C Language CheatSheet

Adminstriva

- All C programs consist of a series of *functions*, which have return values!
- E.g., Assignment `x = v` is the function that updates the value of name `x` with the value of `v` then terminates yielding `v` as a return value.
- Semicolons act as statement *terminators* —in contrast to English, wherein they are *separators*.
- C is whitespace agnostic: Newlines and arbitrary spaces don't matter, for the most part.
- *Compound statements* are formed from primitive statements by enclosing them in `{` curly braces `}`.
  - Compounds may appear where-ever a single statement is allowed.
- All keywords are in lowercase.
- C comments do *not* nest.
- Include files personal files by enclosing them in "quotes", use <brackets> for standard library files.
- *All* statements must terminate with a semicolon.
- Everything must be declared before it can be used —sequential.
- Characters are in-fact aliases for ASCII numerals.

```
printf("'A' ≈ %d  ≡  %s", 'A', 'A' == 65? "true" : "false");
printf("\n'Z'  23  ≈  %c  ≡  %d", 'Z' - 23, 'Z' - 23);
```

```
'A'  ≈  65  ≡  true
'Z'     23  ≈  C     ≡  67
```

- C functions don't have to return anything if it's not appropriate.
  - E.g., `exit(n)` is the function that returns control to the operating system; passing it an argument `n`, usually 0 if everything has gone smoothly and 1 if it's an error exit. Yet this function obviously' can't *return* a value.
- You must specify the type of a variable before you can use it.
- A variable name is only meaningful in the curly brackets that define it, and is otherwise meaningless. This is its *scope*.
  - Whence, the same names can occur in different places to mean different things.
  - To transfer data between functions one thus uses parameter lists and return calls.
- `printf`, "print formatted", is a *dependently-typed function*: The number and type of its arguments depends on its first argument, a string.
  - The number of occurrences of '%' in the string argument is the number of additional arguments the `printf` takes.

Conditionals & Assertion-Based Testing

In C, *true* ≈ *non-zero*. Form of the conditional:

```
if (condition)
  statementBlock
// The rest is optional.
else
  statementBlock
```

- `condition` is *any* expression that returns a numeric value: All numbers are treated as 'true', except 0 which is considered 'false'.

Asserts are essentially compile-checked comments of user intentions!

`assert(e)` does nothing when expression `e` is true; otherwise it gives a message showing the filename, function name, line number, and the condition `e` that failed to be true.

```
#include <stdio.h>

// Disable assertions at compile time by enabling NODEBUG.
// #define NDEBUG

#include <assert.h>
// assert(n)  ≈  if (n) {} else ⟪Terminate and display error message⟫

int sum(int n)
{
  int total = 0, i = 0;
  while (i != n + 1)
    total += i, i++;

  return total;
}

int main ()
{
  // print-based testing
  if (1) printf("here"); else printf("there");
  printf("Sum of 0 + 1 + ··· + 99 + 100 = %d", sum(100));

  // assertion-based testing
  assert( sum(100) == 5050 );
  assert( (1 ? "here" : "there") == "here" );

  // Is completely ignored if the #define is enabled.
  // assert(0);  // Otherwise, this causes a crash.

  return 0;
}
```

```
hereSum of 0 + 1 + ··· + 99 + 100 = 5050
```

Enforce a particular precedence order by enclosing expressions in parentheses.

```
== equals         != differs from    !  not
>= at least       <= at most         && and
>  greater than   <  less than       || or
```

Assignments

```
/* Abbreviations */
  x ⊕= y   ≈   x = x ⊕ y
  x++       ≈   x += 1
  --x       ≈   x -= 1
```

The increment and decrement, ++/--, operators may precede or follow a name:

- If they follow a name, then their behaviour is executed *after* the smallest context —e.g., braces or conditional parentheses— in which they occur.
- When they precede a name, their behaviour is executed before the context in which they appear.
- The order of evaluation is not specified inside a function call and so behaviour varies between compilers.

Avoid using these in complex expressions, unless you know what you're doing.

Loops

Here's the general form.

```
while (condition)
  statementBlock
```

```
/* Abbreviations */
/* for      loop */   for(A; B; C;) S  ≈  A; while(B) S
/* do-while loop */   do S while B     ≈  S; while(B) S
```

do/while: The conditional is evaluated *after* the statement has been executed and so the statement is obeyed at least once, regardless of the truth or falsity of the condition. This is useful for *do once, and possible more* operations.

```
int i = 0;
do printf("%d \n", i++);
while (i != 10); //Note the ending semicolon.
```

Arithmetic and Logic

> *. . . and then the different branches of arithmetic —Ambition, Distraction, Uglification, and Derision.*
>
> —Alice's Adventures in Wonderland

The modulus operator % gives the *remainder* of a division.

```
printf("rem = 10 %% 3 = %d   ⇒   ∃ quot •  10 = 3 × quot + rem  ∧  quot = %d", 10 % 3, 10 / 3);
```

```
rem = 10 % 3 = 1   ⇒  ∃ quot •  10 = 3 × quot + rem  ∧  quot = 3
```

- In conditionals, one may see `n % d` to mean that `n % d` is true, i.e., is *non-zero*. This expresses that `n` is a *multiple* of `d`.
- That is, numerically % yields remainders; but logically, in C, it expresses the is-multiple-of relationship.

When `x` is a number, the shift operations correspond to multiplication and division by 2^n, respectively.

Left Shift	`x << n`	append the bit representation of `x` with `n`-many 0s
Right Shift	`x >> n`	throw away `n` bits from the end of the bit representation of `x`

The *bitwise* operators *and &, or |, not !,* and *xor* ^ operate at the bit representation of an item. For example, the ASCII code of a character consists of 7 bits where

Bit	Function
7	0 digit, 1 letter
6	0 upper case, 1 digit or lower case
5	0 for a-o, 1 for digit or p-z

Whence, to convert a character to uppercase it suffices to change bit 5 to be a 0 and leave the other bits alone. That is, to perform a bitwise *and* with the binary number *11011111*, which corresponds to the decimal number 223.

```
// Mask, or hide, bit  6 to be a '1'.
#define toLower(c) (c | (1 << 6))

#define toUpper(c) (c & 223)

#define times10(x) ( (x << 1)  +  (x << 3)) // Parens matter!
// x   ⇒   2·x + 8·x   ⇒   10 · x
```

How did we know is was 223?

0.	Ninth bit is on	`100000000`	`1 << 8`
1.	Negate it: Eight ones	`11111111`	`~(1 << 8)`
2.	Sixth bit is on	`100000`	`1 << 5`
3.	Xor them	`11011111`	`... ^ ...`
4.	See it as a decimal	`223`	

Ironically, C has no primitive binary printing utility.

Floats & Other Types

- float: Representing huge numbers to tiny fractions.
- double: Like float, but greater precision.
- int: The integers; it has 2 *subtypes*; namely short, long, and unsigned.
  - Short and long allocate less (half) or more (twice) memory, respectively.
  - Unsigned means we omit negative numbers, and therefore have twice as many non-negative numbers since the sign is no longer a necessary place-holder.
    - ⋆ Also useful when negative integers are unreasonable for the current task.

Type aliases: `typedef existingType newName;`

```
// A named product: String × ℤ × ℝ   having names  name, age, height.
struct entry
{
  char   name[30];
```

```
  short  age;
  float  height;
}; // Note that this is a statement

// Alias for readability.
typedef   struct entry   Person;

int main()
{                                              // Exponent form for numbers.
  struct entry qasim = {"qasim", 0, 613e-2}; // 613e-2 ⇒ 613 × 10⁻² ⇒ 6.13
  Person q = qasim;
  q.age = 23;

  printf("Hello! My name is %s and I’m %d years old, being %f ft tall",
     q.name, q.age, q.height);
  return 0;
}
```

Hello! My name is qasim and I'm 23 years old being 6.130000 ft tall

Forming Numbers using Octal & Hexadecimal

Numbers that begin with a 0 are interpreted as octal and those beginning with 0x or 0X are hexadecimal. These forms are obtained from binary by grouping using 3 bits and 4 bits, respectively. E.g., `129` ≈ `0201` ≈ `0x81` since:

- 129 →⟨binary⟩ 10 000 001 →⟨octal⟩ 2 0 1
- 129 →⟨binary⟩ 1000 0001 →⟨hexadecimal⟩ 8 1

Indeed:

```
int x = 129, y = 0201, z = 0x81, zz = 0X81, b = (x == y) && (x == zz);
printf("%d ≈ %d ≈ %d; %d", x, y, z, b);
```

```
129 ≈ 129 ≈ 129; 1
```

The Preprocessor

The preprocessor performs alterations to a program *before* it is compiled; e.g., the following ensures the replacement of every occurrence of `this(··· , ···)` after it with whatever `that` is.

```
#define this(arg1, arg2)  that
```

- All preprocessor commands are preceded by the `#` symbol.
- One generally defines useful constants or abbreviations this way, and if they have a collection of such parameters or utilities in some file, then they can *copy-paste* them into the current program file by using `#include` as mentioned before.

Pointers

- A fly is a bug that flies; so a *fly flies*.
- Likewise, a *pointer points*.

Variables may be used without having values declared! This is akin to buying the land —computer memory— but not building the house —assigning a value— thereby leaving us with a vacant lot that contains trees, garbage, and whatever was there before you get there —as is the case in C.

```
char c;
int i;
long x;
float f;
double d;

// Uninitalised ⇒ have random values.

printf("char   %c\n",c);
printf("int    %i\n",i);
printf("long   %l\n",x);
printf("float  %f\n",f);
printf("double %d\n",d);
```

#RESULTS:

char	
int	32766
long	
float	0.0
double	73896

Rather than being set to 0, variables automatically obtain the random values in memory and one should always initialise variables upon declaration.

- The amount of space a variable needs is its *size* in memory.
- The actual location of the variable in memory is called its *address* —just like the address on your house or on an ugly vacant lot.
- In your computer, memory space is measured in bytes.
  - Not every computer uses the same storage space for its variables.
  - The `sizeof` keyword tells us how many bytes a variable, or structure, uses up.
    - ⋆ Both sizeof(x) and sizeof(int) are valid calls.
  - This keyword is used primarily for manual allocation of memory for new variables we create.

```
char c;
int i;
long x;
float f;
double d;

// Uninitalised ⇒ have random values.

printf("char   %i\n",sizeof(c));
printf("int    %i\n",sizeof(i));
printf("ℤ      %i\n",sizeof(int));
printf("long   %i\n",sizeof(x));
printf("float  %i\n",sizeof(f));
printf("double %i\n",sizeof(d));
```

```c
#define Length 123
#define Type   char
Type arr[Length];

assert( sizeof(arr) == sizeof(Type) * Length );
```

- Memory is set aside for variables, even if they're not used.
  - To avoid hogging up excess memory, programs that need a lot of memory usually request it a little at a time using `malloc` —'m'emory 'alloc'ation.
- C is a mid-level language that has the capability to examine memory and the variables stored there.
- Why care about *where* variables are located in memory?

  Consider sorting an array of complex data, rather than moving the data itself around, it is much cheaper to simply sort an array corresponding to their numeric locations in memory.
  - The variables that hold addresses are called *pointers*.
- A variable consists of 4 pieces: Name, type, size, and location in memory.
  - The first two we know, since we write them down.
  - The third is obtained via the `sizeof` operator.
  - The fourth is obtained by using the "address of" operator `&`, resulting in a pointer. (One may think of '&' as simply projecting the fourth component from a variable structure.)

  A pointer is a variable that holds a memory address.
  - Pointers are represented as numeric locations, but they are not number values.
- `T*` is denotes pointers of type `T`.
  - Pointers are declared like any other variable.
  - Conventionally a declaration looks like `T *t;` —the whitespace around the '*' is completely irrelevant— as a reminder that the expression `*t` denotes a value of type `T`. Huh?
    - ⋆ Without '*', pointers consume and represent memory locations.
    - ⋆ With '*', they denote a value of type `T`.

    Unless you're working with pointers to pointers, you do not want to write down `*p = &x`, which sets the *value* at the location of `p` to refer to an *address*.

```c
// Valid declarations
int*p;
int* q;
int *r;
int * s;

printf("p is location %u \n", p);

char arr[] = "hello";
p = &arr; p = &arr[0]; p = arr;
printf("p is location %u \n", p);

// For arrays, the array name and the address of the array both refer to the address
// of the zeroth element.
assert( &arr    == arr );
assert( &arr[0] == arr );
```

1. Here's some useful conversion laws

   ***&-inverses** `p = &x` $\equiv$ `*p = x`

   - The lhs lives in the land of addresses and locations, whereas the rhs lives in the land of values.

   []&-array `arr + i = &arr[i]`

   From these we obtain:

   []*-array `*(arr + i) = arr[i]` —immediate from the above two laws.

   ***-array** `*arr = arr[0]` —immediate from the above two laws.

   ***-array** `**arr = arr[0][0]` —the previous law iterated twice.

Tips:

✓ Pointers must be declared with the asterisk
✓ When a pointer is used without its asterisk, it refers to a memory location
✓ When a pointer is used with its asterisk, it indirectly refers to a value
✓ To assign a pointer a memory location, prefix the other variable with &
✓ Pointers are memory addresses and can be displayed as unsigned integers using `%u`

```c
int a[30], *p;

// Make p point to the beginning of the array a, since a is itself a pointer to the
// beginning of the array.
p = a;

// Determine the actual machine address of a variable by predicing it with &.
int fred;
p = &fred;

// If we need to know exactly where in the machine C had decidede to allocate space
// to fred, we need only to print out p; whence & is read "address of".
printf("%p", p);
```

```
0x7ffee36348ac
```

Since the name of an array is itself a pointer to the beginning of an array, we have

$$\text{arr} \approx \&\text{arr}[0]$$

```c
// Find furthest points of 'int a[N-1]' that are identical.
#define N 12
int a[] = {9,8,1,2,3,4,4,3,2,1,7,6};

int * p = a, *q = &a[N-1];
while(*p != *q) p++, q--;

assert(*p == a[2] && *q == a[9]);
```

Before C99, function return types could be omitted with the understanding that the default is `int`. However, `main` still needs its return type. The following works with many warnings.

```c
f() {return 9; }
g() {return 'c'; } // Remember that chars are actually numbers.

int main() { printf("%d", f() + g()); return 0; }
```

```
108
```

Arrays and Strings and Things

- C arrays are homogeneous —all elements are of the same type— and their size is static —one cannot readily change how many elements they contain.
- `T name[N]` ⇒ Declares `name` to be an array of type `T` consisting of `N` elements.
  - In an initialisation, `N` may be omitted and is then inferred.
    - ⋆ E.g., `int a[] = {2, 4, 6};`
    - ⋆ E.g., `T a[N] = {x0, ..., xM}` for `M < N` initialises the first `M` elements and the remaining `N - M` are uninitialised, and may consist of random junk.
- Array access and modification is done with square brackets: `arr[int_expression]` may be treated as a normal variable wherever it occurs.
- Strings are just single-character arrays.

```c
char hello  [] = "hello";
char hello1 [] = {'h', 'e', 'l', 'l', 'o', '\0'};
char hello2 [6] = {'h', 'e', 'l', 'l', 'o'}; // Null byte included automatically.

// The null byte is not considered part of the string,
// since it's only a terminating marker.
// As such, it's not counted in the length of the string.

// Strings end in the null byte, 0x00 aka \0;
// that's how C knows where the end of a string is;
// otherwise it'd keep looking for the null byte,
// scanning memory then crashing.

char nope [] = {'h', 'e', 'l', 'l', 'o'}; // Just an array of chars, neither size nor \0 at the end to indicate that this is a string.

// The array name suffices to refer to the whole string,
// no need to mention the brackets.

// Strcmp succeeds when its arguments are different;
// and fails (returning 0) if they're the same.
#define Equal(s,t) assert(strcmp(s, t) == 0);

Equal(hello, "hello")
Equal(hello1, "hello")
Equal(hello2, "hello")

assert(strcmp(nope, "hello")); // different args.

// C stops looking when the null byte is found!
char surprise [] = "surprise! No more, my friend";
assert(strlen(surprise) == 28);

surprise[8] = '\0';
Equal(surprise, "surprise")
```

- string.h contains the strX functions.
- Multi-dimensional arrays: `T arr[d1][d2]...[dN]`.
  - Just as chars are numbers, multi-dimensional arrays are plain one-dimensional arrays: The multiple-dimension declaration syntax simply tells the compiler how to organise the array into (nested) groups and the indexing tells it which (nested) group we're interested in working with.

```c
#define D0 4
#define D1 3
#define D2 2

int arr [D0][D1][D2] = // 4 groups consisting of 3 rows with 2 items in each row
  {
    0, 1, // Items of arr[0][0][]
    2, 3, // Items of arr[0][1][]
    4, 5, // Items of arr[0][2][]

    // Items of arr[1][][]
    6, 7,
    8, 9,
    10, 11,

    // Items of arr[2][][]
    12, 13,
    14, 15,
    16, 17,

    // Items of arr[3][][]
    18, 19,
    20, 21,
    22, 23
  };

// Third group's second row, second item is 15.
assert(arr[2][1][1] == 15); // Note: 2*D0 + 1*D1 + 1*D2 ≈ (+ 8 3 2)

assert(arr[0][1][0] == 2);

int nope[2][1] = {2, 4};
assert( *(nope + 0) == 2);
// assert( *(arr + (3*D0 + 2*D1 + 2*D2)) == 15);

// Anomaly:
// char words [a][b][c]
// ⇒ has a lines, consisting of b words, where each consists of c many characters
```

- The uppercase letters start at ASCII code 65 and lowercase start at code 97.
- The compiler sees all characters as numbers.

- The idea of a pointer is central to the C programming philosophy.

  - It is pointers to strings, rather than strings themselves, that're passed around in a C program.
- C strings like `s = "this"` actually, under the hood, are *null-terminated arrays of characters*: The name `s` refers to the *address of* the first character, the `'t'`, with the array being `'t'` → `'h'` → `'i'` → `'s'` → 𝟘, where 𝟘 is *not* ASCII zero —whose value is 48— but ASCII *null* —i.e., all bits set to 0.
  - Note that null 𝟘 is denoted by `'\0'` and has *value* 0, i.e., false, so conditions of the form `p != '\0'` are the same as `p`.
- *An array name is a pointer to the beginning of the array.*
  - Yet, an array name is a constant and you can't do arithmetic with it.

```c
int length(char c[]) // A string is a character array
{
  // While c[l] is not an ASCII null, keep counting until.
  int l = 0;
  while( c[l] ) l++;
  return l;
}

int main()
{
  char str[] = "hello world 0123";
  printf("length(‘‘%s’’) = %d", str, length(str));
   return 0;
}
```

`length(‘‘hello world 0123’’) = 16`

- `T *p;` ⇒ declare `p` to be a pointer to elements of type `T`.
- `*p = v` ⇒ "put the value of `v` in the location which `p` points to"

We can now rewrite the length function with even less square brackets.

```c
int length(char* c)
{
  char* start = c;
  while( *c ) c++; // Local copy of c is affected.
  return c - start;
}

assert(length("hello world") == 11);
```

Input –getting things *into* the machine

- `getchar` returns the next character input from the keyboard.

```c
#include <stdio.h>

int main()
{
  printf("Please enter a character: ");
  char c = getchar();
  printf("You entered: %c", c);
  printf("\nBye\n");

  return 0;
}
```

Let's extend `getchar` to work on buffers of length, say, 20:

```c
#include <stdio.h>

#define BUFFER_LENGTH 10

int main()
{
  printf("Please enter a string, only first %d chars or newline will be read: \n\t\
         BUFFER_LENGTH);
  char cs[BUFFER_LENGTH];
  int keep_reading = 1;
  char * p = cs;
  while(keep_reading && p < cs + BUFFER_LENGTH)
    {*p = getchar(); if (*p == '\n') keep_reading = 0; else p++;}

  *p = '\0'; // Strings are a null-terminated arrays of chars.

  printf("\nYou entered: %s", cs);
  printf("\nBye\n");

  return 0;
}
```