C Cheat Sheet

Commonly used headers

stdio, stdlib, string, math

```
#include <stdio.h>
```

Using relative paths:

```
#include "..\my_header.h"
```

Pitfalls:

- Incorrect spelling. No “;” needed.

if else

```
if(<condition>)
{
 <statement>
}
else
{
 <other statement>
}
```

Pitfalls:

- Forgetting brackets:
```
  if(pincode == 1234)
    printf("pincode correct");
    transferFunds();
```

- Using the assign operator instead of the compare operator:
```
  if(crashLandDrone = 1)
  { /* Writes the value 1 in
  crashLandDrone, then evaluates the
  value between the ()-brackets. Since
  it is non-zero, it is equal true,
  which means the if-statement content
  is executed. */
    initiateCrash();
  }
```

- An extra semicolon:
```
  if(crashLandDrone == 1);
  { /* will always call the crash
  function. The if-statement ends at
  the ; and the {} are interpreted as
  scope-operators.*/
    initiateCrash();
  }
```

Loops

```
while(<condition>)
{
  <statements>
}

for(<initial>; <condition>; <update>)
{
  <statements>
}

do
{
  <statements>
}
while(<condition>);
```

Pitfalls:

- An update statement that does not update:
```
int i;
for(i=0; i<10; i+1) //i+1 does nothing
```
- A for loop uses semicolon separators (;), not commas!

Arithmetic operators

+ (add), - (subtract), *(multiply), / (division), % (modulo)

Shorthand

```
b += a; // b = b + a;
c >>= 1; // c = c >> 1;
```

Comments:

```
// Single line
/* Multi
 line */
```

Pitfalls:

- You can nest single line comments in a multi line comment, but you can't nest multiple multi line comments

Relational operators

== (equal to), != (not equal to), > (greater then), < (less then), <= (less than or equal to), >= (greater than or equal to)

Logical operators

|| (logical OR), && (Logical AND), !(Logical NOT)

Bitwise operators

| (bitwise OR), & (Bitwise AND), ~(Bitwise invert), << (Shift left), >> (Shift right)

Pitfalls:

- Unintentionally using bitwise operators in if statements:
```
  a = 1; b = 2;
  if( a & b )
  { // The printf is NOT be executed, since 0x01 & 0x02 = 0
    printf("a and b are not equal to 0\n");
  }
```

Arrays

```
// An array called arr which can store 5 integers
int arr[5];
// Add 1 to the THIRD element in arr
b = arr[2] + 1;
// Initialize.
int other[2] = {52, 356};
```

Pitfalls:

- Indexing an array at invalid locations:
```
  a = arr[5];     // index 5 is illegal
  int c = 2, d=3;
  a = arr[c-d];   // index -1 is illegal
```

Data or variable types

```
int, float, double, char, bool, void
```

Pitfalls:

- Integer arithmetic floors results: 7/2 = 3

- Numbers stored in integers are floored:
```
  int a = 3.0/4.0; // results in a = 0
```

- Floats and doubles cannot represent all numbers (especially nasty when used as loop iterator):
```
  float a = 16777216;
  a = a + 1;
  printf("a: %f", a); // a: 16777216.000 (so apparently
           // 16777217.000 cannot be represented as a float)
```

- Not initializing variables:
```
  int a;           // Unknown value
  int b = a + 1; // Unknown value2
```

printf format specifiers

%i, %d: int
%u: unsigned int
%f: float
%lf: double (remember: Long Float)
%c: char
%s: string (make sure there is a \0 char at the end of the string!)
%x: hexadecimal

Pitfalls:

- printf doesn’t check the types of its input arguments! However, they are cast to the type of the specifier when printed:

```
int c = 3;
printf("c: %s", c);
/* This will cast the variable c to char* (a
pointer to a char). This means that the
number 3 is used as the address from which
the printf will start printing bytes, until
the first byte of value 0 (='\0') is found.
Very evil (and wrong). */
```

Strings (arrays of chars)

```
// An array called arr, which can store 5
// chars
char arr[5];
// Initializes myStr with 't', 'e', 's',
't', // '\0'
char myStr[5] = "test";
char name[] = "Compiler determines length, and
accounts for the \\0";
```

Pitfalls & Remarks:

- You can only initialize once. (But you can strcpy into the string later).

```
char name[] = "My name is Bob";
strcpy(name, "I'm Alice");
```

  – Incorrectly comparing strings in if-statements:

```
char t[] = "test";
if(t == "test"){ // <- VERY WRONG!
  // This compares the address of t with the
  // address of the constant "test".
  // Probably NOT what you would want. Use
  // if(strcmp(t, "test") == 0) instead.
}
```

- Overwriting the \0 char:

```
char arr[4];
/* This strcpy writes a 't' at location
arr[3], and a \0 at an invalid location in
the memory. arr can only store 4 chars.*/
strcpy(arr, "test");
```

scanf

```
scanf(" %d", &myInt);
```

Pitfalls & Remarks:

- Include the leading space in the pattern to ignore all leading whitespace chars in the user input.
- Give scanf the address of the variable in which the input must be stored (include a **&** for all non-pointer types).
- Make sure the data types match the expected input
- (For real-life applications: never trust the user !)

Declaring functions

```
<return type> function_name(<arg1>, <arg2>, …)
{
  <statements>
}
// Function that determines the result of a
// quadratic function of form y(x)=a+bx^2+c
float quad(float x, float a,
           float b, float c){
  float ans = a + b * x;
  ans += c * x * x;
  return ans;
}
// Function that returns the 4th array element:
int fourth(int arr[]){
  return arr[3];
}
/* In general, variables are passed by value, i.e.
the function receives copies of the variables you
use as arguments: */
void doesNothing(int input){
  input = 500;
}
/* Exception: passing variables as pointers
allows you to edit them inside other functions.
The 3 arguments of the next function are all
pointers: */
void doesSomething(float* num, int list[],
                   char word[10]){
  *num = 9.3;
  list[2] = 5;
  strcpy(word,"test");
}

void main(void){
  int var = 9001;   // name does not matter
  doesNothing(var); // call our function
  printf("var: %i", var); // prints: var: 9001

  float myFloat = 1337.0;
  char t[10];
  int myList[3] = {1,2,3};
  doesSomething(&myFloat, myList, t);
  printf("%s - %f - %i", t, myFloat, myList[2]);
  // prints: test - 9.300 - 5
}
```

String manipulation functions (strcpy, strcmp, strlen, strcat, sprintf, toupper, ispunct, etc)

```
char * strcpy ( char * destination, const char * source );
```

Copy source into destination. Make sure there is enough space at the destination!

```
int strcmp ( const char * str1, const char * str2 );
```

Compares the string str1 to the string str2. Returns 0 when they are equal. A value greater than zero indicates that the first character that does not match has a greater value in str1 than in str2; And a value less than zero indicates the opposite. Can be used to sort alphabetically.
Most string functions have case insensitive versions (stricmp), or length delimited versions (strncpy).

Best Practices

- Write lots of comments (which will help yourself understand your own code (so you can reuse it next week/month/year)). Others (like your instructors, boss, colleagues) will also appreciate this.
- Even better: start with a skeleton of comments, and fill it in with code as you go along!
- Use a consistent indentation style and variable naming (camelCase or underscore_separated).
- Think about the problem you are trying to solve, and break it down into small parts.
- Use the debugging tools you have: for example, add extra printf's to display intermediate variables, or place breakpoints and inspect the contents of variables (using the “locals” tab, or the mouseover messages).