Chapter 43: Storage Classes

A storage class is used to set the scope of a variable or function. By knowing the storage class of a variable, we can determine the life-time of that variable during the run-time of the program.

Section 43.1: auto

This storage class denotes that an identifier has automatic storage duration. This means once the scope in which the identifier was defined ends, the object denoted by the identifier is no longer valid.

Since all objects, not living in global scope or being declared `static`, have automatic storage duration by default when defined, this keyword is mostly of historical interest and should not be used:

```
int foo(void)
{
    /* An integer with automatic storage duration. */
    auto int i = 3;

    /* Same */
    int j = 5;

    return 0;
} /* The values of i and j are no longer able to be used. */
```

Section 43.2: register

Hints to the compiler that access to an object should be as fast as possible. Whether the compiler actually uses the hint is implementation-defined; it may simply treat it as equivalent to `auto`.

The only property that is definitively different for all objects that are declared with `register` is that they cannot have their address computed. Thereby `register` can be a good tool to ensure certain optimizations:

```
register size_t size = 467;
```

is an object that can never *alias* because no code can pass its address to another function where it might be changed unexpectedly.

This property also implies that an array

```
register int array[5];
```

cannot decay into a pointer to its first element (i.e. `array` turning into `&array[0]`). This means that the elements of such an array cannot be accessed and the array itself cannot be passed to a function.

In fact, the only legal usage of an array declared with a `register` storage class is the `sizeof` operator; any other operator would require the address of the first element of the array. For that reason, arrays generally should not be declared with the `register` keyword since it makes them useless for anything other than size computation of the entire array, which can be done just as easily without the `register` keyword.

The `register` storage class is more appropriate for variables that are defined inside a block and are accessed with high frequency. For example,

```
/* prints the sum of the first 5 integers*/
/* code assumed to be part of a function body*/
```

```
{
    register int k, sum;
    for(k = 1, sum = 0; k < 6; sum += k, k++);
        printf("\t%d\n",sum);
}
```

Version ≥ C11

The `_Alignof` operator is also allowed to be used with `register` arrays.

Section 43.3: static

The `static` storage class serves different purposes, depending on the location of the declaration in the file:

1. To confine the identifier to that translation unit only (scope=file).

```
/* No other translation unit can use this variable. */
static int i;

/* Same; static is attached to the function type of f, not the return type int. */
static int f(int n);
```

2. To save data for use with the next call of a function (scope=block):

```
void foo()
{
    static int a = 0; /* has static storage duration and its lifetime is the
                       * entire execution of the program; initialized to 0 on
                       * first function call */
    int b = 0; /* b has block scope and has automatic storage duration and
                * only "exists" within function */

    a += 10;
    b += 10;

    printf("static int a = %d, int b = %d\n", a, b);
}

int main(void)
{
    int i;
    for (i = 0; i < 5; i++)
    {
        foo();
    }

    return 0;
}
```

This code prints:

```
static int a = 10, int b = 10
static int a = 20, int b = 10
static int a = 30, int b = 10
static int a = 40, int b = 10
static int a = 50, int b = 10
```

Static variables retain their value even when called from multiple different threads.

Version ≥ C99

3. Used in function parameters to denote an array is expected to have a constant minimum number of elements and a non-null parameter:

```
/* a is expected to have at least 512 elements. */
void printInts(int a[static 512])
{
    size_t i;
    for (i = 0; i < 512; ++i)
        printf("%d\n", a[i]);
}
```

The required number of items (or even a non-null pointer) is not necessarily checked by the compiler, and compilers are not required to notify you in any way if you don't have enough elements. If a programmer passes fewer than 512 elements or a null pointer, undefined behavior is the result. Since it is impossible to enforce this, extra care must be used when passing a value for that parameter to such a function.

Section 43.4: typedef

Defines a new type based on an existing type. Its syntax mirrors that of a variable declaration.

```
/* Byte can be used wherever `unsigned char` is needed */
typedef unsigned char Byte;

/* Integer is the type used to declare an array consisting of a single int */
typedef int Integer[1];

/* NodeRef is a type used for pointers to a structure type with the tag "node" */
typedef struct node *NodeRef;

/* SigHandler is the function pointer type that gets passed to the signal function. */
typedef void (*SigHandler)(int);
```

While not technically a storage class, a compiler will treat it as one since none of the other storage classes are allowed if the `typedef` keyword is used.

The `typedef`s are important and should not be substituted with `#define` macro.

```
typedef int newType;
newType *ptr;        // ptr is pointer to variable of type 'newType' aka int
```

However,

```
#define int newType
newType *ptr;        // Even though macros are exact replacements to words, this doesn't result to
a pointer to variable of type 'newType' aka int
```

Section 43.5: extern

Used to **declare an object or function** that is defined elsewhere (and that has *external linkage*). In general, it is used to declare an object or function to be used in a module that is not the one in which the corresponding object or function is defined:

```c
/* file1.c */
int foo = 2;  /* Has external linkage since it is declared at file scope. */
```

```c
/* file2.c */
#include <stdio.h>
int main(void)
{
    /* `extern` keyword refers to external definition of `foo`. */
    extern int foo;
    printf("%d\n", foo);
    return 0;
}
```

Version ≥ C99

Things get slightly more interesting with the introduction of the **`inline`** keyword in C99:

```c
/* Should usually be place in a header file such that all users see the definition */
/* Hints to the compiler that the function `bar` might be inlined */
/* and suppresses the generation of an external symbol, unless stated otherwise. */
inline void bar(int drink)
{
    printf("You ordered drink no.%d\n", drink);
}

/* To be found in just one .c file.
   Creates an external function definition of `bar` for use by other files.
   The compiler is allowed to choose between the inline version and the external
   definition when `bar` is called. Without this line, `bar` would only be an inline
   function, and other files would not be able to call it. */
extern void bar(int);
```

Section 43.6: _Thread_local

Version ≥ C11

This was a new storage specifier introduced in C11 along with multi-threading. This isn't available in earlier C standards.

Denotes *thread storage duration*. A variable declared with `_Thread_local` storage specifier denotes that the object is *local to that thread* and its lifetime is the entire execution of the thread in which it's created. It can also appear along with `static` or **`extern`**.

```c
#include <threads.h>
#include <stdio.h>
#define SIZE 5

int thread_func(void *id)
{
    /* thread local variable i. */
    static _Thread_local int i;

    /* Prints the ID passed from main() and the address of the i.
     * Running this program will print different addresses for i, showing
     * that they are all distinct objects. */
    printf("From thread:[%d], Address of i (thread local): %p\n", *(int*)id, (void*)&i);

    return 0;
}

int main(void)
```

```
{
    thrd_t id[SIZE];
    int arr[SIZE] = {1, 2, 3, 4, 5};

    /* create 5 threads. */
    for(int i = 0; i < SIZE; i++) {
        thrd_create(&id[i], thread_func, &arr[i]);
    }

    /* wait for threads to complete. */
    for(int i = 0; i < SIZE; i++) {
        thrd_join(id[i], NULL);
    }
}
```