Functions and Program Structure

Basic of Functions

```c
return_type function_name(argument declarations) {
    declarations and statements

    return expression;
}
```

Functions Returning Non-Integers

External Variables

- A C program consists of a set of external objects, which are either variables or functions.
- External variables are defined outside of functions.
- Functions themselves are always external, because C does not allow functions to be defined inside other functions.
- External variables are globally accessible.
- Automatic variables are internal to a function. They come into existence when the function is entered, and disappear when it is left.
- External variables are permanent.

Scope Rules

- The scope of a name is the part of the program within which the name can be used.
- For an automatic variable declared at the beginning of a function, the scope is the function in which the name is declared.
- Local variables of the same name in different functions are unrelated.
- Parameters of the function are in effect local variables.
- The scope of an external variables or a function lasts from the point at which it is declared to the end of the file being compiled.
- The *definition* of the external variables causes storage to be set aside, and also serve as the declaration for the rest of that source file.

```c
int external_var;
double external_arr[MAXSIZE];
```

- The *declaration* doesn't create the variables or reserve storage for them

```c
extern int external_var;
double external_arr[MAXSIZE];
```

- There must be only one definition of an external variable among all the files that make up the source program.
- Other files may contain **`extern`** to access it.
- Initialization of an external variables goes only with the definition.

Header Files

- Those definitions and declarations shared among the files can be placed in a *header file*.

Static Variables

- **`static`** for extern variables

Suppose we have a extern variable named **`var`** without **`static`** declaration in the file **`foo.c`** as follow.

```c
// foo.c
int var = 2333;
// end foo.c

// foo.h
extern int var;
// end foo.h

// main.c
#include <stdio.h>
#include "foo.h"

int main() {
    printf("var is %d\n", var);
}
// end main.c
```

Now the variable **`var`** is visible for other source files including **`foo.h`** and can be used directly by name. In order to hide the variable, we can use the **`static`** declaration which can limit the scope of **`var`** to the file **`foo.c`**.

```c
// foo.c
static int var = 2333;
// end foo.c
```

If we try to ran the program after modifying, we got the error message like this

> undefined reference to 'var'

The variable **`var`** is hidden.

- **static** for functions

A function with prefix of **static** is the same as a extern variable.

- **static** for internal variables

```c
#include <stdio.h>

void fun() {
    static int intern_var = 1;
    printf("intern_var is %d\n", intern_var++);
}

int main() {
    for (int i = 0; i < 10; ++i)
        fun();

    for (int i = 0; i < 10; ++i) {
        static int x = 1;
        printf("x is %d\n", x++);
    }

    fun();

    return 0;
}
```

The result is

```
intern_var is 1
intern_var is 2
intern_var is 3
intern_var is 4
intern_var is 5
intern_var is 6
intern_var is 7
intern_var is 8
intern_var is 9
intern_var is 10
x is 1
x is 2
x is 3
x is 4
x is 5
x is 6
x is 7
x is 8
x is 9
x is 10
intern_var is 11
```

The local variables with **static** declaration remain in existence and have a permanent storage.

Register Variables

- A heavily used variable can be declared as a register variable, which may result in smaller and faster program.
- The register declaration is just an advice and compiler may ignore it.
- It can be declared by

```
register type-name var-name;
```

- The register declaration can only be applied to a automatic variables and to the formal parameters of a function.

Block Structure

- Variables declared in inner blocks will hide any identically named variables in outer blocks and remain in existence until the matching right brace.
- An automatic variable declared and initialized in a block is initialized each time the block is entered.
- A static variable is initialized only the first time the block is entered.
- Automatic variables, including formal parameters, also hide external variables and functions of the same name.

Initialization

- without explicit initialization
  - External and static variables are guaranteed to be initialized to zero;
  - Automatic and register variables have undefined initial values.
- with explicit initialization
  - For external and static variables
    - the initializer must be constant expression;
    - the initialization is done once, conceptually before the program begins execution.
  - For automatic and register variables,
    - the initializer may be expression involving previously defined values, even functions calls.
  - An array may be initialized by following its declaration with a list of initializers enclosed in braces and separated by commas.
    - when the size is omitted, the compiler will compute the length by counting the initializer.
    - If there are fewer initializers for any array than the number specified, the missing elements will be zero for external, static and automatic variables.
    - It is error to have too many initialization.

    ```
    int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
    ```

  - Character arrays can be initialized as follow.

    ```
    char str = "Hello";
    ```

which is equivalent to

```c
char str = { 'H', 'e', 'l', 'l', 'o' };
```

Recursion

The C Preprocessor

File Inclusion

see Header Files.

Macro Substitution

What is Macro Substitution

A macro substitution is a definition having the form **`#define name replacement-text`**. Any occurrence of the token **`name`** in the source file will be replaced by the **`replacement-text`**.

Let us see an example.

```c
#include <stdio.h>

#define for(n) for(int i = 0; i < n; ++i)

int main() {
    for (3) printf("i = %d\n", i);

    return 0;
}
```

The code segment above is equivalent to

```c
#include <stdio.h>

#define for(n) for(int i = 0; i < n; ++i)

int main() {
    for (int i = 0; i < 3; ++i) printf("i = %d\n", i);

    return 0;
}
```

Ant the result is

```
i = 0
i = 1
i = 2
```

Define a Macro Substitution with Several Lines

A long definition of macro substitution can be separated into several lines by placing **** at the end of each line to be continued (be cautious with the blank).

```
#include <stdio.h>

#define HW "Hello \
world!\n"

int main() {
    printf(HW);

    return 0;
}
```

The result of this code is

```
Hello world!
```

Macro Substitution Made Only for Tokens

```
1  #include <stdio.h>
2
3  #define HW "Hello world!\n"
4
5  int main() {
6      printf("HW");
7
8      return 0;
9  }
```

The result is

```
HW
```

The string `"HW"` will not be substituted. If we want to print the string `"Hello world!"` we should modify the sixth line to

```
printf(HW)
```

and we get

```
Hello world!
```

The Scope of Macro Substitution

The scope of a name define with **`#define`** is from its point of definition to the end of the source file.

Define Macros with Arguments

We can define a macro with some arguments.

```c
#include <stdio.h>

#define MAX(A, B) (((A) > (B))? (A) : (B));

int main() {
    int p = 1, q = 2, r = 3, s = 4;
    int ans = MAX(p+q, r+s);

    printf("ans = %d\n", ans);

    return 0;
}
```

The result is

```
ans = 7
```

Any formal arguments in the macro will be replaced by the actual arguments. The code segment is equivalent to

```c
#include <stdio.h>

#define MAX(A, B) (((A) > (B))? (A) : (B));

int main() {
    int p = 1, q = 2, r = 3, s = 4;
    int ans = (((p+q) > (r+s))? (p+q) : (r+s));

    printf("ans = %d\n", ans);

    return 0;
}
```

This macro can be used for different data types.

Conditional Inclusion

> The line **#if** evaluates a constant integer expression (which may not include **sizeof**, casts, or enum constants). If the expression is non-zero, subsequent lines until an **#endif** or **#elif** or **#else** are included. (The preprocessor statement **#elif** is like else if.) The expression defined(name) in a **#if** is 1 if the name has been defined, and 0 otherwise.

```
#if !define(HDR)
#define HDR

/* some contents */

#endif
```

```
#if SYSTEM == SYSV
    #define HDR "sysv.h"
#elif SYSTEM == BSD
    #define HDR "bsd.h"
#elif SYSTEM == MSDOS
    #define HDR "msdos.h"
#else
    #define HDR "default.h"
#endif
#include HDR
```