Chapter 4: Function and Program Structure

- Functions break large computing tasks into smaller ones.
- Allow others to build on previous work.
- Hide details of operation from parts of the program that don't need to know about them.
- Use many small functions instead of few large ones.

4.1 Basics of functions

- Communication between functions is by arguments and values returned.
    - Also through external variables.
- Functions can be put in any order in the source file.
- The program can be split up between multiple files.
    - The functions can't be split.
- The *return* statement is how a value is sent from the function to the caller.
    - Syntax `return expression;`
    - The expression will be converted to the return type of the function if necessary.

- The calling function may ignore the returned value.

    - Execution is returned to the caller when the program execution reaches the terminating right brace of the function.

- **Compiling multiple files**

    - We can do so on the command line or in a make file.

4.2 Functions Returning Non-Integers

- functions can return other value types than void or int.

4.3 External Variables

- A C program consists of a set of external objects.
    - Either variables or functions.
- External variables are defined outside of any function.
    - They are potentially available to many functions.
- Functions are always external because C does not allow internal functions.
- All external variables and functions have the property that all references to them by the same name are references to the same thing.
- External variables are globally accessible
    - Provide an alternative to function arguments.
    - Also provide an alternative to return values for communicating data between functions.
    - If a large number of variables must be shared among functions external variables are more convenient and efficient.
        - Can cause issues so must be done with care.
        - Too many data connections between functions can cause problems.
- External variables have greater scope and lifetime.
    - Internal/Automatic variables are only in existence while the function is active.
    - If two functions must share data but do not call each other global variables offer a way of doing so.

4.4 Scope Rules

- The functions and external variable that make up a C program need not all be compiled at the same time.
    - There are some questions of interest about this:
        - How are declarations written so that variables are properly declared during compilation?
        - How are declarations arranged so that all the pieces will be properly connected when the program is loaded?
        - How are declarations organized so there is only one copy?
        - How are external variables initialized?
- The *scope* of a name is the part of the program within which the name can be used.
    - For a local/automatic variable declared at the beginning of a function the scope is the function in which it is declared.
        - Local variables of the same name in different functions are unrelated.
    - The scope of an external variable or function lasts from the point at which it is declared to the end of the file being compiled.
        - If an external variable is to be referred to before it is defined, or defined in a different source file, the use of an *extern* declaration is mandatory.
- It is important to distinguish between the *declaration* of an external variable and the *definition*.
    - *declaration* announces the properties of the variable.
    - *definition* causes storage to be set aside.
        - There must be only one definition of an external variable among all the files that make up the source program.
    - Example:
        - *definition*: `int sp;`
        - *declaration*: `extern int sp;`

4.5 Header files

- We can divide a program into several source files.
- We can put the definitions and declarations of functions in a header file.
    - Up to moderate file size it's best to have one header file.

4.6 Static Variables

- The *static* declaration applied to an external variable or function limits the scope of that object to the rest of the source file being compiled.
- External static provides a way to hide names so they can be shared but not visible to users of a function.
- Prefix a normal declaration by the keyword *static* and no other routine will be able to access the variable.
    - The names will not conflict with other names outside the source file either.
    - Can be used for functions as well but normally function names are global.

4.7 Register Variables.

- A *register* declaration advises the compiler that the variable in question will be heavily used.
    - *register* variables are flagged for storage in the computer registers but this suggestion may be ignored by compilers.
- May only be applied to automatic variables.
- There are practicalities based on the underlying hardware for these:
    - Only a few register variables in each function may be kept in registers.
    - Only certain types are allowed.
    - Declaring too many doesn't make much difference as they can be ignored.
    - It is not possible to take the address of a register variable.

        - Even if the variable is not stored in a register.
- The specific types and number of these variables is different from machine to machine.

4.8 Block Structure

- We can't define functions inside of functions so block structure doesn't happen for functions in C.
- We can use block structure for variables in C.
    - Defining variables inside if statements for example.
- Its best to not use local variables of the same name as global variables.

4.9 Initialization.

- In the absence of explicit Initialization external and static variables are guaranteed to be initialized to zero.
- Automatic and register variables are initialized to garbage unless explicitly initialized.
- Scalar variables may be initialized when they are defined.
- External and Static variables must be initialized with a constant expression.
    - Done once before the program begins execution.
- For automatic or register variables Initialization is done when the function or block is entered.
- When an array is initialized the compiler will define the length based on the number of initializers (if the size isn't specified).
    - If there are fewer initializers for an array than the number of initializers the remaining elements will be set to zero.
    - It is an error to have more initializers than the length of the array.
    - A character array may be initialized with a string.
        - Note character arrays include a terminating null char.

4.10 Recursion

- C functions may be used recursively.
  - The function may call itself.

4.11 The C Preprocessor

- C provides certain language functionality by means of a preprocessor.
  - The preprocessor is conceptually a separate first step in compilation.
- There are two frequently used features:
  - '"#include"' to include the contents of a file during compilation.
  - '"#define"' to replace a token by an arbitrary sequence of characters.

File Inclusion

- File inclusion makes it easy to handle collections of #defines and declarations
- There are often several #include lines at the beginning of a source file.
  - They include common #define statements and extern declarations.
  - They may also access library functions.
- #include is the preferred way to tie the declarations together for a large program.
  - It guarantees that all source files will be supplied with the same definitions and variable declarations.
  - When the included file is changed all the files depending on it must be recompiled.

Macro Substitution

- The #define command is the simplest form of a macro substitution.

    - If the replacement text is longer than one line the \ character may be used to continue on to the next line.
- The scope of the name defined with #define is from its point of the definition to the end of the source file being compiled.
- Substitutions are made for tokens and do not take place with quoted strings.
    - i.e. if the replacement is for YES it would not be replaced in '"printf("YES");"'
    - It will also not be replaced as a substring of another string.
- We can replace any name with any replacement text.
    - Even in line code can be used.
    - When we use this for in line code we have to be careful about unintended side effects.
- In getchar and putchar are defined as macros.
    - Saves runtime overhead of a function call.
- We can undefine a name by using #undef

Conditional inclusion

- We can control preprocessing with conditional statements.
    - Allows inclusion of code selectively.