**Learn C the hard way**

PART 1

Strings

* When working with strings, use the *char \*str = value* syntax, as opposed to *char str[] = value*.
* C treats strings as just arrays of bytes, and it’s only the different printing functions that recognize a difference.

// These 2 are the exact same:

Char name[] = “ali”;

Char name[] = {‘a’, ‘l’, ‘i’};

Numbers

Int areas[] = {10, 12, 13, 14, 20}; //sizeof(areas) = 20 (int = 4 bytes)

Int areas[10] = {10, 12, 13, 14, 20}; //sizeof(areas) = 40 (40 bytes allocated)

// In the latter example, there are technically 10 values in this array, index 5-9 will have the value 0. Accessing a variable outside this range will result in an error/unpredictable output.

Arrays

An array of strings is basically a multidimensional array:

Char \*names[] = {“ali”, “lauren”};

Char names[][10] = {“ali”, “lauren”};

Char names[][10] = {

{‘a’, ‘l’, ‘i’},

{‘l’, ‘a’, ‘u’, ‘r’, ‘e’, ‘n’}

};

Pointers

**Example 4.1**

Although similar, pointers are not arrays. For example (from example 4.1), sizeof(p\_ages) would give the size of the pointer, not the size of what it points at. If you want the size of the array, you would need to use the arrays name *ages.*

Structs

A structure in C is a collection of other data types that are stored in one block of memory where you can access each variable independently by name. **Example 5.1.** Use malloc to assign memory when using structs. IMPORTANT – always free up space once it is no longer needed!

**Example 5.2**

Stack & Heap memory allocation

**Heap**

All the remaining memory (RAM) in your computer. It can be accessed with malloc(). Each time you call malloc(), the OS uses internal functions to register that piece of memory to you, and then returns a pointer to it. Use free() to return it to the OS. Failing to free() will result in a memory leak.

**Stack**

Special region of memory that stores temporary variables, which each function creates as locals to that function. Each argument to a function is *push* on to the stack and then used inside the function. LIFO (last in first out).

If you get a block of memory from malloc() and have that pointer on the stack, then when the function exits, the pointer will get popped off and lost.

If you put too much data on the stack, then you can cause a stack overflow and the program will abort.

If you take a pointer to something on the stack, and then pass or return it from your function, then the function receiving it will *segmentation fault*, because the actual data will have got *popped off* and lost. You will be pointing to dead space.

Macro functions

**Example 6.1**

Other

|  |  |
| --- | --- |
| **Name** | **Description** |
| Forward declaration | Defining a function/blueprint before it is used (**example 7.1**) |
| Void perror(const char \*str) | Prints descriptive error message to stderr |
| Void rewind(FILE \*stream) | Sets the file position to beginning of given stream |
| Int atoi(const char \*str) | Converts string argument to integer. *Atoi(“123”); //123.* Returns 0 if no valid conversion. |

Examples

**4.1**

Int ages[] = {28, 29, 30, 24};

Int count = sizeof(ages) / sizeof(int);

Int i = 0;

// CASE 1

For (i = 0; i < count; i++) {

Printf(“%d\n”, ages[i]);

}

// CASE 2

Int \*p\_ages = ages; // no need for & when pointing to array

For (i = 0; i < count; i++) {

Printf(“%d\n”, \*(p\_ages + i));

}

// case 1 will also work here, e.g. p\_ages[i]

**5.1**

Struct Person {

Char \*name;

Int age;

Int height;

Int weight;

}

Struct Person \*Person\_create(char \*name, int age, int height, int weight) {

Struct Person \*who = malloc(sizeof(struct Person));

Assert(who != NULL); // <assert.h> - info written to stderr

Who->name = strdup(name); // strdup returns a ptr to a string

Who->age = age;

Who->height = height;

Who->weight = weight;

Return who;

}

Void Person\_destroy(struct Person \*who) {

Assert(who != NULL);

Free(who->name); // this was initialized with strdup() which returns ptr

Free(who);

}

Void Person\_print(struct Person \*who) {

Printf(“%s - %d yrs old - %dcm - %dkg\n”, who->name, who->age,

who->height, who->weight);

}

Int main(int argc, char \*argv[]) {

Struct Person \*ali = Person\_create(“Ali Issaee”, 28, 180, 80);

Person\_print(ali);

Printf(“%p memory location\n”, ali);

//overwrite

ali->age = 29;

Person\_print(ali);

Person\_destroy(ali);

Return 0;

}

**5.2**

Struct Address {

Int houseNumber;

Char \*streetName;

Char \*postcode;

};

Struct House {

Struct Address \*address;

Char \*brickColour;

};

Void print\_house(struct House \*home) {

Printf(“Brick colour: %s\nAddress:\n%d %s\n%s”, home->brickColour,

Home->address->houseNumber, home->address->streetName,

home->address->postcode);

}

Int main(int argc, char \*argv[]) {

Struct Address \*addr = malloc(sizeof(struct Address));

Address->streetName = strdup(“Fake road”);

Address->houseNumber = 1502;

Address->postcode = strdup(“HU53FD”);

Struct House \*home = malloc(sizeof(struct House));

Home->address = addr;

Home->brickColour = strdup(“red”);

Print\_house(home);

Free(address->streetName);

Free(address->postcode);

Free(address);

Free(home->address);

Free(home->brickColour);

Free(home);

}

**6.1**

#define test(msg) printf(“%s\n”, msg)

#define debug(M, …) fprintf(stderr, “DEBUG: %s:%d ” M “\n”, \_\_FILE\_\_, \_\_LINE\_\_\

##\_\_VA\_ARGS\_\_)

#define check(A, M, …) if(!(A)) {\

Printf(M, ##\_\_VA\_ARGS\_\_); errno=0; goto error;}

Int main(int argc, char \*argv[]) {

Char \*name = “type”

Check(NULL, “%s error”, name); // the M arg allows us to use function like printf() where you pass variables

Printf(“Should not see this…”);

Error: // goto in check() goes here

Printf(“Should see this”);

}

**7.1**

#include <stdio.h>

#include<stdlib.h>

#include<ctype.h>

//forward declarations

Int can\_print\_it(char ch);

Void print\_letters(char arg[]);

Void print\_arguments(int argc, char \*argv[]) {

Int i = 0;

For (i=0; i < argc; i++) {

Print\_letters(argv[i]);

}

}

Void print\_letters(char arg[]) {

Int i = 0;

For (i=0; arg[i] != ‘\0’; i++) {

Char ch = arg[i];

If(can\_print\_it(ch)) {

Printf(“%c == %d”, ch, ch);

}

}

Printf(“\n”);

}

Int can\_print\_it(char ch) {

Return isalpha(ch) || isblank(ch); // if alphabetical or empty space (ctype)

}

Int main(int argc, char \*argv[]) {

Print\_arguments(argc, argv);

Return 0;

}

// isalpha and isblank will return 0 if false.

// A char in C is also a number, which is the ASCII code equivalent, e.g. *char r = ‘r’; printf(“%c == %d”, r, r); // r == 114*

**Quiz**