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.

  `'A'` $\approx$ `65` $\equiv$ `true`
- 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* $\approx$ *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;
}
```

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.

- 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`, 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.

Strings, Arrays, and Pointers

- 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'` $\to$ `'h'` $\to$ `'i'` $\to$ `'s'` $\to$ $\mathbb{0}$, where $\mathbb{0}$ is *not* ASCII zero —whose value is 48— but ASCII *null* —i.e., all bits set to 0.
- *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.

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

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

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

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