C Primer

Karl Hallsby

Last Edited: October 4, 2020

Contents

1 Introduction

This document is intended for programmers that are newer to the C language and its facilities. This is meant as a quick, supplementary, reference document for these programmers. Many of the code examples in this document are either from IIT's CS 351 course, or Kernighan and Ritchie's Kernighan and Ritchie 1978 manual, *The C Programming Language*.

1.1 Properties

C is referred to as "low-level" today. That means there are relatively few abstractions and few "syntactic sugars" for expressing computations. This means when you write C, you can typically guess what the assembly would look like, which also lends itself to C's execution speed. However, this also means that you have **VERY FEW** built-in language protections for your computations. So, you open yourself up to a whole new class of problems when developing and writing your programs. The language will not protect you, but C compilers will typically attempt to throw warnings or errors about the most egregious errors you may write.

Here are some properties of C that you should know about.

1.1.1 Imperative

C is an imperative language, meaning you express computation as a series of steps. This is based off a finite-state based understanding of computation.

This stands in stark contrast to functional languages, which express computation as function applications to expressions.

1.1.2 Procedural

A procedural language allows you to organize repeated computations into logical blocks, typically referred to as functions or procedures.

1.1.3 Lexically Scoped

C is lexically scoped because variables inside of procedures cannot exist outside of their logical blocks.

This stands in contrast to Dynamically Scoped languages, where variables are sometimes available outside of the written scope for that variable. Bash is an example of a dynamically scoped language.

1.1.4 Statically Typed

C is statically typed, because the types of **ALL** expressions **MUST** be specified **BEFORE compilation**. This also means that when something is declared to be a certain type, it stays that way, unlike Python. This means that type checking happens during compilation, ensuring that all expressions have well-formed types.

To ensure program flexibility, we also introduce type-casting, where we either widen the type or narrow it. For example, taking an `int` and turning it into a `long` is a widening type cast, which are usually safe. This means that sometimes we must deal with type polymorphism, although C's handling of this class of problems is iminal and quite basic.

1.1.5 Weakly Type Checked

C is technically strongly typed on its operations, however, it becomes weakly typed because compilers cannot always ensure well-formedness of expressions when Pointers are used. Because C is weakly typed, you are not always guaranteed that when you access data that you are interpreting the bytes the right way.

C becomes weakly-typed because you can typecast pointers, pull values out of unions with different types, and in general do weird things with anything in memory. This also means that the compiler might not throw type-checking errors during the compilation phase of program development. Some of these pointer issues will only arise after running the program and ensuring that it is tested properly.

Some of these issues are illustrated in Listing 1.

2 Syntax

The C language helped define a whole class of syntax that is used by many programming languages today. C's syntactic decisions can be seen in Java, Rust, C++, C#, and many others.

```c
#include <stdio.h>

int main(void) {
/* Types are implicitly converted */
        char c = 0x41424344;
        int i = 1.5;
        unsigned int u = -1;
        float f = 10;
        double d = 2.5F; // Note: 'F' suffix for floating point literals

        printf("c = '%c', i = %d, u = %u, f = %f, d = %f\n", c, i, u, f, d);

/* Typecasts can be used to force conversions */
        int r1 = f/d;
        int r2 = f / (int) d;

        printf("r1 = %d, r2 = %d\n", r1, r2);
        return 0;
}
```

```
$ ./a.out
c = 'D', i = 1, u = 4294967295, f =10.00000
r1 = 4, r2 = 5
```

Listing 1: Illustration of C's Weak Type Checking

2.1 Primitive Types

C has just 4 primitive types:

`char` : **One byte** integers (0–255), meant to represent ASCII characters.
`int` : Integers, which is defined to be *at least* 16 bits.

- Additional prefixes can be used to increase or decrease the range of the integer.
- These are shown in Section 2.1.1.

`float` : Single precision IEEE floating point number.
`double` : Double precision IEEE floating point number.

2.1.1 Integer Type Prefixes

`signed` : The default for integers, meaning you **do not** have to specify this. Can represent both negative and positive integers. The range for this is $-2^{\#\text{ bits}-1}$–$2^{\#\text{ bits}-1} - 1$
`unsigned` : , Can only represent 0 and positive integers. Its range is 0–$2^{\#\text{ bits}}$
`short` : Tells the compiler that the integer must be at least 16 bits.
`long` : Tells the compiler that the integer must be at least 32 bits.
`long long` : Tells the compiler that the integer must be at least 64 bits.

2.2 Basic Operators

Operators perform some operation on expressions. This could be an arithmetic, a relational, logical, etc. operation.

2.2.1 Arithmetic Operators

These operators are for performing mathematical operations. These are well-defined for integers and floating-point numbers.

`+` The addition operator. Works similarly for integers and floating-point numbers.

- The subtraction operator. Works similarly for integers and floating-point numbers.

* The multiplication operator. Works similarly for integers and floating-point numbers.

/ The division operator. This returns the quotient of a division. This has a different result for integers and floating-point numbers.

  - Integers return the quotient of the division, but no fractional part.
  - Floating-points return the entire remainder of the division.

% The modulo operator. Returns the remainder of a division operation when dividing integers. Note that this is only defined for integers.

2.2.2 Logical

Logical operators work with boolean values.

! The logical NOT operator.

&& The logical AND operator. Returns `1` if and only if the left and right expressions are **BOTH** `1` at the same time. Otherwise, `0` is returned.

|| The logical OR operator. Returns `0` if and only if both the left and right expressions are `0` at the same time. Otherwise, `1` is returned.

2.2.3 Relational

These relational operators are used to define a value in relation to another. Typically, these are used for boolean comparisons.

== The equality operator. Returns `1` if and only if the two expressions are equal, `0` otherwise.

!= The inequality operator. Returns `0` if and only if the two expressions are not equal, `1` otherwise.

> The greater-than operator. Returns `1` if and only if the left expression has a greater value than the right one.

>= The greater-than-or-equal-to operator. Returns `1` is and only if the left expression has a greater value or equal value than the right one.

< The less-than operator. Returns `1` if and only if the left expression has a lesser value than the right one.

<= The less-than-or-equal-to operator. Returns `1` is and only if the left expression has a lesser value or equal value than the right one.

2.2.4 Assignment

Unlike many other languages, in C, the assignment operator `=` is also an expression. This means that when an assignment is performed, it also returns a value, in this case, it returns the value that was assigned to that particular name.

= The assignment operator. Assigns a value to a given name. Returns the value of the assignment.

+= The add-and-assign operator. Takes the name on the left, adds the value on the right to the value on the left, and stores the result in the value on the left.

*= The multiply-and-assign operator. Takes the name on the left, multiplies the value on the left by to the value on the right, and stores the result in the value on the left.

Only `+=` and `*=` are shown, but there are similar ones defined for other Arithmetic Operators too.

2.2.5 Conditional Operator

There is only one conditional operator, sometimes called the ternary operation or conditional expression. It is defined like so: `bool ? true_exp : false_exp` .

2.2.6 Bitwise Operators

Bitwise operators work on the component bits of a number. This means they behave slightly differently than any other operator, but are not typically used in day-to-day calculations. Usually, they are used to efficiently work with memory.

& Bitwise AND

| Bitwise OR

^ Bitwise exclusive OR (XOR)

`~` Bitwise negation, one's complement.

`>>` Bitwise SHIFT right

`<<` Bitwise SHIFT left

2.3 Boolean Expressions

Because C is so low-level, the concept of `true` and `false` are defined as integers.

`0` `false`.

`1` `true`. Technically, any non-zero value is considered `true`.

```c
#include <stdio.h>

int main(void) {
        printf("%d\n", !(0));          // 1
        printf("%d\n", 0 || 2);        // 1
        printf("%d\n", 3 && 0 && 6);   // 0
        printf("%d\n", !(1234));       // 0
        printf("%d\n", !!(-1020));     // 1

        return 0;
}
```

```
$ ./a.out
1
1
0
0
1
```

Listing 2: Logical Operators

2.4 Control Flow

Control flow is the idea of changing the direction a program executes based on some predicate. Whether this change in direction is to a new direction is because of a change in state, or because something must be repeated is irrelevant.

2.4.1 Branching

Branching has to deal with the changing of a program's control flow based on a predicate to perform some separate actions based on the state of the predicate. There are 4 types for this:

`if` The most basic change in flow control. If the predicate provided is `true`-thy, performs an action, then returns to normal.

`if else` If the predicate is `true`-thy, then perform some action, if the predicate is `false`, then perform some other action.

`if else if else` If the predicate in the first `if` is `true`-thy, then that branch is taken. If the first predicate is `false`, then the `else if`'s predicate is checked for truth. The the `else if`'s predicate is `false`, then execution continues through any other `else if`s that may be present. If none of the `if` or `else if`s' predicates were `true`-thy, then the `else` is taken.

`switch case` The `switch-case` statement allows you to choose from many different paths based on the **VALUE** of some expression.

2.4.1.1 `if`

The `if` statement has a very basic syntax, shown below in Listing 3. Typically, this is only used when there needs to be a small change in the state of the program based off the predicate's value.

```
if (logical-predicate) {
        predicate-true-clause;
}

/* Note that the ( and ) MUST be present in the statement to denote the predicate.
 * { and } can be omitted ONLY IF the clause has a SINGLE statement in it. */
```

Listing 3: `if` Statement Syntax

2.4.1.2 `if else` The `if else` statement is used to perform two distinct actions based on the predicate's value. The syntax of this is shown in Listing 4.

```
if (logical-predicate) {
        predicate-true-clause;
} else {
        predicate-false-clause;
}

/* Note that the ( and ) MUST be present in the statement to denote the predicate.
 * { and } can be omitted ONLY IF the clause has a SINGLE statement in it. */
```

Listing 4: `if else` Statement Syntax

2.4.1.3 `if else if else` The `if else if else` is used to choose between n options. However, this comes with the downside that **EACH** potential path's predicate **MUST** be evaluated before any action can occur. The syntax for this is shown in Listing 5.

```
if (logical-predicate) {
        this-predicate-true-clause;
} else if (logical-predicate) {
        this-predicate-true-clause;
} else {
        all-predicates-false-clause;
}

/* Note that the ( and ) MUST be present in the statement to denote the predicate.
 * { and } can be omitted ONLY IF the clause has a SINGLE statement in it. */
```

Listing 5: `if else if else` Statement Syntax

2.4.1.4 `switch case` The `switch case` statement is used to choose between n different options. However, unlike the `if else if else`, the expression is only evaluated the once, and the branch is then jumped to in constant time, making this a better option for making one decision of many. The syntax for this is shown in Listing 6.

2.4.2 Repetition

Here, we want to perform some action a number of times. There are 3 structures for performing a repeating action:

`while` **Loop** This is generally used when the number of repetitions is unknown or uncountable at any given point in time. It continues to repeat until the predicate ceases to be `true`-thy or an explicit `break` occurs.

```
switch (predicate) {
case 1:
        break;
case 2:
        expression1;
case 3: {
        statement1;
        statement2;
}
default:
        break;
}

/* The ( and ) are MANDATORY for the switch to occur. In addition, the outer { and }
 * are MANDATORY for each of the cases.
 * If there will be many STATEMENTs in a case, then the body of the case MUST be
 * surrounded by { and }.
 * In addition, each case will "fall-through" by default, meaning the body of an
 * case will also execute the bodies of lower cases. The only way to prevent this
 * action is to include a break in the body of the case. */
```

Listing 6: `switch case` Statement Syntax

`for` **Loop** The `for` loop is used when the number of repetitions is both known and countable.

`do while` **Loop** The `do while` loop is the same as the `while` loop, but the difference is that the body of the loop is executed **ONCE before** evaluating the predicate.

2.4.2.1 `while` The `while` loop is typically used when the number of reptitions is unknown or uncountable. This allows for easy definition of infinite loops, allowing for a program to continue execution until some condition has been met. The syntax for this is shown in Listing 7.

```
while (condition) {
        statements;
}

/* The body of the while-loop will continue to execute UNTIL condition becomes
 * false.
 * The ( and ) are NEEDED for the predicate. In addition, the { and } are
 * NEEDED if there is more than one statement in the body.  */
```

Listing 7: `while` Loop Syntax

2.4.2.2 `for` The `for` loop is used when the number of repetitions is both countable. This type of loop can be modelled by the `while` as well, but this helps ensure that all the components of the loop are present, helping prevent infinite loops. The syntax of this is shown in Listing 8.

2.4.2.3 `do while` The `do while` loop performs some action an uncountable or unknown number of times, **AT LEAST** once. This is typically used when the action to be repeated can be repeated any number of times, but must happen at least once. The syntax for this is shown in Listing 9.

```c
for(init value; predicate; per-loop change) {
        statement;
}

/* The body of the ( and ) part must be written that way, and MUST be in the
 * parentheses.
 * The { and } are MANDATORY if there is MORE than 1 statement in the for loop.
 * The initial value MUST be countably changeable, meaning an integer or
 * something else.*/
```

Listing 8: `for` Loop Syntax

```c
do {
        statement;
}
while(condition);

/* The { and } SHOULD ALWAYS be included for the body of a do-while loop, although
 * the language and compiler do NOT require them. However, the ( and ) ARE required
 * to surround the condition.
 * In addition, the ending semicolon ; at the end of the while IS mandatory. */
```

Listing 9: `do while` Loop Syntax

2.5 Variables

Because C is a statically typed language, the type of every expression **MUST** be known before or during compilation. In addition, C compilers do not have any type inferencing, so you **MUST** explicitly tell the compiler the type of your variables. This means that you **MUST** declare before use. It is important to note that the declaration implicitly allocates storage for the data that will be stored.

One thing that will come up throughout this section is the concept of aliveness and scope. It is import to note that variables do **not** have to be in-scope to be alive, and vice versa.

2.5.1 Visibility

Visibility or *scope* is where a symbol can be seen from. If the symbol cannot be seen, then it cannot be used in any way

In addition, we need to ask *how* we can refer to the symbol. This includes what kid of identifiers/modifiers/namespacing is needed to identify the symbol in question.

2.5.1.1 Global Variables They **MUST** be declared outside any function. These are not deallocated **AT ANY TIME** during a program's execution, as they are always in-scope, however the variable may not be alive. Additionally, these are **ALWAYS** available, until the program terminates.

Using global variables is typically bad practice as this can introduce weird and hard-to-debug errors into a program. So, most variables that you will use will be local variables. Local variables are limited to the scope they were created within. Typically, the scope is a function, but can be an `if`, a `while`, etc.

2.5.1.2 `extern` Variables `extern` is used to denote a variable that is external to this program. This means the variable in question is a global variable in another file

The opposite of the `extern` keyword is the `static` keyword, which limits the scope of a symbol to the file it is declared in. In addition, when the variable is declared to be `static` , the value lasts throughout the program's execution, ensuring the variable is always available.

2.5.2 Lifetime

The lifetime of a variable asks how long a variable has storage allocated to it. In C, this is distinctly different than a symbol not being visible. One example of this is: a pointer can have the memory underneath it deallocated, ending the lifetime of the pointer, but keeping the pointer in-scope.

2.6 Functions

Functions are the highest form of modularity present in the language. Here, the procedural aspect of the language comes into play, and becomes useful. Repeated computations can be expressed as a function, allowing for easy code reuse and modularization.

Because C is so low-level, it has the unique distinction of requiring the programmer to separate the declaration of a function from its definition.

2.6.0.1 Declaration A declaration of a function simply announces to the compiler that a function with those input parameters and output values will be defined. However, a declaration says nothing about **HOW** the function will be defined, only that one such function will exist.

In C programming, function declarations (and sometimes some variable definitions) are placed in *header* files, that end with the `.h` extension. Sometimes, the function declarations here are called *function prototypes*.

An example of this is shown in Listing 10.

```
unsigned long hash(char *str);
hashtable_t *make_hashtable(unsigned long size);
void ht_put(hashtable_t *ht, char *key, void *val);
```

Listing 10: Declaration of `hashtable.h`

2.6.0.2 Definition The definition of a function is the actual implementation of a function. These are typically done in `.c` files.

An example of this is shown in Listing 11.

```
#include "hashtable.h"

unsigned long hash(char *str) {
        unsigned long hash = 5381;
        int c;
        while ((c = *str++))
                hash = ((hash << 5) + hash) + c;
        return 0;
}
```

```
#include "hashtable.h"

int main(int argc, char *argv[]) {
        hashtable_t *ht;
        ht = make_hashtable(atoi(argv[1]));
        return 0;
}
```

Listing 11: Definition of `hashtable.c`

2.6.1 Passing Parameters

The act of passing parameters is key to actually using functions in our code. There are two ways to pass parameters through to functions in C.

1. Pass-by-Value
2. Pass-by-Reference

2.6.1.1 Pass-by-Value Passing by value involves making a copy of the value passed to a function and giving it a new name. Since this is a **COPY**, it means **ANY** modifications to the copy **DO NOT** affect the original.

This is useful for small pieces of information, but the problem is when the size of the information increases. This is because the entire contents of a thing must be copied before running the function. This is where the other form of parameter passing comes into play.

2.6.1.2 Pass-by-Reference Passing by reference involves giving the called function a pointer (reference) to the data to manipulate. This is extremely efficient for very large data structures, because only the pointer to the data needs to be copied around. This is particularly valuable for large `struct` s, arrays, etc.

3 Pointers

4 Arrays

5 Strings

6 Memory Allocation

Because C is a language that does not provide many abstractions, it also requires the programmer to remember and manage their memory usage. So, **YOU** must be the one to manage the memory, there is **NO** built-in garbage collector for you to use.

Memory allocation is done on the heap of the program's execution space in memory. When you allocate memory in your program, you are actually requesting the operating system to give you the memory you want.

6.1 `malloc`

This is the simplest function of all possible memory allocation functions. `malloc`:

- Takes one argument:
  1. The number of bytes to allocate.
- Returns a **POINTER** to the front of the allocated memory.

`malloc` ***DOES NOT*** initialize memory, so it will be garbage.

6.2 `calloc`

This is quite similar to malloc. `calloc`:

- Takes 2 arguments:
  1. The number of spaces to allocate, for example the number of elements in an array.
  2. The number of bytes to allocate, for the type being stored.
- Returns a **POINTER** to the front of the allocated memory.

`calloc` ***ZEROS*** memory, so this does have a slight performance penalty.

6.3 `realloc`

`realloc` is used to **REALLOCATE** an existing memory location.

- Takes 2 arguments:
  1. The pointer to the memory location previously allocated with either `malloc` or `calloc`.
  2. The amount of memory to reallocate, in bytes.
- If the `NULL` pointer is passed to `realloc`, it will behave exactly like `malloc`.
- Returns a **POINTER** to the front of the reallocated memory

6.4 `free`

`free` is used to free memory that was previously allocated, removing from the programming space entirely.

- Takes 1 argument:
  1. A pointer to the memory to be deallocated.
- Returns `void`.

7 Composite Data Types

8 Compilation

A `man` Pages

References

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. English. Second. Prentice Hall, Feb. 22, 1978. ISBN: 9780131101630.