Compiled C

`C` is a compiled language.

Compile steps:

source	original file
pre-processor	clean up file (remove comments, expand macros/`#define`)
compile	create machine instructions
assembler	convert this to raw binary functions
linker	link func calls to defs
executable	bundle into final file

Compile `input.c` into exec `output`

```
gcc -Wall -Wextra input.c -o output
```

Escape sequences

`\n`, newline; `\0`, null character

Numerical Data

Integer/base 10: `76`

Binary `0100 1100`

Converting from decimal to binary, repeatedly ÷ 2, and taking the remainder.

$$\begin{aligned} 41 \div 2 &= 20r\ 1 \\ 20 \div 2 &= 10r\ 0 \\ 10 \div 2 &= 5r\ \ 0 \\ 5 \div 2 &= 2r\ \ 1 \\ 2 \div 2 &= 1r\ \ 0 \\ 1 \div 2 &= 0r\ \ 1 \end{aligned}$$

result is `0b101001`, you have to write it backwards to be correct (*bottom to top*)

Octal (base 8): `0114`

Hex (base 16): `0x4C`

Floating point numbers: $m \times b^e$, with m: mantissa, b: base (2), e: exponent.

String Types

Uses `ASCII` for string representation, i.e., `'A'` is 41, `'a'` is 61.

Creating strings,

```
char* my_str = "frog"; // CANNOT modify later
char my_mod_str[] = "turtle"; // can mod.
```

Data Types

Qualifiers,

- `unsigned`, no sign bit, can wrap around add/sub. Postfix: `U`
- `long`, *doubles* the byte count/storage. P.: `L`
- `short`, *halves* the byte count
- `const`, makes it unchangeable/'static'

For `signed` types the behavior around integer overflow is not defined, compiler dependent.

Operators

`+`, `+=` ~ compound assignment & op, `-`, `-=`, `*`, `*=`, `/`, `/=`, `%` ~ modulo `100 % 9 = 1` - remainder, `%=`

`++` inline increment, `--` inline decrement, pre/post fix

Postfix modifies the value after the value has been used.

```
int b = 10;
int a = b++; // a is 10, b is 11
int c = ++b; // c & a are 12
```

Type cast: `(int)(5.401) = 5`. Truncation of div: `(5 / 9) = 0`.

Relational

EQ `==`, NEQ `!=`, GT `>`, GTE `>=`, LT `<`, LTE `<=`

Logical

AND `&&`, OR `||`, NOT `!`

Ternary/Conditional, `(a > b) ? 'Y' : 'N'`

Define

Can define user constants with,

```
#define MY_COOL_STR "Hello!"
#define MY_INT 10
#define MY_INT_INT MY_INT * 2
```

Arrays

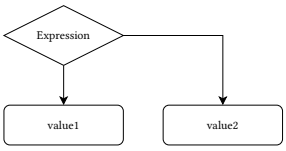
```
char name[3] = "Hi"; /* ['H', 'i', '\0'] */
int yes[] = {1, 3, 4, 5}; // auto size of 4
float empty[100]; // empty
```

Size of an array: `sizeof(arr)/sizeof(arr[0])`, this only works for local array doesn't work for an array passed into a function.

Passing arrays into a function

```
int my_arr[] = {1, 2, 3, 4};

void my_func(int arr[], char** sarr) {
  // arr points to the first elem in the array, allows for modification - but no idea of size
  // sarr is an array of strings (which itself an array of chars)
}
```

Branching



if,

```
if (a > b) {
  return 'Y';
} else if (a == b) {
  return 'M';
} else {
  return 'N';
}
```

while,

```
while (a < 5) {
  a++;
}
```

do while,

```
do {
  a++;
} while (a < 5);
```

for,

```
for (int i = 0; i < 5; i++) {
  printf("%d\n", i);
}
```

Switch

```
switch (moon_state):
  case 1:
    moon = "🌒"
    break; // stops code from falling through
  default:
    moon = "🌓"
    break;
```

Functions

Functions, user-defined - can take in args of certain type and return of other types.

Function prototype: `char my_func(int my_arg, float* p, char my_str[])`. With `char` being the return type. Defined at top of file or in header.

Main function,

```
int main(int argc, char** argv) {
  // argc number of args pass in through argv
  // argv[0] == name of program
  return 1; // 1 successful, 0 not
}
```

`sizeof` the size of a type in **bytes**, `sizeof((int) 10) = 4`

Pointers

```
int d = 10;
int *p = &d; // points to 10 value
int pd = *d; // val: 10

*p = 500; // now d is 500 too
```

`void*`, special pointer which points to data of *any* type in memory. Has to be casted to be used. i.e., `void *ptr = &b; // b is 10 (int *)ptr // 10`

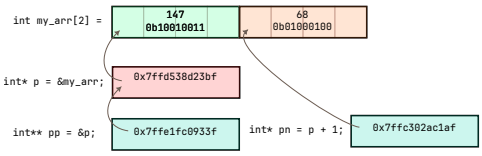
`NULL`, special pointer which a *nothing* value

Pointer arithmetic

```
int my_arr[] = {0, 1, 2, 3, 4, 5};
int* my_ptr = my_arr; // points to 0, don't need & as my_arr is already ptr
*(my_ptr + 2) // 2
*(++my_ptr) // 1
```

`const int* ptr`, cannot be used to change data.

functions cannot return a pointer, as the pointed to memory would have been deallocated/freed



Scope

```
#include <stdio.h>
int G = 10; // file scope
static int S_FILE = 20;
void P(int a, int b); // Function Prototype Scope

void S_F() {
    static int S_FUNC = 0; S_FUNC++; // persists between calls
}

int main() {
    int L = 30; // Function Scope
    printf("FileG:%d, StaticF:%d, Extern:%d\n", G, S_FILE, EXT_VAR);
    { int L = 50; printf("Block L: %d\n", L); } // Nested/Block
    printf("Func L: %d\n", L); // Back to L=30
    P(100, 200);
    return 0;
}
// File Scope: Definition for EXT_VAR
int EXT_VAR = 99;

void P(int x, int y) { // func defined above
    printf("  P Result: %d\n", x + y);
}
```

Multifile Code

```
// main.c
#include "animal.h"

int main() {
  frog(); // now has access to frog func!
  BEST_ANIMAL == "🐠"; // 1/true
  WORST_ANIMAL == "🦏"; // 1/true
  SEAL == "🦭"; // runtime error
}
```

```
// animal.c
const char* frog() {
  return "🐸";
}
const char* SEAL = "🦭";
```

```
// animal.h
const char* frog();
#define BEST_ANIMAL "🐠"
const char* WORST_ANIMAL = "🦏";
```

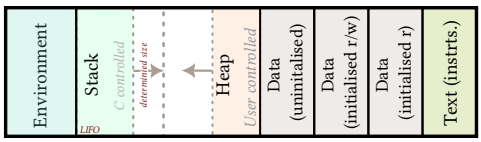
Header guard, `#ifndef MY_HEADER_H`, `#define MY_HEADER_H` then `#endif` - stops header content being linked multiple times.

Makefile

```
# basic syntax
target : dependency
  command
```

```
CC=gcc
CFLAGS=-g -Wall -Wextra -std=c99 -Iinclude
LIBS=-lm
TARGET=proc_node

SRC_DIR = src
OBJ_DIR = build
INC_DIR = include
# you may need to put your extra files here
_DEPS = message.h defs.h control.h array.h
_OBJS = main.o array.o control.o

DEPS = $(patsubst %,$(INC_DIR)/%,$(_DEPS))
OBJS = $(patsubst %,$(OBJ_DIR)/%,$(_OBJS))


$(OBJ_DIR)/%.o: $(SRC_DIR)/%.c $(DEPS)
  $(CC) -c -o $@ $< $(CFLAGS)

$(TARGET): $(OBJS)
  $(CC) -o $(TARGET) $(OBJS) $(LIBS)

.PHONY: clean
clean:
  $(RM) $(TARGET) $(OBJS)
```

Dynamic memory



Dynamic memory functions

- `malloc(size_a)`, allocates memory for the desired size
- `calloc()`, like `malloc` but clears the memory before
- `*realloc(void *ptr, size_t new_size)`, resizes or allocated a block of memory
- `free`, frees a block of memory from *heap*

expandable array, `int *data = (int *)malloc(capacity*sizeof(int));`

Rules:

1. always free allocated memory
2. Should only free memory you *allocated*, and only free once
3. Once memory is freed, you cannot access it again

```
double **create_matrix(int m, int n) { // dynamically created matrix
  double **p = (double**)malloc(m*sizeof(double*)); // beacuse malloc kept alive
  for (int i = 0; i < m; i++) {
    p[i] = (double*)malloc(n*sizeof(double));
  }
  return p;
}
```

Structures

```
typedef struct my_struct {
  int field;
  char my_bit_field: 2; // only takes up 2 bits
  struct* my_struct; // pointer to my own struct
} MyStruct;
```

Accessing fields,

```
MyStruct s = {1, 0x02, NULL}; // doesn't point to anything
MyStruct* ps = &s;
s->field; // 1, equiv to (*s).field
MyStruct sc = s; // copies fields
```

File `i/o`

`FILE *fp = fopen(name, mode);`

mode	types	a/c if file ex.
`r`	read	r. from start
`w`	write	overwrite
`a`	append	w. to end
`r+`	read/write	r./w. from start
`w+`	read/write	overwrite
`a+`	read/write	r./w. from end

then close, `fclose(FILE* stream)`

Standard streams,

Name	Buf?	from
stdin	✓	keyboard
stdout	✓	screen
stderr		screen

command	desc
`fgetc(FILE* fp)` `getchar()`	get a character
`fputc(intc, FILE* fp)` `putchar(char c)`	write a character
`fputs(char* str, FILE* fp)`	write a string
`fgets(char* str, int count, FILE* fp)`	get a string into a buffer
`printf(char* format, ...)`	print string to stdout
`scanf (char * format, ...)`	get formatted string from stdin
`fprintf(FILE* fp, char* format, ...)`	print formatted string to stream
`fscanf(FILE* fp, char* format, ...)`	get formatted string from stream
`sprintf(char* buf, char* format, ...)`	print formatted string to buffer
`sscanf(char* buf, char* format, ...)`	get formatted string from buffer

Example,

```
char* example = "frog, 123 12.4 c";
char animal[10]; int num; float fnum;
char c;
sscanf(example, "%9s, %d, %f, %c ", animal, &num, &fnum, &c);
/* use %9s to limit input to 9 chars + \0 */
```

Binary Operations

AND: `1010 & 1100` = `1000`

OR: `1010 | 1100` = `1110`

XOR: `1010 ^ 1100` = `0110`

Left shift: `1010 << 1` = `0100`

Right shift: `1010 >> 1` = `0101`

for shifts, with `signed` values, their sign is kept in arithmetic shift (where it is extended)

```
int a = -27; // 0b11100101
a >> 3; // 0b11111100 (-4 in decimal), sign bit is extended
```

NOT: `~1010` = `0101`

Bit Mask

Get the 3rd bit

```
int x = 0b01101010;
x >>= 2; // move 3rd bit to 0th position
x &= 0x01; // mask just the 1st bit, res: 0
```

Algorithms

Big **O** complexity, the worst-case performance of an algorithm as the input grows, e.g., $O(n^2)$

- Time complexity, how many operations it will take
- Space complexity, how much memory the algorithm will take

```
int calculateSum(int arr[], int N) {
  int total = 0;
  for (int i = 0; i < N; i++) {
    total += arr[i];
  }
  return total;
}
```

Complexity $T(N) = 3N + 3$ number of operations, reduce to most significant term N - which is the time complexity.

Operations on data: *access, insert, delete, traverse, search, sort.*

List

Can have an ordered list which drastically speeds up searching time.

Linked List

List of elements joined to one another.

normal

```
struct Node {
  int data;
  struct Node *next;
};
```

doubley

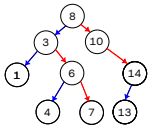
```
struct Node {
  int data;
  struct Node *prev;
  struct Node *next;
};
```

```
void insert_after(struct Node *node, struct Node *existing) {
  node->next = existing->next;
  existing->next = node;
}
```

```
void remove_after(struct Node *existing) {
  struct Node *toremove = existing->next;
  existing->next = toremove->next;
  toremove->next = NULL;
}
```

```
struct Node *node = head;
while (node->next != NULL) {
  node = node->next;
}
```

Binary Search Tree



Tree structure, where the larger element is always on the right and the smaller element on the left. Search in $O(\log N)$ and all keys are *unique*.

```
struct Node {
  int data;
  struct Node* left; // Node or NULL
  struct Node* right;
};
```

	Array	Linked List	BST
Search	$O(N)$	$O(N)$	$O(H)$
Insert	$O(N)$ $O(1)$ if end	$O(1)$	$O(H)$
Delete	$O(N)$ $O(1)$ if end	$O(1)$	$O(H)$

BSTs data is presorted, and H is $\log N$ if balanced N if not.

External Libraries

`stdio.h`	printf, scanf, sprintf, sscanf, fopen, fclose, fgets, getchar
`stdlib.h`	atoi, atof, malloc, free, qsort
`ctype.h`	isdigit, isalpha, isspace
`string.h`	strlen, strcpy, strcmp, strcat, strstr, strtok
`math.h`	sqrt, pow, sin, cos, tan, exp, log

Errors

`Syntax Error`	Grammatical rules violated
`Runtime Error`	Invalid operations i.e., $\frac{2}{0}$
`Linker Error`	Cannot find function
`Memory Error`	Underflow/overflow, use after free, double free

Good luck ☺

Data types

name	size (bytes)	min	max	min U	max U	postfix
`char`	1	−128	127	0	255	
`int`	4	−32768	32767	0	65535	
`float`	4	-3.4×10^{-38}	3.4×10^{38}	prec: rounding errors		`1.4f`
`double`	8					`3.14L` *long double*

Qualifiers

Anatomy: `%[flags][width][.precision][length]specifier`,

Flags: `-` left justify, `+` force sign

Width: width taken up characters `int a = 5; printf("%3d", a); // prints 005`, can use `*` for variable controlled.

Precision, `.number` minimum number of digits to be written, `float a = 5.12324; printf("%.1d", a); // prints 5.1`

specifier	description
`%c`	character
`%s`	string
`%d`/`%i`	integer

specifier	description
`%u`	*unsigned* integer
`%x`	hex int
`%f`	floating
`%lf`	long floating