C Programming Language

Xukun Lin

The reference is the book *C Primer Plus*.

A rather simple C code looks like this

```
#include <stdio.h>  // prepocessor part

int main(void) {
  printf("Hello world!\n");  /* print out something */
  return 0;                  // also a comment
}
```

Data Types

Data types include `int long short unsigned char float double signed void _Bool _Complex _Imaginary`. The basic types are `int float` and other types are more for variations. `int` and `long int` (or simply `long`) are 32bits, `short` is 16bits, and `long long int` is 64bits.

We can initialize and define a variable at the same time, for example,

```
long int x = 5;
```

Note that a floating point number is not that accurate. In the following case, the result could be `0`.

```
float x, y;
x = 1e20;
y = 1e20 + 1;
printf("%f", x-y);
```

Strings and I/O

To create a string of characters, we can use an array of characters,

```
char mystring[40]
```

so 40 bytes are assigned to `mystring` to store a string. For every string, there is always a null character `'\0'` to indicate the end of the string, so in this case only 39 bytes are available.

There are two ways to find the length of a string, `sizeof()` and `strlen()`. The first returns the number of memory cells used by the string, while the second returns the number of characters (including white space, null character) of the string.

In the **prepocessor** part, we can define a constant, for example π, by

```
#define PI 3.1415926536  // Capitalized constant name is a tradition
```

It can also be used to define a constant character of string. And we can also do the following instead

```
const float SQRT2 = 1.4142136;  // more flexible than using #define
```

The `printf()` function has many **conversion specifiers**. Some of them are `%c` (single character), `%d` (signed decimal), `%e` (float in e notation), `%f` (standard float), `%s` (string), etc.

There are also **modifiers**, for example, `%10d` prints a integer that is 10 characters wide, even though the int itself is smaller. Other include `%10.2f` (width=10, decimal places=2), `%-10d`, and so on.

There are 5 **flags** we can use in modifiers: `-` (left-justified), `+` (show its sign), ` ` (with a leading space), `0` (fill with 0s), and `#`.

The return value of `printf()` is the number of characters it printed.

You can also concatenate two strings in `printf()` by separating two strings with a whitespace,

```
printf("This is" " a long string.");
```

To use `scanf()`, we can write

```c
scanf("%d %f", &integer, &floating);  // notice the whitespace and &
scanf("%s", string);                  // no & only for strings (char arrays)
```

The modifiers for `scanf()` are a little different.

The return value of `scanf()` is the number of items it successfully read.

Operators, Expressions, and Statements

The common operators in C include `+ - * /`. Note that `/` works differently for `int` and `float`.

Other operators are `%` (modulus), `sizeof`, `++ --` (these two have two types, either before or after an operand, the results are different).

A `while` loop can look like this

```c
while (++index < 10) {
    printf("%d", index);
}  // a while loop
```

A **cast operator** has the form `(type)`. For example,

```c
int m;
m = (int) 1.2 + (int) 1.3
```

is valid and shows type conversion explicitly.

C Control Statements: Loops

All values except `0` are considered to be `true`.

A `for` loop in C looks like

```c
for (m = 1, n = 12; m < 10; m++) {}
```

The first expression uses the comma operator. It may be used for multiple times.

Here are more operators: `+= -= *= /= %=`. `var += 2` is equivalent to `var = var + 2`, and so on.

Different than `while` and `for` loops, which are entry-condition loops, the `do while` loop is exit-condition. It looks like

```
do {} while ();
```

Note the semicolon in the end.

An **array** is a series of values of the *same* type. The index of an array in C starts at `0`.

C Control Statements: Branching and Jumps

The functions `getchar()` and `putchar()` are the `char` type equivalence of `scanf()` and `printf()`.

The `ctype.h` header file contains useful functions for characters, including `isalpha()`, `isblank()`, and so on.

When using an `if () {} else if () {} else {}` loop, the `{}` may be ignored if only one statement is in each part. Also note that the `else` pairs with the latest `if`, unless your use `{}`. Unlike Python, indentation in C is useless.

The following loop

```
if (y < 0)
    x = -y;
else
    x = y;
```

is equivalent to the conditional expression using the `?:` conditional operator,

```
x = (y < 0) ? -y : y;
```

Logical operators are `&& || !`.

The `continue;` statement skips the rest of an iteration and starts the next iteration. So it cannot be used in `if` statements.

The `break;` statement breaks the loop and proceeds to the next stage of the program.

A `switch` statement looks like this,

```
switch () {
    case c1:
        // something
        break;
    case c2:
        // something
        break;
    default:
        // something
}
```

Note that `break` is essential. Without it all statements in `switch` that are after the chosen case are executed. `continue` does not work with `switch`.

Note that each label (`c1 c2`, etc.) must be an `int` type (including `char`). `float` does not work, but if you have to, use `if else` instead.

We can also assign multiple labels to one case,

```
switch () {
    case c1:  // this will jump to the next case since no break is present
    case C1:
        // something
        break;
    // ...
}
```

C also has a `goto` statement. We can use it like

```
if (x > 0)
    goto a;

a: x *= 12;
```

But for modern programming we should avoid using `goto`.

Character I/O and Input Validation

A **buffer** is an area of temporary storage.

A **stream** is an idealized flow of data to which the input or output is mapped.

When reaching the end of a file, the `getchar()` returns `EOF` (end of file), which in most cases is defined to be `-1` by `stdio.h`. For keyboard input, we can use `Ctrl+D` as `EOF`.

The default input is keyboard, but we can redirect to other sources, such as a text file named `infile`. Suppose we have an executable file named `file` compiled from C. In Terminal, instead of typing `./file`, we can type

```
./file < infile
```

If we want to send the output to a file named `outfile`, we can type

```
./file > outfile
```

Functions

The arguments of a function and variables defined in the function are *local*. You can use the same names in other parts of the program and there will be no conflicts.

Remember that a function should be defined outside and after `main()`.

In a function **prototyping**, we don't have to include arguments. That is, the following is perfectly fine:

```
void myfun(int, int, float);
```

The `return` will cause the function to terminate and return control to the calling function.

Use the `math.h` header file for more mathematical functions.

We can write a function to do recursive work. The following function converts a base10 integer to base2:

```c
void to_binary(unsigned long n) {
    int r;
    r = n % 2;
    if (n >= 2)
        to_binary(n / 2);
    putchar(r == 0 ? '0' : '1');

    return;
}
```

A C **header** file (`.h`) is simply a `.txt` file that contains declarations, constants, and so on.

The operator `&` gives the memory address of a variable. For example, we can write

```c
printf("%p", &var);
```

to print the address, which in most computers is displayed in hexadecimal form.

A **pointer** is a variable that has a memory address as its value. But note that it is different from the `int` type, and should be regarded as a new type.

To declare a pointer, we can write

```c
int * ptr;  // ptr is a pointer pointing to an int
```

where `int` may be replaced by other types. To define the pointer we write,

```c
ptr = &var;  // ptr is a pointer pointing to the memory address of var
```

To retrieve the value stored at `ptr`, we can use the `*` operator,

```c
value = *ptr;  // get the value at ptr
```

Pointers are extremely helpful if we want to alter variables in a calling function.

Arrays and Pointers

We can initialize an array by

```c
int myarray[3] = {1, 2, 3};  // initialize an int array
```

or simply

```c
int myarray[] = {1, 2, 3};  // no need to deal with length
```

If C99 is supported by the compiler, we can also do the following

```c
int myarray[4] = {12, [1] = 13, [2] = 14, 15};  // assign values to different places
```

To assign values to an array, we have to do it one by one.

To declare a multi-dimensional array, we can write

```c
float myarray[5][7];  // An array of 5 arrays of 7 floats
```

the declaration makes perfect sense if we read from left to right.

To initialize a multi-dimensional array, we use

```c
int myarray[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
}
```

or equivalently,

```c
int myarray[2][3] = {1, 2, 3, 4, 5, 6};
```

The two methods fill each entry differently, so trouble can arise if the initialization does not fill all entries.

The generalization to higher-dimensional array is trivial,

```c
int myarray[2][3][4][5];  // declare a 4-dimensional array
```

An array name is the memory address of the first element of the array. That is, the following is true,

```
myarray == &myarray[0];
```

In C, arrays and pointers are closely related and notations can be converted both ways.

When you add `1` to a pointer, the pointer actually adds one **storage unit**. If the pointer points to a `double`, which is 64bits or 8bytes, adding `1` causes the pointer to point to a new address that is `8` larger (in most cases the address is in unit of bytes). So the following is true,

```
myarray[3] == *(myarray + 3);
```

Operations on pointers include `+ -` integers, subtracting one pointer from another, and comparing two pointers.

Note that if calling a function that takes an array (which is a pointer) as argument, the function may change the original values of the array, and we need to be careful. To avoid errors, we can write

```
int sum(const int myarray[], int n) {}  // add const to avoid accidental changing
```

To declare a pointer to an array, which is helpful for dealing with higher-dimensional arrays, we can use

```
int (* ptr)[2];  // a pointer to a 2-int array
```

For a function, we can write

```
int myfun(int (*ptr)[2]) {}  // function definition
```

or equivalently,

```
int myfun(int ptr[][2]) {}  // equivalent function definition
```

A **compound literal** can be used to represent an array. For example, we can write

```c
int * ptr;                // declare a pointer
ptr = (int [2]) {1, 2};   // initialize the pointer
```

Character Strings and String Functions

A **string** is a character array with the null character (`\0`) at the end.

The function `puts()` displays a string and automatically adds a new line.

Unlike a pointer, if you use array, the array name is constant. For example

```c
char mystring[] = "hello";  // mystring is automatically a constant
char * ptr = "hi";          // nothing is constant
```

Note that in the first case the name `mystring` is not a variable, but each element, such as `mystring[2]` , can still be altered. For the second case, since the pointer points to a string literal, it would be highly recommended to use

```c
const * ptr = "hi";  // add const when pointing to a constant array
```

The functions `fgets()` and `fputs()` can be used to read input. For example,

```c
char word[12];
fgets(word, 12, stdin);   // read a string of length 11 from keyboard (stdin) to
word
fputs(word, stdout);      // display word to screen (stdout)
```

There are several useful string functions. They are in `string.h` .

`strlen()` finds the length of a string.

`strcat()` takes two string arguments, a copy of the second string is concatenated to the first one to create the new first string, and the second string is not altered. Not that the first string needs to have enough length for the new string, otherwise `strcat()` can cause problems.

`strncat()` takes a third argument to know the max number of characters to add.

`strcmp()` compares two strings. Two strings can have different memory sizes, but `strcmp()` only cares about characters before the null character. The return value is `0` if two strings are same and nonzero otherwise.

`strncmp()` compares strings up to a max length, so it takes a third argument.

`strcpy()` and `strncpy()` copy strings. For example,

```
strcpy(myarray, ptr);  // copy string pointed by ptr to array pointed by
myarray
```

`sprintf()` is just like `printf()`, except that it has an extra first argument which is a pointer to a string (so it is used to write to a string instead of a display).

```
sprintf(mystring, "%d", 125);  // write 125 to the string mystring
```

The `stdlib.h` file contains conversion functions.

`atoi()` is from a string to an `int`. For example, `"210"` to `210`. There are also `atof()` (to `double`) and `atol` (to `long`). Others include `strtol()`, `strtoul()`, `strtod()`, etc.

```
char mystring = "550 this";
char * ptr;
strtol(mystring, &ptr, 10);  // base 10, ptr would point to anything after 550
```

Note that we only provide the memory address of the pointer `ptr`, and the value that the pointer points to, which would be stored in the address of the pointer, is set by `strtol()`.

Storage Classes, Linkage, and Memory Management

Scope describes the regions of a program that can access an identifier (a variable name).

- Variables defined inside a block (`{}`) has **block scope**, which means they are only visible inside the block. Note that the `()` part of a function or a loop is also considered to be part of the block.
- **Function scope** only applies to labels used with `goto` statements. As long as a label is defined inside a function (anywhere, does not matter in block or not), it is visible to the whole function.
- **Function prototype scope** applies to function prototypes. The variables are only visible within the prototype declaration.

- A variable that is defined outside of any function (including `main`) has **file scope**. It is visible to the whole file. File scope variables are also called **global variables**.

A file scope variable is actually visible to the whole **translation unit**. When a header file, for example, is included in a source code file, the content inside the `.h` file is copied to the `.c` file by the compiler, and the new single file is called a translation unit.

Variables with the first three scopes have **no linkage**, which means they are private to the block, function, or function prototype. Variables with file scope have either **internal** or **external linkage**. A variable with internal linkage can be accessed within a translation unit, and with external linkage it can be accessed anywhere in a multi-file program. The word `static` is used for internal linkage.

```
int x = 5;         // file scope, external linkage
static int y = 6;  // file scope, internal linkage
```

There are 4 types of storage duration,

- An object that has **static storage duration** exists throughout program execution. All file scope variables have static storage duration, and the previous `static` refers to the linkage type, not the duration type.

- An object that has **thread storage duration** exists inside a thread.

- Variables with block scope normally have **automatic storage duration**. They exist inside their blocks and the memory is freed after the block exits.

- There is also **allocated storage duration**.

We can also add `static` to a block scope variable to let it have static storage duration. This kind of variable only initializes once, no matter how many times the function that contains it is called.

There are 5 storage classes,

Table 12.1 **Five Storage Classes**

Storage Class	Duration	Scope	Linkage	How Declared
automatic	Automatic	Block	None	In a block
register	Automatic	Block	None	In a block with the keyword `register`
static with external linkage	Static	File	External	Outside of all functions
static with internal linkage	Static	File	Internal	Outside of all functions with the keyword `static`
static with no linkage	Static	Block	None	In a block with the keyword `static`

We use `auto` for automatic class (default for variables inside blocks). But it is recommended not to use it unless necessary. Note that if two variables, one in the outer block and one in the inner block, have the same name, then the inner variable *hides* the outer variable. But when the inner block exits, the outer variable comes back to scope.

We use `register` for register class. This kind of variable is stored in CPU registers, and hence the address operator `&` cannot be used.

We use `extern` for static with external linkage class. Note that if an external variable is defined in one `.c` file, then using `extern` is mandatory to use that variable in the current `.c` file. Note that a variable with block scope hides the variable with file scope that has the same name.

```
int x = 50;  // external variable
int main(void) {
    extern int x;  // if only without extern, x here would be a new local
variable
    // something
}
```

In the code above, the first is called a **defining declaration**, and the second is called a **referencing declaration**.

When defining file scope variables, we can only use constants. That is, the following is invalid,

```
int x = 2;      // file scope
int y = 2 * x;  // file scope, cannot use x, which is a variable
```

To declare a static storage variable with internal linkage, we use

```
static int x = 1;  // file scope, static storage, internal linkage
```

Function prototypes are internal by default. To make it external so that other source files can use the function, we use `extern`.

```
double myfun(double);       // external by default, a good practice is to add
extern
static int mynextfun(int);  // internal function
```

The function `malloc()` can be used to allocate memories in bytes. The return value of the function is a void typed pointer pointing to the address of the first byte of the memory block, but we can use type conversion to convert it to other types.

```
double * ptd;
ptd = (double *) malloc(sizeof(double));  // convert void pointer to double
pointer
```

This method can be used to create arrays whose length is defined during running. (remember previously an array length must be a constant, but here it can be a variable).

The `free()` function is used to free the memory block that has been allocated. It only works with memory allocated by `malloc()`.

```
free(ptd);  // free the memory block allocated in the last code block
```

Another function for memory allocation is `calloc()`, whose first argument is the number of cells to allocate, and the second argument is the length of each cell in bytes. The `free` function can also be used.

```
long * ptd;
ptd = (long *) calloc(10, sizeof(long));  // an alternative for malloc
```

There are 3 type qualifiers, `const`, `volatile`, and `restrict`.

`const` means the program cannot change the value of a variable.

`volative` means that the variable can be altered by agencies other than the program.

`restrict` is applied only to pointers, and it indicates that a pointer is the sole initial means of accessing a data object.

File I/O

A **file** is a named section of storage.

In C, there are 2 ways to see a file, **binary mode** and **text mode**.

`exit()` causes a program to terminate. The argument of it indicates how the program terminates.

```
exit(EXIT_FAILURE);  // program terminates abnormally (EXIT_FAILURE is in
stdlib.h)
```

`fopen()` opens a file. To use it, we can type

```
fopen("name", "r");  // open the file with name "name" and read it only
```

The first argument is the file name. The second argument is a string identifying the mode, including `"r"` (reading), `"w"` (writing, clean the file if it exists), `"a"` (writing, but append to its end if the file exits), etc.

The return value of `fopen()` is a **file pointer**, which is defined in `stdio.h`. The pointer points to a data object containing information about the file, not the file itself. To declare such a pointer, simply type

```
FILE * ptr;  // a file pointer
```

The functions `getc()` and `putc()` are like `getchar()` and `putchar()`, but the difference is the new ones work with files. For example,

```
mychar = getc(fin);  // get a character from the file identified by pointer fin
putc(mychar, fout);  // put mychar to the file identified by fout
```

`fclose()` takes a file pointer as an argument and closes a file. It returns `0` if successful and `EOF` it not.

`fprintf()` and `fscanf()` work like `printf()` and `scanf()`. But the new ones take a file pointer as the first argument.

`fseek()` and `ftell()` work with contents in a file. `fseek()` is used to set the location where we are at, and `ftell()` returns the the number of bytes from the beginning to the current location of the file. For example,

```
FILE * ptr;
long current;
ptr = fopen("myfile", "rb");  // read file in binary mode
fseek(ptr, -1L, SEEK_END);    // start from end of file and offset is -1
current = ftell(ptr);         // assign number of bytes to current
```

The 1st argument of `fseek()` is the file pointer, the 2nd argument is the offset with respect to the starting point (must be type `long`), and the 3rd argument defines the starting point, including `SEEK_SET` (beginning), `SEEK_CUR` (current), and `SEEK_END` (end).

There are other functions, such as `fgetpos()` and `fsetpos()`, that can be useful.

Other I/O functions mentioned in the book are omitted here.

Structures and Other Data Forms

We can declare a structure by

```
struct book {
    char title[20];   // title of a book
    char author[20];  // author of a book
    float value;      // price of a book
}
```

To create a structure variable, we write

```
struct book mybook;  // mybook is a variable of book structure
```

Note that `struct book` acts like a new type, just like `int` or `float`.

To initialize a structure variable, we can write

```
struct book mybook = {
    "The Title",
    "the author",
    0.25
}
```

To access an element of the structure, for example `title`, by

```
mybook.title;  // this is the title, a string
```

To initialize a structure in a different way, we use

```
struct book mynextbook = {
    .title = "Next Title",
    .author = "next author",
    .value = 0.2  // we don't have to initialize all 3 elements
}
```

To declare an array of structures and get access to elements, we use

```
struct book library[100];  // an array of book structures
// some definitions of the elements
library[14].title;  // title of the 15th book
```

We can also create nested structures,

```
struct bookcase {
    struct book abook;   // suppose struct book is defined
    int index;           // index of the bookcase
}
struct bookcase mycase;  // declare a bookcase structure
// some definitions
```

To initialize a structure contained in another structure, we can use `{entry, entry, ...}`. To access an individual element, we simply use

```
mycase.abook.value;  // value
```

To declare a pointer to a structure, we use

```c
struct book * ptr;
* ptr = &mybook;  // initialize the pointer
```

Note that unlike arrays, the name of a structure variable is not the memory address of it. With the `->` operator for pointer, the following are equivalent,

```c
mybook.title;  // title of mybook
ptr->title;    // equivalent
```

We can use a structure, a pointer to a structure, or an element in a structure as a function argument.

We can assign one structure to another structure of the same type, even when the elements include arrays. (Note that we cannot assign one array to another array).

We can use a compound literal to create a temporary structure,

```c
(struct book) {"Title", "Author", 20};  // not assigned to any variable
```

There is a *flexible array number* feature in C99, but it is omitted here.

Other parts will be added later.