DA4002 - Algorithms, Data Structures and Problem Solving

-----------------------------------------------------------------------------------------------------------------

**Lab3 - Group 1**

-----------------------------------------------------------------------------------------------------------------

Eric Heindl

Esteban Rueda

1. **Arrays - prime.c**
   1. *This program implements a classical algorithm for computing prime numbers called Sieve of Eratosthenes. That Wikipedia page contains a nice animation explaining how it works. Can you explain why the above implementation sometimes adds or removes 1 or 2 to the indices and output*

If we want to analyze up to 100 to collect the prime numbers, we need an array of 100 - 1.

Because we start at “2” ( since 1 can’t be a prime number ), then we need to substract “1” to the MAXNUM “100” to get the 100 numbers.

We add and substract +2 & -2 for the conversion from address value to number value. i.e: if array[0] is number 2, the address number is +2.

If we want to store something for number 2, then we need to remove 2 to get the array location.

* 1. *Modify the program to produce a histogram of the number of integer divisors except 1 and itself, for every number from 2 to 20. The output of the program should be:*

#include <stdio.h>

#define MAXNUM 100

int main (int argc, char \*\* argv)

{

char is\_prime[MAXNUM - 1];

int ii, num;

for (ii = 0; ii < MAXNUM - 1; ++ii) {

is\_prime[ii] = 1;

}

for (num = 2; num <= MAXNUM / 2; ++num) {

for (ii = num \* 2 - 2; ii <= MAXNUM - 2; ii += num) {

**is\_prime[ii]++;**

}

}

printf ("prime numbers up to %d:\n ", MAXNUM);

for (ii = 0; ii < MAXNUM - 1; ++ii) {

if (is\_prime[ii]==1) {

printf (" %d \n", ii + 2);

}

**else {**

**printf (" %d", ii + 2);**

**for(int i =1; i< (int)is\_prime[ii]; i++){**

**printf("\*");**

**}**

**printf("\n");**

**}**

**}**

**printf ("\n");**

return 0;

}

**Output**:

prime numbers up to 20:

2

3

4\*

5

6\*\*

7

8\*\*

9\*

10\*\*

11

12\*\*\*\*

13

14\*\*

15\*\*

16\*\*\*

17

18\*\*\*\*

19

20\*\*\*\*

1. **a first look at pointers - pointers\_arrays\_strings.c**
   1. *The annotated source code pointers-arrays-strings.c reviews some of the explanations about pointers and arrays that were part of lecture 1. Compile and run pointers-arraysstrings and make sure you understand how the output relates to the source code.*

Let's start with two integers:

ix = 99 and

iy = 333

ix resides at the address 000000000062FE34

iy resides at the address 000000000062FE30

In the source code we are storing the address of each integer into the variable “iptr” and printing it.

Assigning to ix simply changes ITS value.

before: the value of ix is 99

after: the value of ix is 999

In the source code we are just assigning a value to the same variable and printing it twice, once with each value, so only the value is changed.

Assigning to \*iptr changes the value stored AT THAT ADDRESS.

before: the value at 000000000062FE30 is 333

after: the value at 000000000062FE30 is -1111

In the source code it just gives the address a different value, so the new address has the new value without changing the address.

iptr was pointing to iy when we did that, see:

iy is now -1111

The assignment did not change iptr, it changed iy!

The program looks for the value of iy which is stored in iptr, and the value it finds is -1111.

Let's look at arrays next.

An array is a block of values of identical type.

Its elements are accessed using an offset within brackets.

ia contains three values:

ia[0] = 100

ia[1] = 101

ia[2] = 102

An array variable simply points to the beginning of the block.

the array ia starts at address 000000000062FE20

To get at an element, you add offset \* typesize to the beginning.

the value at 000000000062FE20 is 100

the value at 000000000062FE24 is 101

the value at 000000000062FE28 is 102

Each element consists of 4 bytes of space, so , every time we add 1 to the value , the address adds 4 bytes.

The C compiler USUALLY multiplies by the typesize automatically.

the difference between the addresses printed above

is the same as sizeof(int), namely zu

Let's try with an array of doubles, and a pointer to double:

the value at 000000000062FE10 is 42.170000

the value at 000000000062FE18 is -0.000001

In the arrays of doubles, each element is 8 bytes.

the difference between those addresses is zu

now, dptr points to da[1] so we can change that via \*dptr...

the value of da[1] is now -999.888000

Let's look at character strings.

In C, strings are arrays of char which store ASCII codes.

The end of strings are signalled by placing a zero into the array

the string at 0000000000405527 is "hello"

the ASCII code at 0000000000405527 is 104 (hex 68)

the ASCII code at 0000000000405528 is 101 (hex 65)

the ASCII code at 0000000000405529 is 108 (hex 6c)

the ASCII code at 000000000040552A is 108 (hex 6c)

the ASCII code at 000000000040552B is 111 (hex 6f)

the ASCII code at 000000000040552C is 0 (hex 00)

Each letter/character of the word “hello” has its own address, each one of them taking 1 byte.

You can manually assemble strings, too...

the result is: Byebye!

The code writes the ASCII code of each letter and then prints the array, the result is as we can see above, “Byebye!” , the last element of the array is a 0 for the program to know its finished.

1. **Command line arguments - printargs.c**

to print a **size\_t** variables use **%lu** not %zu

* 1. *Copy the above sourcecode into a file called printargs.c, compile it, and run it. It should produce the following (or very similar) output:*

**Our Output:**

argv[0] (length 42): `C:\Users\Eric\Downloads\Lab2\printargs.exe'

* 1. *Change the program arguments to the following using the Eclipse Run Configuration settings:*

we did it via the call function like:

.\printargs some arguments and some spaces

* 1. *Try some more complicated looking arguments:*

.\printargs bravo \\ "" "something with spaces"

**Output:**

argv[0] (length 42): `C:\Users\Eric\Downloads\Lab2\printargs.exe'

argv[1] (length 5): `bravo'

argv[2] (length 2): `\\'

argv[3] (length 21): `something with spaces'

* 1. *Set the arguments so that the output looks like this:*

.\printargs.exe "'"

**Output:**

argv[0] (length 42): `C:\Users\Eric\Downloads\Lab2\printargs.exe'

argv[1] (length 1): `''

1. **strings are character arrays - eascii.c**
   1. *Write a program called eascii that takes a string as command-line argument and prints a table with three lines:*

#include <stdio.h>

#include <string.h>

int main (int argc, char \*\* argv)

{

int ii;

for (ii = 1; ii < argc; ++ii)

{

size\_t len = strlen (argv[ii]);

for( int i=0; i<len; i++){

printf(" '%c'",argv[ii][i]);

}

printf("\n");

for( int i=0; i<len; i++){

if(argv[ii][i]<100){printf(" ");}

printf("%d ",argv[ii][i]);

}

printf("\n");

for( int i=0; i<len; i++){

printf(" %x ",argv[ii][i]);

}

}

return 0;

}

**Output**:

.\eascii.exe Coding\_is\_cool



1. **arrays and for loops - args-bubble-sort.c**
   1. *Based on this example implementation of bubble-sort and what you’ve learned above about command line arguments, write a program called args-bubble-sort that takes each argument string and sorts it character-by-character.*

*For example, if the argument is “743 oiufObIwUWBF”, the output of the program should be:*

#include <stdio.h>

#include <string.h>

int main (int argc, char \*\* argv)

{

int ii, jj, length;

**char tmp;** //we sort char with its ASCII-Code

for (ii = 1; ii < argc; ++ii)//for each argument ex. the first one

{

size\_t len = strlen (argv[ii]);

**printf("\n\nIN:"); //print the Input argument**

**for (int i = 0; i < len; ++i) {**

**printf ("%c", argv[ii][i]);**

**}**

**int sorted = 1; // sorted=1 not sorted**

**while(sorted){ // as long as sorted =1 ( not sorted)**

**sorted=0;**

for (int i = 0; i < len; i++) {

if (argv[ii][i] < argv[ii][i - 1]) {

tmp = argv[ii][i];

argv[ii][i] = argv[ii][i - 1];

argv[ii][i - 1] = tmp;

**sorted=1;**

}

}

}

**printf("\nOUT:"); // print sorted Output**

**for (int i = 0; i < len; ++i) {**

**printf ("%c", argv[ii][i]);**

**}**

}

return 0;

}

**Output**:

./args-bubble-sort.exe 743 oiufObIwUWBF

IN:743

OUT:347

IN:oiufObIwUWBF

OUT:BFIOUWbfiouw