day-2-types-memory-files

September 6, 2023

1 Day 2: Variables and memory management

1.0.1 Topics

- types & memory
  - arrays (1d & 2d)
  - pointers
  - structs
  - casting & typedefs
- memory management
- (advanced) file input & output

Note: some exercises require user input e.g. using `scanf`. When this is the case, you should add ``#define _GNU_SOURCE'' at the top of your code cell (before ``#include <stdio.h>'') to ensure that interactive input works correctly in these notebook, like so:

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

This is a particular requirement for interactive input when using these notebooks. It is **not** normally required when using `scanf`.

1.0.2 Types & memory

1. Write a short C program that creates a `float x` with the value 19.1. Create another variable `y` which references (points to) `x`. What are the sizes of `x` and `y`? Are these sizes related? Introduce `double z = 48.2` and point `y` at it. Does this work? Why/why not? What is the size of `z` and a pointer to it?

[ ]:
```c
//// types-1.c

int main(void) {
    return 0;
}
```

2. Write a short C program that assigns `unsigned int a = 1` and prints the value of `a` assuming different types e.g. assuming `float`, `double`, `(signed) int`, etc. What does your program

print and why? How would you print the memory address of `a`? What if we instead assign `unsigned int a = -1`, will the code compile? What would the printed outputs be?

[ ]:
```
//// types-2.c

int main(void) {
    return 0;
}
```

3. As mentioned in the lectures, the boolean type is in the `stdbool.h` header, rather than being a built-in type, like `int`. It is possible to use booleans in C without `#include <stdbool.h>` by employing macros. In this exercise, define a minimal `bool` type using the `#define` directive, initialize a boolean, and use it in a conditional which prints its size when the value is `true`. What is the minimal size of a boolean, in bits? How large are the booleans in your macro definition? Write a second program which uses `#include <stdbool.h>`, how large are the booleans defined here?

[ ]:
```
//// types-3.c

int main(void) {
    return 0;
}
```

4. In the short C program `pointing.c`, we declare variables `double x, v, a, t, dt` and initialize them with appropriate values. Add to this program pointer variables to `x, v, a, t` and use these to print the updates in the `while` loop.

[ ]:
```
//// pointing.c

#include <stdio.h>

int main(){
   double x = 4.0, v = 0.0, a = -0.6, t = 0.0, dt = 0.1;
   while (x > 0.0){
      t += dt;
      x += dt*v;
      v += dt*a;
      a += 0.0*dt;
      printf("Ball:\tt = %f, x = %f, v=%f, a = %f.\n", t, x, v, a);
   }
}
```

5. Write a short C program which declares an array `int a[10] = {2, 5, 8, 11, 14, 17, 20, 23, 26, 29}` and then declares an uninitialized array `int b[10]`. Inside the loop, put the i-th element of `a` *squared* into the i-th element of `b`.

[ ]:
```
//// types-5.c

int main(void) {
    // declare a & b
    for (int i=0; i<10; i++) {
        // your code here
        b[i] = ...
    }
    return 0;
}
```

6. Write a short program which takes in a string from the user, and encodes it by shifting the characters comprising the string by a fixed value, and then prints this scrambled string. Write a second short program which takes in the scrambled string from the first program as user input, and decodes it by shifting the characters comprising the string, and then prints the original string. You will want to use the function `fgets` in `#include <stdio.h>`. How else might one encode and decode the input (i.e. not using a fixed offset)? How might these pairs of programs fail? What sort of restrictions does the consumption of the original string in the encoder and decoder programs impose on the encoding and decoding patterns?

Note: as this program requires input from the user, you should add `#define _GNU_SOURCE` at the top of your file to ensure that interactive input works correctly in this notebook.

Note: To prevent a code cell from automatically compiling or running, use the `//% NOCOMPILE` and `//% NOEXEC` magic comments.

[ ]:
```
//// types-6-encode.c

// this is needed *before* stdio.h for user
// input to work interactively in this notebook:
#define _GNU_SOURCE
#include <stdio.h>

int main(void) {
    return 0;
}
```

[ ]:
```
//// types-6-decode.c

// this is needed *before* stdio.h for user
// input to work interactively in this notebook:
#define _GNU_SOURCE
#include <stdio.h>

int main(void) {
    return 0;
}
```

7. Write a short C program that asks the user for a `name`, a `date` (e.g. their birthday), and a `number` (e.g. a phone number), places the data into an appropriately defined `struct`, and then prints the `struct` data legibly. Practice good security culture: **do not use your own birthday and phone number for this task**. Modify this program so that it accepts the

inputs from the command line instead of interactively.

[ ]:
```c
//// types-7-person.c

int main(void) {
    return 0;
}
```

8. Write a short program which assigns a single `const` variable identifier no less than six times, at least once at the global scope, with different values and then on the last line of `main` prints the value assigned on the first line of `main`.

[1]:
```c
//// types-8-const-scoping.c

const int num = 1; // the first instance of the const int num.
int main(void) {
    return 0;
}
```

```
wrote file types-8-const-scoping.c
$> gcc    types-8-const-scoping.c -o types-8-const-scoping
$> ./types-8-const-scoping
```

1.0.3 Memory management

1. Multidimensional C arrays are uniform depth, meaning if we allocate, `int arr[10][10]` then each column of `arr` has length 10. You should know that multidimensional arrays in C are just linear arrays with a fixed stride. You may be familiar with nested lists in Python, e.g. `arr = [[1, 2, 3], [4, 5], [6]]`, where each sub-list has its own length, which works because the length of the list is part of the type data in Python. These data structures are sometimes called ``jagged arrays''. Try designing a *static* jagged array in C and use it to reproduce the Python list above (assume each sublist is a row of jagged array). Iterate over the rows and print the values followed by a new line. Where in memory is this array allocated (stack or heap)?

[ ]:
```c
//// memory-1-jagged.c

int main(void) {
    return 0;
}
```

2. Write a function which takes an integer input from the user (or via the command line) defining the upper index limit n. Create two arrays, one on the stack and one on the heap, and iterate in a loop over the indices, assigning the i-th index to i, in both. Loop over the elements of the arrays and print the memory address of each element. Ensure your program doesn't have any memory leaks by compiling it with the flags `-Wall -fsanitize=leak`.

[ ]:
```
//// memory-2-user-defined-arrays.c
//% CFLAGS -Wall -fsanitize=leak

#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {

    // array size
    int n;

    for (int i=0; i<n; i++) {
        static_array[i] = ...
        dynamic_array[i] = ...
    }

    // remember to clean up

    return 0;
}
```

1.0.4 File input & output

For reference information on file input/output, see: File input/output

1. What functions are used for opening/closing files? How can you check whether opening a file was successful? What does the file access mode `"w"` mean? Does it indicate binary access or text access?
2. Write a C program which accepts a filename as a command-line argument and checks whether the file can be opened. If the file can be opened, it should print a message to `stdout`, and to `stderr` if the file can't be opened. It should exit with an error (and a helpful message) if the incorrect number of arguments are given. Remember to clean up at the end of the program.

[ ]:
```
//// file-in-out.c

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

    if (/* incorrect number of arguments */) {
        return 1;
    }

    if (/* unable to open the requested file */) {
        // ...
    } else { // the file was opened ok
        // ...
    }

    return 0;
}
```

3. Modify the program above so that it accepts an arbitrary number of file names as command-line arguments. For each file name, it should check whether the file can be opened. If the file can't be opened, it should print an appropriate message to `stderr`. If it can be opened, it should write a single line to a file `results.txt` with an appropriate message.

[ ]:
```
//// file-in-out-multi.c
//% ARGS foo bar baz

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

    if (/* incorrect number of arguments */) {
        return 1;
    }

    int k = 1; // index argv
    while (/* we have another argument to process */) {
        // ...
        k++; // move to next argument in argv
    }
    return 0;
}
```