**C**

**C requires that each statement end with a semicolon ;**

**-Deepansh Sabharwal**

Even the simplest C programs rely on three key language features: **directives**

(editing commands that modify the program prior to compilation), **functions** (named

blocks of executable code, of which main is an example), and **statements** (commands to

be performed when the program is run).

Directives (always begin with a # character)

Before a C program can be compiled, it must be edited by a preprocessor.

Commands intended for the preprocessor are called directives.

To compile a C program, the first line which should be present is

**#include <stdio.h>**

This is done because C unlike other languages has no built in Read or Write commands

This provides input and output features without using any functions.

Functions:

Functions in C is a bit different than what we see in other programming languages.

Functions fall into two categories: **those written by the programmer** and those **provided a part of the C implementation**.

In C, functions are a series of statements grouped together and given a name.

There are a lot of functions in C, but **main** is the most important out of all.

If main is a function, **does it return a value**? **Yes**: it returns a **status code** that is given to the operating system when the program terminates.

**[See code for example at hello.c]**

In the code example we used int main(void) {here goes the statements}

The word **int** just before main indicates that the **main function returns an integer value**.

The word **void** in parentheses indicates that **main has no arguments**.

Now the last statement inside main is return 0;

**This has 2 effects. One is that it’ll cause the function to terminate and the second is that it indicates that the main function returns a value of 0.**

If no return statement, the function will still terminate but many compilers give a warning message.

Some features

**printf** is used to print in C

If you want to print multiple statements inside the main function but you wrote return 0 after the first one, the function terminated, and the second one won’t be printed

**When printing something use “\n” always.**

Comments

Single line comments: //

Multi line comments: /\* …… \*/

We can use cmd + / to single line comment on MacOS

**printf("My "); */\** forgot to close this comment**

**printf("cat ");**

**printf("has "); /\* so it ends here \*/ printf("fleas");**

**This prints ‘My fleas’**

**printf("%d\n", i, j); */ \* \* \** WRONG *\* \* \* /***

In this case, printf prints the value of i but doesn’t show the value of j.

**We can also use methods for printing instead of scanf**

**Like for string which we will study later**

**string answer = get\_string(“What is your name”);**

**printf(“Hello %s\n”, answer);**

**Note: %s is for strings like we did for float and integers.**

Because we’ve neglected to terminate the first comment, the compiler ignores the middle two

statements, and the example prints **My fleas.**

Variables and Assignments:

A variable of type float (short for *floating-point)* can store much larger numbers than an

Int variable. Furthermore, a float variable can store numbers with digits after the decimal

point, like 379.125.

Float variables have drawbacks. However, arithmetic on float numbers may be slower

than arithmetic on int numbers.

We declare the same way as we do in JAVA or various other high-level languages

int height; or float marks;

For assignment:

Correct: int height; height = 8;

Wrong: height = 9; int height;

Also, for float like in JAVA: **profit = 2150.48f;**

The order matters.

The numbers 8 and 9 are known as constants

Now if we create a simple program for calculating the volume of a cylinder, we

can use various methods, but a simple one would be to identify each variable and then print it.

**[See code for example at hello.c]**

Now if we want to print the volume, we must use placeholders ‘%’

For float we use %f and for integers we use %d

**So, the print statement is: printf(“The volume is: %f or %d” , volume);**

A simple yet important program to understand fundamentals:

Fahrenheit to Celsius converter

**[See code for example at ftoc.c]**

Important concepts to look from these are how to take user input, define certain variables

which have a fixed value and how to print the result up to 1 precision.

How to take user input in C

scanf("format\_string", &variable) ;

Now in place of "format\_string", we’ll use %d or %f.

**[See code for example at hello.c]**

I can define default values like Freezing point and Scale factor as show in the code

Just use #define Variable\_name data\_type (no ‘;’ needed)

Escape Sequence

\n is the new line character

\t is tab

\b is backspace

Scanf in detail

**int i, j ; float x, y;**

**scanf("%dld%f%f", &i, &j, &x, &y);**

Suppose that the user enters the following input line:

**1 -20 .3 -4.0e3**

scanf will read the line, converting its characters to the numbers they represent, and then assign 1, -20, 0.3, and ^000.0 to i, j, x, and y, respectively.

“Tightly packed" format strings like "%d%d%f%f" are common in scanf calls, printf format strings are less likely to have adjacent conversion specifications.

YOU CANNOT WRITE TEXT INSIDE THE SCANF BLOCK, JUST FOLLOW THE SYNTAX.

**Many professional C programmers avoid scanf, instead reading all data in character form and converting it to numeric form later. We’ll use scanf quite a bit, especially in the early chapters of this book, because it provides a simple way to read numbers.**

**If:**

The first character is 1 and second is – and we have %d, it will put the – character back and will store only 1

The first character is -2, 0 and . and we have %d, it will put the . character back and will store only -20 as integer cannot have a decimal point in it.

The first character is ., 3 and third is – and we have %f, it will put the – character back and will store only 0.3 as a negative sign cannot be placed after the number.

**[See code for example at Adding2fractions.c]**

Expressions

Arithmetic Operators

The unary + operator does nothing; in fact, it didn’t even exist in K&R C. It’s used primarily to emphasize that a numeric constant is positive.

The / and % operators require special care:

**The / operator can produce surprising results. When both of its operands are integers, the / operator “truncates” the result by dropping the fractional part. Thus, the value of 1 / 2 is 0, not 0.5.**

**The % operator requires integer operands; if either operand is not an integer, the program won’t compile.**

**In C99, on the other hand, the result of a division is always truncated toward zero (so - 9 / 7 has the value -l)and the value of i % j has the same sign as i (hence the value of -9 % 7 is -2).**

Assignment Operator

Since assignment is an operator, several assignments can be chained together: i = j = k = 0;

The = operator is right associative, so this assignment is equivalent to i = (j = (k = 0) ) ;

The effect is to assign 0 first to k, then to j, and finally to i.

**i = i+2 is the same as i +=2**

The compound assignment operators have the same properties as the =

operator. They’re right associative, so the statement

**i += j += k;**

**means**

**i += (j += k) ;**

Increment and Decrement

When ++ or - - is used more than once in the same expression, the result can often be hard tounderstand. Consider the following statements:

**i = 1; j = 2; k =** **++i + j++;**

What are the values of i, j, and k after these statements are executed? **Since i is incremented *before* its value is used**, but **j is incremented *after* it is used**, the last statement is equivalent to

**i = i + 1 k = i + j**

**j** **= j + 1**

So, the final values of i, j,and k are 2, 3, and 4, respectively. In contrast, executing the statements

**i = 1; j = 2; k =**

**i++ + j++;**

will give i, j, and k the values 2, 3, and 3, respectively.

Precedence table for reference:

A table of symbols and maths

AI-generated content may be incorrect.

Important Example

**a = 5; b = a + 2; a = 1; c = b - a;**

The value of c will always be 6 after these statements are executed.

Selection Statements

Logical Expressions

Relational operators

>, <, <=, >=

Equality operators

== for equal

!= for not equal

Logical operators

j: logical NOT

&&: logical AND

||: logical OR

The if statement:

if (expression) statement

Ex: **if (line\_num == MAX LINES) line\_num = 0;**

Compound Statement

**if (line\_num == MAX\_LINES) { line\_num = 0;**

**page\_num++;**}

The else clause:

Example

**if (i > j) max = i; else max = j;**

Add braces {} always, makes it more readable

Cascaded if else statement (simple example)

**if (n < 0)**

**printf("n is less than 0\n"); else**

**if (n == 0)**

**printf("n is equal to 0\n"); else**

**printf("n is greater than 0\n");**

Loops

The while statement

**i = 1;**

**while (i < n)**

**i = i \* 2 ;**

The do statement

**i = 10;**

**do {**

**printf("T minus %d and counting\n", i);**

**--i;**

**}**

**while (i > 0);**

Essentially just a while statement whose controlling expression is tested *after*

each execution of the loop body.

The ‘for’ statement

**for (i = 10; i > 0; i--)**

**printf("T minus %d and counting\n", i);**

Important terms

Break: we use break; when we want to exit the loop at a given situation

Continue: Passes on to the next statement, doesn’t terminate the program

**[See code for example at primenumbers.c]**

Functions

Syntax

***return-type function-name ( parameters )***

***{***

***declarations***

***statements***

***}***

Example

**double average(double a, double b)**

{

**return (a + b) / 2;**

}

The word double at the beginning is average’s *return type:* the type of data that the function returns each time it’s called

In this example, both a and b have type double. (It may look odd. but the word double must appear twice, once for a and once for b.) A function parameter is essentially a variable whose initial value will be supplied later, when the function is called.

Not all functions return a value

We use void for that as shown in the example below

**/\* Prints a countdown \*/**

**#include <stdio.h> void print\_count(int n)**

{

**printf("T minus %d and counting\n", n);**

}

**int main(void)**

{

**int i ;**

**for (i = 10; i > 0; --i) print\_count(i);**

**return 0;**

}

Initially, i has the value 10. When print\_count is called for the first time, i is copied into n, so that n takes on the value 10 as well. As a result, the first call of print\_count will print **T minus 10 and counting.**

Functions may not return arrays, but there are no other restrictions on the return type.

Specifying that the return type is void indicates that the function doesn‟t return a

value.

Return and Exit

The exit function is another way to terminate a C program besides using return in main. It is included in the <stdlib.h> header. The argument passed to exit() serves the same purpose as the return value from main: both indicate the program’s termination status.

To indicate **normal termination, use exit(0);**, though 0 can be unclear. **Instead, C provides EXIT\_SUCCESS**, which is equivalent but more readable: exit(EXIT\_SUCCESS);. **For abnormal termination, use EXIT\_FAILURE**, like exit(EXIT\_FAILURE);.

EXIT\_SUCCESS and EXIT\_FAILURE are macros defined in <stdlib.h>. Their exact values are implementation-defined but commonly EXIT\_SUCCESS is 0 and EXIT\_FAILURE is 1.

Calling return from main is equivalent to calling exit(expression);. However, there is a key difference: return only terminates the program when used in main, while exit() ends the program no matter which function calls it.

Some programmers prefer to use exit() throughout their code to make all termination points more visible and consistent.

**Local vs Global Variables in C: [See code for better understanding at scope\_global\_local.c]**

**Global variables** are declared outside all functions and accessible throughout the file.

**Local variables** are declared inside functions and exist only within that function's scope.

If both exist with the same name, the **local variable overrides** the global one inside that function.

Pointers

In most modem computers, main memory is divided into *bytes,* with each byte capable of storing eight bits of information

If n bytes, then we can address from 0 to n-1.

Pointer variables can appear in declarations along with other variables:

**int i, j, a [10] , h[20], \*p, \*q;**

In this example, i and j are ordinary integer variables, a and b are arrays of integers, and p and q are pointers to integer objects.

**int \*p; /\* points only to integers \*/**

**double \*q; /\* points only to doubles \*/ char \*r;**

int \*p; — declares p as a pointer to int

p = &i; means pointer p stores the address of integer i.

\*p lets you access or modify the value of i through the pointer.

The Indirection Operator

Once a pointer variable points to an object, we can use the \* (indirection) operator to access what’s stored in the object. If p points to i, for example, we can print the value of i as follows:

**printf("%d\n", \*p);**

printf will display the *value* of i, *not* the *address*

If we change the value of p, the value of i changed as well

Diagram to help visualise the process which is happening

A table with text and images

AI-generated content may be incorrect.

Never apply the indirection operator to an uninitialized pointer variable. If a pointer variable p hasn’t been initialized, attempting to use

the value of p in any way causes undefined behavior. In the following example, the call of printf may print garbage, cause the program to crash, or have some other effect:

**int \*p;**

**printf("%d", \*p); /\*\*\* WRONG \*\*\*/**

Pointer Assignment

Suppose we have created a pointer \*p = &i, which stores the address of i.

Now we wrote q = p, this will copy the contents of p, which is the address of i in this case. This implies that the value of i can be changed by changing the value of p or q **(Be careful to not confuse q=p as \*q = \*p)**

Any number of pointer variables may point to the same object.

If I create \*p = &i; \*q = &j;

The assignment \*q = \*p copies the value that p points to (the value of i) into the object that q points to (the variable j).

Pointers as arguments

A function max\_min(x, y, z, \*max, \*min) takes 3 numbers and returns max/min via pointers.

We pass the addresses of variables using & to allow the function to modify them.

Inside the function, \*max and \*min is used to store the highest and lowest values.

Comparison is done using simple if conditions.

This avoids arrays and still lets the function return multiple values.

**[See code min\_max\_pointers.c](my work) or [See code pointers.c](GPT)**

\*If you want to use a defined function inside main: Let’s, take the example of swapping 2 numbers using pointer **[See code swap\_2\_variables.c]**. In that we made a function and then called it inside main using functionname(&x,&y). Here x and y are the parameters we passed for the argument in the function. It works!

Pointer and Arrays

Making a pointer p point to an element of an array a isn’t particularly exciting.

However, by performing *pointer arithmetic* (or *address arithmetic)* on p, we can access the other elements of a. C supports three (and only three) forms of pointer arithmetic:

Adding an integer to a pointer

Subtracting an integer from a pointer

Subtracting one pointer from another

Subtracting one pointer from another

P = &a[5];

Q = &a[1];

i = p - q; /\* i is 4 \*/ but i = q – p; /\* i is now -4 \*/

Comparing Pointers

**p = &a [5] ; q = &a[1] ;**

the value of p <= q is 0 and the value of p >= q is 1.

Pointers to Compound literals

Consider the following example:

**int \*p = (int []){3, 0, 3, 4, l};**

p points to the first element of a five-element array containing the integers 3, 0, 3, 4, and 1.

Using a compound literal saves us the trouble of first declaring an array variable and then making p point to the first element of that array:

**int a [] = {3, 0, 3, 4, l}; int** **\*p = &a[0];**

Using Pointers for Array processing

p initially points to a [0]. Each lime through the loop, p is incremented: as a result, it points to a [1], then a [2], and so forth. The loop terminates when p steps past the last element of a.

**#define N 10**

**int a[N], sum, \*p;**

**sum = 0 ;**

**for (p = &a[0] ; p < &a[N]; p++) sum** **+= \*p;**

Strings

We use **%s** for strings as we used for integers and float.

We have discussed Escape Sequences previously.

Continuing a string literal

**printf("When you come to a fork in the road, take it. \ --Yogi Berra");**

In general, the \ character can be used to join two or more lines of a program into a single line (a process that the C standard refers to as “splicing”).

The \ technique has one drawback: the string must continue at the beginning of the next line, thereby wrecking the program’s indented structure.

There’s a better way to deal with long string literals, thanks to the following rule: when two or more string literals are adjacent (separated only by white space), the compiler will join them into a single string. This rule allows us to split a string literal over two or more lines:

**printf("When you come to a fork in the road, take it. " "--Yogi Berra");**

In essence, C treats string literals as character arrays. When a C compiler encounters a **string literal of length *n*** in a program, it sets aside *n +* 1 bytes of memory for the string. This area of memory will contain the characters in the string, **plus one extra character—the *null character*—to mark the end of the string.** The **null character is a byte whose bits are all zero**, so it’s **represented by the \0** escape sequence.

A string variable can be initialized at the same time as it is declared

char date[] = “June 14” OR char \*date = “June 14”

**char \*p;**

**p[0] = 'a'; /\*\*\* WRONG \*\*\*/**

**p[l] = 'b'; /\*\*\* WRONG \*\*\*/**

**p[2] = 'C'; /\*\*\* WRONG \*\*\*/**

**p[3] = '\0'; /\*\*\* WRONG \*\*\*/**

Since p hasn’t been initialized, we don’t know where is it pointing. Using the pointer to write the characters a, b, c, and \0 into memory causes undefined behavior.

### **fgets vs scanf (string input in C):**

**fgets()** reads input until a **newline (\n)** or until the **buffer is full**, **including spaces**.

**scanf("%s", ...)** stops at the **first whitespace** (space, tab, newline).

**fgets() keeps** the **newline character** (\n) in the string (unless it's too long).

**scanf() stops** before any whitespace — only stores the **first word**.

Example

To C, or not to C: that is the question.

**scanf("%s", sentence);** → stores: "To"  
**fgets(sentence, sizeof(sentence), stdin);** → stores: "To C, or not to C: that is the question.\n"

### Accessing the Characters in a String — Using Indexing s[i]

In C, strings are arrays of characters terminated by a null character '\0'.

You can access each character in a string using array indexing notation: s[i], where i is the index (starting from 0).

For example, s[0] is the first character, s[1] the second, and so on.

The last character before the null terminator is at index strlen(s) - 1.

**Important:**

Always ensure i is within bounds to avoid accessing invalid memory.

The string ends when '\0' is found, so you can loop until you hit that.

### Using the C String Library

C provides a standard library <string.h> with many useful functions to manipulate strings safely and efficiently.

Here are some common ones:

#### 1. strcpy(dest, src) — Copy a string

Copies the string from src to dest.

The destination must have enough space to hold the source string plus the null terminator.

The contents of dest are replaced.

#### 2. strlen(s) — Get the length of a string

Returns the number of characters in the string s, excluding the null terminator.

Useful for determining the string size.

#### 3. strcat(dest, src) — Concatenate strings

Appends the string src to the end of string dest.

dest must have enough space to hold the result.

The null terminator of dest is overwritten, and a new null terminator is added.

#### 4. strcmp(s1, s2) — Compare two strings

Compares the strings s1 and s2 lexicographically (dictionary order).

Returns:

0 if s1 and s2 are equal,

A negative value if s1 < s2,

A positive value if s1 > s2.