Memory

Accessing memory locations and values at locations using C

Using `&` (the address operator) allows you to get the physical address of a variable in C. This is because all variables have a section of memory that it stores it's data. For example:

```c
#include <stdio.h>

int main(void){
    int n = 50;
    printf("%p\n", &n);
}

// Output: 0x7ff7bdfc4fdc
```

Using `*` (the dereference operator) allows you to get the value inside an address. For example:

```c
#include <stdio.h>

int main(){
    int n = 50;
    int *p_n = &n;
    printf("%p = %i\n", p_n, *p_n);
}

// Output: 0x7ff7bed6efdc = 50
```

As you can see, there is two different syntaxes for the `*`. `[type] *pointer_var` is the initalisation of a pointer. This is the address of a variable of a data type. On the other hand, `*pointer_var` is the syntax for dereferencing. This gives you the value at the address that is stored in `pointer_var`.

Strings

Chapter

Strings have been a white lie since week 1. We have been treating them as an `array` of `char`'s with a terminating null character `\0`.

A string, in C, is actually a pointer to the starting address of the string. So if you had the following:

Address	Value
0x123	H
0x124	I
0x125	!

Then the *"string"* variable would be a pointer to `0x123`. This therefore means, from our reasoning, that you can write a string like the following:

```
#include <stdio.h>

int main(){
    char* s = "HI!";
    printf("%s\n", s);
}

// Output: HI!
```

Or, without using the `%s` operator in `printf` you can do:

```
#include <stdio.h>

void print_string(char* s){
    while(*s != '\0'){
        printf("%c", *s);
        s++;
    }
    printf("\n");
}

int main(){
    char* s = "HI!";
    print_string(s);
}

// Output: HI!
```

Pointer Arithmetic

Chapter

Pointer arithmetic is used to be able to manipulate where a pointer is pointing. This was done in the previous example by using `s++` to incremement the pointer by one address value, i.e. `s` went from`0x123` to `0x124` by doing `s++`.

This can also be done without, changing the value of the pointer address, by adding an offset such as `*(s + n)` where `n` is the offset. Therefore you can do:

```c
#include <stdio.h>

void print(char* s);

int main(){
    char* s = "HI!";
    print(s);
}

void print(char* s){
    int offset = 0;
    while(*(s + offset) != '\0'){
        printf("%c", *(s + offset));
        offset++;
    }
    printf("\n");
}

// Output: HI!
```

This pointer arithmetic is actually how the array syntax `s[n]` works under the hood in the compiler.

String Comparison

Chapter

For integer comparsion we can do the following:

```
#include <stdio.h>

int main(){
    int a  = 1;
    int b = 2;

    if(a == b){
        printf("Same\n");
        return 0;
    } else {
        printf("Not same\n");
        return 1;
    }
}

// Output: Not same
// $? = 1
```

However for strings, it is a bit different as we need to compare two arrays of characters, and therefore can't just use the `==` operator. This is because if we use the `==` syntax, then it will be asking C *Is the address of* ***`str1`*** *the same as* ***`str2`*** because the values are of the addresses. Instead, we need to compare the value at each point in the string, this is what `string.h` does with it's `strcmp` funciton

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

int main(){
    char* a = "Hello";
    char* b = "world";

    if(strcmp(a, b) == 0){
        printf("Same\n");
        return 0;
    } else {
        printf("Not same\n");
        return 1;
    }
}

// Output: Not same
```

```
// $? = 1
```

`strcmp` also returns three values: - `0` if they're the same - `>0` if one comes before the other - `<0` if one comes after the other

You can therefore use strcmp to alphabetise an array of strings.

Copying

Chapter

The issue we now have is that if we want to take a string, copy it to a new variable, and manipulate it, it is going to change both strings. This is because if we have string `s` and say `char* t = s;` then we are saying *"set `t` to the address of `s`"*. If we then go on to say *"change the first character of `t` to be a 1"* then it is going to change the first character for `t` as well as `s`. This is because we have said *"take the first location of `t` (which is the same as `s`) and change it to 1"*.

```
#include <stdio.h>

int main(){
    char* s = "Hello";
    char* t = s;

    t[0] = '1';

    printf("%s, ", s);
    printf("%s\n", t);

    return 0;
}

// Output: 1ello, 1ello
```

> *It actually gives you a bus error because the string literal `char*` goes into a piece of read-only memory, and therefore when you go to change `t[0]` to something else, you can't rewrite the memory. However, theoretically, if it was in a piece of read-write memory, then this would be the case.*

The solution to this is to use **memory allocation** (`malloc`) and `__free__` which are found in `stdlib.h`

malloc

`malloc` is a way to get a chunk of free memory of n bytes, and returns the first location in memory. If `malloc` returns `NULL` then there isn't enough memory for the allocation, this can be checked in the program:

```
char* str_cpy = malloc(n);
if(str_cpy == NULL){
    return 1;
}
```

free

`free` is used to clean this memory after it has been used, as to not cause any memory leaks.

Therefore, we can do the following:

```
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(){
    char* s = "hi!";
    if(s == NULL) return 1;

    char* t = malloc(strlen(s) + 1); // we need each character + '\0'
    if(t == NULL) return 1;

    for(int i = 0, n = strlen(s); i <= n; i++){
        t[i] = s[i]; // Assign the characters from s to t
    }

    t[0] = toupper(t[0]); // Change character 0 to uppercase

    printf("%s, ", s);
    printf("%s\n", t);

    free(t);
    return 0;
}

// Output: hi!, Hi!
```

However, because string copying has been around for a while, there is a function in the `string.h` library called `strcpy` that has the parameters of destination and source. Therefore, the improved code is:

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

int main(){
    char* s = "hi!";
    if(s == NULL) return 1;

    char* t = malloc(strlen(s) + 1); // we need each character + '\0'
    if(t == NULL) return 1;
```

```
    strcpy(t, s) // Assign the string from s to t.

    t[0] = toupper(t[0]); // Change character 0 to uppercase

    printf("%s, ", s);
    printf("%s\n", t);

    free(t);
    return 0;
}

// Output: hi!, Hi!
```

So finally, what is `NULL`? Well, `NULL` is the address `0x0`. This is interesting, because `NUL` means `\0` for a useless character to define that this is the end, `NULL` is a useless address in memory to tell us that something went wrong.

Malloc and Valgrind

Chapter

Valgrind is a piece of software that is used to analyse what memory has been used in a compiled c program. This can be useful when debugging memory related issues. For example:

```c
#include <stdlib.h>

int main(){
    int* x = malloc(3 * sizeof(int));
    x[1] = 72;
    x[2] = 73;
    x[3] = 33;
}
```

In this example, we are trying to make an `int` array by hand by using `malloc`. We have given the array a length of 3 by using `3 * sizeof(int)` (`sizeof` returns the number of bytes of a type). We then assign `x[1]`, `x[2]`, and `x[3]` to values `72`, `73`, and `33`. This is buggy software.

We get the following output from `valgrind ./maclloc_and_valgrind_1.out`

```
==1== Memcheck, a memory error detector
==1== Copyright (C) 2002-2024, and GNU GPL'd, by Julian Seward et al.
==1== Using Valgrind-3.23.0 and LibVEX; rerun with -h for copyright info
==1== Command: ./build/malloc_and_valgrind_1.out
==1==
==1== Invalid write of size 4
==1==    at 0x1091BF: main (malloc_and_valgrind_1.c:7)
==1==  Address 0x48be04c is 0 bytes after a block of size 12 alloc'd
==1==    at 0x48AD733: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==1==    by 0x109196: main (malloc_and_valgrind_1.c:4)
==1==
==1==
==1== HEAP SUMMARY:
==1==     in use at exit: 12 bytes in 1 blocks
==1==   total heap usage: 1 allocs, 0 frees, 12 bytes allocated
==1==
==1== 12 bytes in 1 blocks are definitely lost in loss record 1 of 1
==1==    at 0x48AD733: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==1==    by 0x109196: main (malloc_and_valgrind_1.c:4)
==1==
==1== LEAK SUMMARY:
==1==    definitely lost: 12 bytes in 1 blocks
==1==    indirectly lost: 0 bytes in 0 blocks
==1==      possibly lost: 0 bytes in 0 blocks
```

```
==1==    still reachable: 0 bytes in 0 blocks
==1==         suppressed: 0 bytes in 0 blocks
==1==
==1== For lists of detected and suppressed errors, rerun with: -s
==1== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

The first error we see is *"Invalid write of size 4" "at 0x1091BF: main (malloc_and_valgrind_1.c:7)"*. Line 7 is `x[3] = 33;`. The bug on this line is that we are trying to initalise the fourth element in the array, depsite the array only being a size of three. Therefore we need to change all the indices from lines 5 to 7 to be correct (starting at 0):

```c
#include <stdlib.h>

int main(){
    int* x = malloc(3 * sizeof(int));
    x[0] = 72;
    x[1] = 73;
    x[2] = 33;
}
```

We get the following output from `valgrind ./malloc_and_valgrind_2.out`

```
==1== Memcheck, a memory error detector
==1== Copyright (C) 2002-2024, and GNU GPL'd, by Julian Seward et al.
==1== Using Valgrind-3.23.0 and LibVEX; rerun with -h for copyright info
==1== Command: ./build/malloc_and_valgrind_2.out
==1==
==1==
==1== HEAP SUMMARY:
==1==     in use at exit: 12 bytes in 1 blocks
==1==   total heap usage: 1 allocs, 0 frees, 12 bytes allocated
==1==
==1== 12 bytes in 1 blocks are definitely lost in loss record 1 of 1
==1==    at 0x48AD733: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==1==    by 0x109196: main (malloc_and_valgrind_2.c:4)
==1==
==1== LEAK SUMMARY:
==1==    definitely lost: 12 bytes in 1 blocks
==1==    indirectly lost: 0 bytes in 0 blocks
==1==      possibly lost: 0 bytes in 0 blocks
==1==    still reachable: 0 bytes in 0 blocks
==1==         suppressed: 0 bytes in 0 blocks
==1==
==1== For lists of detected and suppressed errors, rerun with: -s
==1== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

We now have a new error *"12 bytes in 1 blocks are definitely lost in loss record 1* *of 1" "by 0x109196: main (malloc_and_valgrind_2.c:4)"*. Line 4 is `int* x = malloc(3* sizeof(int));`. The bug is that we didn't free the memory, hence the *"12 bytes definitely lost..."* in the error log. To fix this, we can use the `free` keyword to free the memory after usage:

```c
#include <stdlib.h>

int main(){
    int* x = malloc(3 * sizeof(int));

    x[0] = 72;
    x[1] = 73;
    x[2] = 33;

    free(x);
}
```

We get the following output from `valgrind ./malloc_and_valgrind_3.out`

```
==1== Memcheck, a memory error detector
==1== Copyright (C) 2002-2024, and GNU GPL'd, by Julian Seward et al.
==1== Using Valgrind-3.23.0 and LibVEX; rerun with -h for copyright info
==1== Command: ./build/malloc_and_valgrind_3.out
==1==
==1==
==1== HEAP SUMMARY:
==1==     in use at exit: 0 bytes in 0 blocks
==1==   total heap usage: 1 allocs, 1 frees, 12 bytes allocated
==1==
==1== All heap blocks were freed -- no leaks are possible
==1==
==1== For lists of detected and suppressed errors, rerun with: -s
==1== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

The line *"All heap blocks were freed – no leaks are possible"* show that everything is now memory safe in the program.

Garbage Values

Chapter

Garbage values are *"values of variables that you did not proactively set yourself as intended"*

A good example of garbage values, is the output of this program:

```c
#include <stdio.h>

int main(){
    int size = 1024;
    int scores[size];
    int garbage_count = 0;

    for(int i = 0; i < size; i++){
        if(scores[i] != 0){
            garbage_count++;
        }
    }

    printf("Garbage Values Count: %i\n", garbage_count);
    return 0;
}

// Output: Garbage Values Count: 715
```

This shows that when we initalised `scores` as an `array` of `int`'s with a size of 1024, it had 715 values that weren't initalised to 0 (**70%**). This therefore means that we can never trust the values of an uninitalised piece of memory as it can contain garbage values.

Swapping

Chapter

Naive way

```c
#include <stdio.h>

void swap(int a, int b);

int main(){
    int x = 1;
    int y = 2;

    printf("x: %i, y: %i | ", x, y);
    swap(x, y);
    printf("x: %i, y: %i\n", x, y);
}

void swap(int a, int b){
    int temp = a;
    a = b;
    b = temp;
}

// Output: x: 1, y: 2 | x: 1, y: 2
```

Pointer Solution

```c
#include <stdio.h>

void swap(int* a, int* b);

int main(){
    int x = 1;
    int y = 2;

    printf("x: %i, y: %i | ", x, y);
    swap(&x, &y);
    printf("x: %i, y: %i\n", x, y);

}

void swap(int* a, int* b){
    int temp = *a;
    *a = *b;
```

```
    *b = temp;
}

// Output: x: 1, y: 2 | x: 2, y: 1
```

So why does this work? Well previously we have used *"Pass by value"* which means that the function will get a copy of the values and then, if nothing returns, will forget the values once the function is complete. This can be fixed by using a technique called *"pass by reference"* where you pass the memory locations instead of a copy of the values. Therefore, this function now swaps the values at the memory locations of the variables, instead of swapping the values internally in the function.

scanf

Chapter

`scanf` is in the `stdio.h` library and is used to get the characters from `stdin`. This is what `get_int()` in the `cs50.h` library does under the hood. For example, the equivelant of `get_int()` does this:

```c
#include <stdio.h>

int get_int(char* message);

int main(){
    int n = get_int("n: ");
    printf("n: %i\n", n);

    return 0;
}

int get_int(char* message){
    printf("%s", message);

    int number;
    scanf("%i", &number);
    return number;
}

// Output:
// n: {user input}
// n: {user input}
```

Basically, `scanf` takes a string literal for what to expect as input (similar to how `printf` has a string literal to specify the types on output) and then takes a memory address to store each value that you recieve from `scanf`. For example, if you want to store an integer and a character (i.e. 1a), then you can do:

```c
#include <stdio.h>

int main(){
    int i;
    char c;

    printf("code: ");
    scanf("%i%c", &i, &c);
    printf("code: %i%c\n", i, c);

    return 0;
}
```

```c
// Output:
// code: {user input}
// code: {user input}
```

This is quite dificult to do with strings however, as you don't know the size of the string before you start, therefore you have to allocate and guess:

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

#define MAX 16

int main(){
    char* s = malloc(MAX);

    printf("s: ");
    scanf("%s", s);
    printf("s: %s\n", s);

    free(s);

    return 0;
}

// Output:
// s: {user input}
// s: {user input}
```

This program will most likely work, because `printf` will keep on printing the characters until a `NUL` byte is found, however if this was used in a bigger project, the overflow of data can be rewritten to something else, therefore leading to undefined behaviour.

File I/O

Chapter

The most common functions related to file, includes: - `fopen`, opens a file - `fclose`, closes a file - `fprintf`, allows you to print/write to a file - `fscanf`, allows you to read data from a file rather than the keyboard - `fread`, allows you to read binary data from a file - `fwrite`, allows you to write binary data to a file - `fseek`, allows you to move back and forwards in a file.

For example:

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

#define MAX 16

char* get_string(char* msg);

int main(){
    FILE* file = fopen("phonebook.csv", "a"); // Open file phonebook.csv in append mode, re

    char* name = get_string("Name: ");
    char* number = get_string("Number: ");

    fprintf(file, "%s,%s\n", name, number); // Append name and number in csv file formattin

    fclose(file); // Close file

    printf("Added contact %s @ %s\n", name, number);

    free(name);
    free(number);
    return 0;
}

char* get_string(char* msg){
    printf("%s", msg);
    char* input = malloc(MAX);
    scanf("%s", input);
    return input;
}
```

This creates a phonebook.csv file that will save names to numbers. Therefore if I run this with inputs *john_smith* and *07123456789* as well as *jane_doe* and *07234567891* then I will get a file called *phonebook.csv* with the contents:

```
john_smith,07123456789
```

```
jane_doe,07234567891
```

name	number
john_smith	07123456789
jane_doe	07234567891

Obviously, checks need to be made for if pointers return `NULL`, so an updated c file would be:

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

#define MAX 16

char* get_string(char* msg);

int main(){
    FILE* file = fopen("phonebook.csv", "a");
    if(file == NULL) return 1;

    char* name = get_string("Name: ");
    if(name == NULL) return 1;

    char* number = get_string("Number: ");
    if(number == NULL) return 1;

    fprintf(file, "%s,%s\n", name, number);

    fclose(file);

    printf("Added contact %s @ %s\n", name, number);

    free(name);
    free(number);
    return 0;
}

char* get_string(char* msg){
    printf("%s", msg);
    char* input = malloc(MAX);
    scanf("%s", input);
    return input;
}
```

This will now double check for invalid pointers for strings as well as the file.

An example to copy the functionality of `cp` in linux would be:

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

typedef uint8_t BYTE;

int main(int argc, char* argv[]){
    if(argc != 3){
        printf("Usage: cp src dst");
        return 1;
    }

    FILE* src = fopen(argv[1], "rb");
    if(src == NULL){
        printf("%s not found", argv[1]);
        return 1
    }

    FILE* dst = fopen(argv[2], "wb");
    if(dst == NULL){
        printf("%s not found", argv[2]);
        return 1
    }

    BYTE buffer;

    while(fread(&buffer, sizeof(buffer), 1, src) != 0){ // While the number of bytes we read
        fwrite(&buffer, sizeof(buffer), 1, dst);        // Write the byte in the buffer to
    }

    fclose(dst);
    fclose(src);
}
```

This is quite slow, as we are only doing a byte at a time. This can be improved by increasing the size of the buffer.