Laboratory 2 tasks

Alexander Lundqvist

**For compiling and installing on chipKit**

make clean

make

make install TTYDEV=/dev/tty.usbserial-A503WFGA

**Task 1**

**Task 2**

*What does it mean when a function does not return a value? How do you state that in a*

*program? How can then the function (or more precisely, the procedure) perform anything*

*useful?*

*How did you implement the side effect that is needed to make print\_number behave*

*correctly?*

A program that doesn’t return a value (void) means that that function does not return a value to the caller explicitly.

In assembly this would translate as not storing any value in the $v-registers.

In C we define this by declaring the function type void.

Void function(parameter){do something}

A function doesn’t necessarily need to return something to be useful. For example, we could have a function swap, that takes in two addresses and then perform value swaps with pointers. This operation performs its task in memory and it certainly does SOMETHING, but it doesn’t return any value.

To implement the print\_number function, we used a global count variable. This is most of the times considered bad practice, but for the sake of keeping the function somewhat independent, we used it.

Text

Description automatically generated

**Task 3**

*How did you represent the marking of 'prime' and 'not a prime' in the memory array?*

*Which are the main steps in the algorithm? How have you implemented these steps?*

*What is the largest prime number that you can print within 2 seconds of computation? What*

*is the largest number you can print within 10 seconds? Is it the same for print\_prime.c,*

*sieves.c, and sieves-heap.c? Why or why not?*

To represent the numbers, we use an array populated with 1’s (true) and 0’s (false).

We use this to determine if a number that corresponds to the index, is a prime or not.

The algorithm is as follows:

Text

Description automatically generated

We start by initializing an array, then we populate it with 1’s with a loop (this could be 0’s also).

The main part of the algorithm is as follows:

* We start by checking the entries from index 2 (since we ignore 0 and 1).
* We then check if the “number” at the index is true (This will run regardless in first iteration)
* If the number is a prime, we then start to mark all multiples of that number as non-prime.
* Repeat for next entry (i++) up until the square root of the input. This is because the for loop would not be able to carry on with values higher than the square root of the input. All non primes will be taken care of with the prime multiples that are below the square root.

The printing part of the algorithm is simple. It starts at index 2, since 2 is the first prime. It then checks every number at index i. If it is 1, then call print\_number(i).

To test the different implementations, we can use functions included in time.h to determine the execution time of the different print functions.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **print\_prime.c** | **sieves.c** | **sieves-heap.c** |
| **2 seconds** |  |  |  |
| **10 seconds** |  |  |  |

*This was performed on a*

**Task 4**

*Explain how you get the pointer addresses to the two char arrays (text1 and text2) and*

*the counter variable (count) in function work().*

When you pass an array, you actually pass a pointer to the first element in the array. This is why we can just pass the arrays text1 and text2 without having to use the adress symbol.

To pass the counter variable, we use the & symbol.

*What does it mean to increment a pointer? What is the difference between incrementing the*

*pointer that points to the ASCII text string, and incrementing the pointer that points to the*

*integer array? In what way is the assembler code and the C code different?*

To increment a pointer means that the pointer will refer to the next value in memory. Different data types will have different size of pointer increment. So in assembly we will have to specify explicitly that the pointer have to increase by 4 bytes, since list is an int array, and text is a char array -> increment by only 1 byte.

In C we increment the pointers by 1, since we can define what type the pointer is from the start.

*What is the difference between incrementing a pointer and incrementing a variable that a*

*pointer points to? Explain how your code is incrementing the count variable.*

Incrementing a pointer means that it will refer to the following value in memory. Incrementing a variable will just change the value that the pointer refers to.

In our code, we use the following statement to increment the counter:

(\*count)++;

Order of operation for dereferencing and incrementing are the same,

therefore we have to put count in parentheses first.

*Explain a statement in your code where you are dereferencing a pointer. What does this*

Text

Description automatically generated*mean? Explain by comparing with the corresponding assembler code.*

To perform the copy, we have to dereference the pointers. This means that we perform action on the values in memory that each pointer refers to.

*“Dereferencing is used to access or manipulate data contained in memory location pointed to by a pointer. \*(asterisk) is used with pointer variable when dereferencing the pointer variable, it refers to variable being pointed, so this is called dereferencing of pointers.”*

*Is your computer using big-endian or little-endian? How did you come to your conclusion?*

*Is there any benefit of using either of the two alternatives?*

**Big Endian Byte Order:** The most significant byte (the "big end") of the data is placed at the byte with the lowest address. The rest of the data is placed in order in the next three bytes in memory.

**Little Endian Byte Order:** The least significant byte (the "little end") of the data is placed at the byte with the lowest address. The rest of the data is placed in order in the next three bytes in memory

**The advantages of Little Endian are:**

It’s easy to read the value in a variety of type sizes. For example, the variable A = 0x13 in 64-bit value in memory at the address B will be 1300 0000 0000 0000. A will always be read as 19 regardless of using 8, 16, 32, 64-bit reads. By contrast, in Big Endian we have to know in which size we have written the value to read it correctly.

**It’s easy to cast the value to a smaller type like from int16\_t to int8\_t since int8\_t is the byte at the beginning of int16\_t.**

Easily to do mathematical computations “because of the 1:1 relationship between address offset and byte number (offset 0 is byte 0), multiple precision math routines are correspondingly easy to write.”

**Some main advantages of Big Endian are:**

We can always test whether the number is positive or negative by looking at the byte at offset zero, so it’s easy to do a comparison.

**The numbers are also stored in the order in which they are printed out, so binary to decimal routines are particularly efficient.**

*“Big endian*

*+ behöver inte översättas om man ska skicka data via nätverk - använder generellt big endian*

*+ lättläst, "som vi förväntar oss"*

*ex. ettusen tvåhundra trettiofyra (1234)*

*minne lo -> hi: 1234*

*- leder till fet förvirring om man lärt sig att man räknar "höga värden till vänster"*

*bit 31 -> bit 0 (som man gör om man räknar binärt, OCH decimalt, coolt!)*

*1101 == 8+4+1, inte 1+2+8*

*Little endian*

*+ blir alltid samma värde oavsett om man läser 1, 2, 4, 8, 32, 64... bytes*

*0x41 == 0x0041 == 0x000041*

*-> leder till att man lätt kan casta värden mellan storlekar.”*

**Diagram

Description automatically generated**

**Endian experiment: 0x23,0x00,0x00,0x00**

When we start from a lower byte and increment, we would expect a big-endian system to print

**Task 5**

*Before the oral exam, you should prepare the answers to the following questions. You will need to*

*be able to answer these questions to pass the assignment.*

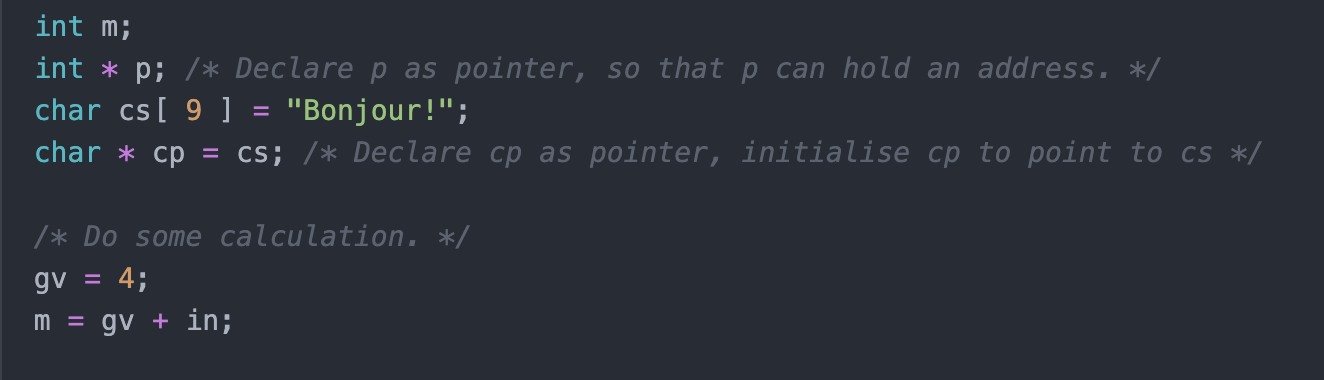
*1. Consider AM18, AM19, and AF1. Explain why gv ends up with the incremented value, but*

*m does not.*

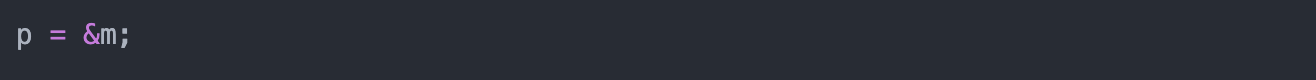
AM18 (m)

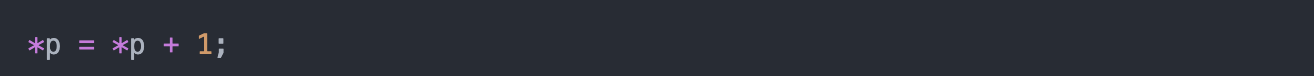
AM19 (gv)

AF1 (param)

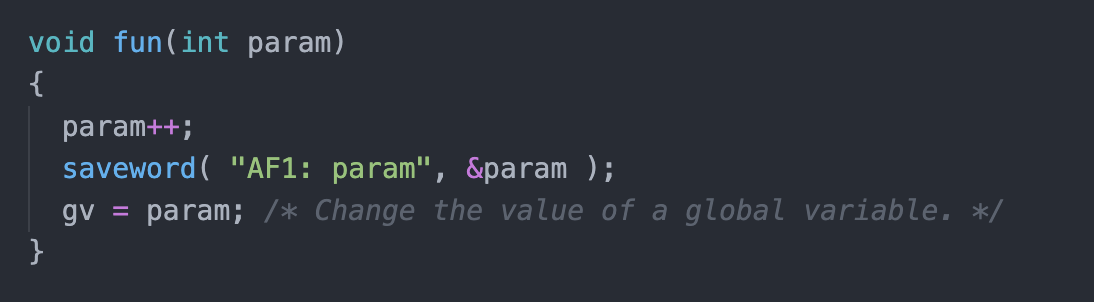


We first initialize some variables. Gv gets the value 4, and m gets gv + in (Which is a global variable initialized to 3.) -> m has the value 4+3 =7.

We then state that the pointer p holds the same address as m. This means that we can manipulate the value in memory with both p and m, since they both point to the same address.



We then dereference p and increment the value that p points to, which in turn means that when we dereference m, we will get that updated value. P&m will refer to the value 8.

We then call the function fun with m as argument.

What this does is it takes the param argument (m) and increases it with 1. Then saves this value at the address &param. We then declare that gv will have this new value (9).

Since we don’t modify m in memory, it will not get the updated value.

*2. Pointer cp is a character pointer that points to a sequence of bytes. What is the size of the*

*cp pointer itself?*

4 Bytes. A pointer refers to an address and the addresses have the format 0xnnnnnnnn, which is 32 bits.

*3. Explain how a C string is laid out in memory. Why does the character string that cp points*

*to have to be 9 bytes?*

A string in C is actually an char array with a NUL byte.

To signal that a string has ended, we need a NUL byte, hence we have 9 instead of 8.

*4. Which addresses have fun and main? Which sections are they located in? What kind of*

*memory are they stored in? What is the meaning of the data that these symbols points to?*

Fun:

Main:

*Before the examination, you should also try to answer the following. When the lab-assistant*

*performs the examination, he/she can also clarify anything that you did not understand with the*

*following questions:*

*5. Which addresses are variables in and gv located at? Which memory sections according to*

*the PIC32 memory map? Why?*

*6. Variables p and m are not global variables. Where are they allocated? Which memory*

*section is used for these variables? Why are the address numbers for p and m much larger*

*than for in and gv?*

*7. At print statement AM5, what is the address of pointer p, what is the value of pointer p, and*

*what value is pointer p pointing to?*

*8. At print statement AM7, what is the address of pointer p, what is the value of pointer p, and*

*what value is pointer p pointing to?*

*9. Consider AM14 to AM17. Is the PIC32 processor using big-endian or little-endian? Why?*

**Task 6**

*Surprise assignment!*