1 Storage Class Specifiers

The C language contains four keywords that let you specify how a variable is stored in memory. These are as follows:

```
auto
extern
register
static
```

The first of these keywords, `auto`, is almost never seen in a C program. This is because it is used to tell the compiler that the variable being declared is an automatic variable, and it should be destroyed when program execution leaves the current scope. But, as you should know, this is already the default behavior. However, if you really really wanted to use it, here is how it would be used:

```
int main(void)
{
  auto int i;

  ...
}
```

The second keyword, `extern` is used when you write a program that is split across more than one file. As you know, programs can have global variables. But, if a global variable needs to be used in two (or more) files of the same program, we have a problem. You can not declare the variable in both files, because variable names must be unique per program, not per file. You can not use the variable without declaring it either. So what do you do? The answer is, you declare the variable normally in only one file, and in other files, you declare it using the `extern` keyword. This tells the compiler that the variable exists, but not to actually allocate room for the variable, because it will be used from an external source.

The third keyword, `register`, is meant as a helper for the compiler in optimization. If you prepend the keyword `register` to any variable name, it is a hint for the compiler to actually store that variable in one of the registers of the CPU. Such variables can be accessed much faster than variables that are actually stored in memory. However, these days such decisions are usually left to the compiler, which does a better job than humans at figuring out what variables should be stored in registers. A side effect of a variable being declared as `register` is that it will not have an address in memory, thus you can not use the "address of" operator (`&`) on the variable.

The fourth and last keyword, when prepended to a local variable declaration means that the variable should not be destroyed when execution leaves scope, but that it should be static. This means that such variables will keep their value when execution returns to the same scope.

2 Access Modifiers

C contains two additional keywords, letting the programmer change how the compiler treats access to variables. These are `const` and `volatile`.

You have already seen `const` used in code. It has two main uses. First, when applied to a regular variable (not a pointer), it means that your program will not be able to modify the value of that variable. But, of course, you can (and actually should) give that variable an initial value by use of an initializer. The second use is using `const` with a pointer, which is usually done with function parameters. In that case, the program will be unable to modify the *value* pointed to by the pointer.

The `volatile` probably will sound useless, but sometimes it is vital. By declaring a variable to be `volatile`, you are telling the compiler that the value of that variable may change by agents external to the program. This causes the compiler not to assume that the variable will not change value unless it generates code that modifies it. This is necessary, say, in case you are writing input/output code and one of your variables is in a well-known memory location where values are placed automatically by the hardware.

3 Enumerations

Sometimes, you may need to define a series named integer constants to represent some sort of data. A typical example could be the names of the days fof the week. Here is how you would do it:

```
enum day_t {
    sunday,
    monday,
    tuesday,
    wednesday,
    thursday,
    friday,
    saturday
};
```

Note that these are just names for integer constants. But what integer constants? If declared as above, the compiler automatically assigns integer values to each name starting at zero. Thus `sunday` would be zero, `monday` would be one, and so on.

Optionally, you can assign your own values to the variables as follows:

```
enum day_t {
    sunday = 4,
    monday = 7,
    tuesday = 9,
    wednesday = 14,
    thursday = 44,
    friday = 59,
    saturday = 61
};
```

Then, you can declare (and use) a variable of this `enum`eration type as follows:

```
enum day_t day;

day = tuesday;
```

4 typedef

Note that when we define structures or enumerations, we have to use the full type name. So, if you defined `struct vector_3d` to hold data for a three dimensional real vector, you have to type:

```
struct vector_3d v1;
\begin{verbatim}

This can get tiresome. In addition, in some instances you may want to
use a type which you can change later. For such cases, the C language
contains the \texttt{typedef} keyword. It allows you assign a new
\emph{name} to an existing data type. The general form is as follows:

\begin{verbatim}
typedef <old-name> <new-name>;
```

So, an example would be:

```
typedef struct vector_3d vec3_t;

vec3_t v1;
```

This has the same effect as the previous declaration. Note that `typedef` statements go in the global portion of a C file.

5 Unions

Almost in every C book, structures and unions are introduced together. I did not follow the example, because I think structures are much more fundamental for programming than unions. However, I do think that you should have an idea about what they are, so here we go:

Unions are similar to strutctures in the way they are defined. Here is a union definition:

```
union my_union {
  int a;
  float b;
};
```

You might think that you can use this union to store an `int` called `a` and a `float` called `b`. But this is not the case. The case is that you can use this union to store *either* an `int` called `a` *or* a `float` called `b`. The two variables share the same memory location. This would still be true if we had more than two variables.