INSIDE;
  OUTSIDE:
    if (a!=0) {
        int i=0;
      INSIDE:
        printf("a=%d\n",a);
        goto OUTSIDE;
    }
}
```

`INSIDE` may seem defined *inside* the `if` block, as it is the case for `i` which scope is the block, but it is not. It is visible in the whole function as the instruction `goto INSIDE;` illustrates. Thus there can't be two labels with the same identifier in a single function.

A possible usage is the following pattern to realize correct complex cleanups of allocated ressources:

```
#include <stdlib.h>
#include <stdio.h>

void a_function(void) {
   double* a = malloc(sizeof(double[34]));
   if (!a) {
      fprintf(stderr,"can't allocate\n");
      return;                 /* No point in freeing a if it is null */
   }
   FILE* b = fopen("some_file","r");
   if (!b) {
      fprintf(stderr,"can't open\n");
      goto CLEANUP1;          /* Free a; no point in closing b */
   }
   /* do something reasonable */
   if (error) {
      fprintf(stderr,"something's wrong\n");
      goto CLEANUP2;       /* Free a and close b to prevent leaks */
   }
   /* do yet something else */
CLEANUP2:
   close(b);
CLEANUP1:
   free(a);
}
```

Labels such as `CLEANUP1` and `CLEANUP2` are special identifiers that behave differently from all other identifiers. They are visible from everywhere inside the function, even in places that are executed before the labeled statement, or even in places that could never be reached if none of the `goto` is executed. Labels are often written in lower-case rather than upper-case.

Chapter 40: Implicit and Explicit Conversions

Section 40.1: Integer Conversions in Function Calls

Given that the function has a proper prototype, integers are widened for calls to functions according to the rules of integer conversion, C11 6.3.1.3.

> **6.3.1.3 Signed and unsigned integers**
> When a value with integer type is converted to another integer type other than _Bool, if the value can be represented by the new type, it is unchanged.
>
> Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.
>
> Otherwise, the new type is signed and the value cannot be represented in it; either the result is implementation-defined or an implementation-defined signal is raised.

Usually you should not truncate a wide signed type to a narrower signed type, because obviously the values can't fit and there is no clear meaning that this should have. The C standard cited above defines these cases to be "implementation-defined", that is, they are not portable.

The following example supposes that `int` is 32 bit wide.

```
#include <stdio.h>
#include <stdint.h>

void param_u8(uint8_t val) {
    printf("%s val is %d\n", __func__, val);  /* val is promoted to int */
}

void param_u16(uint16_t val) {
    printf("%s val is %d\n", __func__, val);  /* val is promoted to int */
}

void param_u32(uint32_t val) {
    printf("%s val is %u\n", __func__, val);  /* here val fits into unsigned */
}

void param_u64(uint64_t val) {
    printf("%s val is " PRI64u "\n", __func__, val); /* Fixed with format string */
}

void param_s8(int8_t val) {
    printf("%s val is %d\n", __func__, val);  /* val is promoted to int */
}

void param_s16(int16_t val) {
    printf("%s val is %d\n", __func__, val);  /* val is promoted to int */
}

void param_s32(int32_t val) {
    printf("%s val is %d\n", __func__, val); /* val has same width as int */
}
```