Exercises-Pointers

Description
A pointer is simply a data value that has been initialized with an address. As we have seen, the
manipulation of pointers is central to how everything works at the machine level. However, their use for
humans can be confusing, and programming with them the source of many bugs. For this reason,
modern programming languages like Java, and, to a lesser extent, C++, try to shield programmers from
the use of pointers.

In this exercise set we will practice with various kinds of pointers: doubly-indirect pointers, an array of
pointers that must be indexed into, pointers as arguments, pointers to functions, and even use two
pointers that go in different directions. All of the programs will rely on util.s

The partially-written programs and/or solutions indicated below can be found in exercises/pointers

Procedure
mesg.s
A first pass of this program is written for you. You will be modifying it twice.
The purpose of this program is simply to output a [Linux-standard] message corresponding to a error
code. (Valid codes are integers in the range [0,??], where ?? is the maximum valid error code, which is
calculated (see below)) The program asks the user for an error code, giving the user the range of valid
codes. Once the number is input,
- if the error code was not in range, it outputs an error message indicating that. Otherwise
  - if the error code has a message, it outputs the message corresponding to the error code
    by accessing an array of pointers to strings. The error code is an index into that array.
  - if the message for that error code is missing (i.e., the pointer is NULL), your program
    should indicate there is no message available.

The data area of mesg.s corresponds to a standard C initialized data area such as:

```c
char *messages[] = {"message 0", "message 1", "message 2", NULL, "message 4"};
```
(These are not the text messages. They are too long to copy here)
The maximum valid error code is calculated by using the two labels messages and Endmessages.
This allows the user to add messages later without modifying the code.
Start by examining mesg.s in the exercises/pointers directory. We may have gone over it in-class.

First modification to mesg.s
For your first modification, move the code that actually outputs the error message into a function. The
function has the prototype

```c
void errmsg(int code)
```
where code is the error code, which is the index into the array messages. Your function simply outputs
the appropriate message, or outputs that there is no message available, if that error message is missing.
this version allows a function to access a global array without passing the array as an argument. This is
not a good idea in general.
The code to check that the error code is in-range and to get the variable from the user should still reside
in the main program.
This version should be a fairly simple rearrangement of mesg.s. As a hint, do not use register $a1 in
the function errmsg(). You will need it for the next modification.

A sample solution for this exercise is in the file answers/mesg.s

mesg1.s
Modify your errmsg() function in mesg.s to take two arguments:
void errmsg (int code, char *messages[])  

errmsg should output the message using [the address of] the messages array, which is passed as its second argument.

This is a simple change to your version of mesg.s  
The solution to mesgl.s is in the pointers/answers directory.

stringsout.s  
Starting with the initialized data region stringsout.s in the pointers directory, add code to output the array strings, whose prototype is below  

char *strings[];  
The end of the strings array is indicated by a NULL pointer.

You should use SPIM syscalls to output the strings so you don't have to confirm each individual string.  

For practice, write a main program that simply calls a function stringsout() like this  

void stringsout(char **strings);  
The function stringsout would then output each string until a NULL pointer is found.

When your code is successful, you will see a simulated message from one of your classmates.  

There is a no solution to stringsout.s provided. However, this is very similar to the problem done in the mipsIV (pointers) section of the online notes, messages.s. We may also have done a version in class.

diff.s  
Beginning with the data areas defined in the file diff.s in the pointers directory, write this program as follows:

● The main program calls a function  

void doit(int *Result, int A[], int B[], int nelems)  

The quantity nelems should be calculated by assuming the array B starts immediately after the array A, and that A and B are the same length. The array Result must be dynamically allocated by main() using malloc(). It must be be the same length as A. (You will see that Result is not defined in the beginning copy of diff.s.) Note that the three arguments Result, A and B are the same type - equivalently pointer to an integer or array of int. (The only way that doit could fail is if one of its arguments was invalid, so we wont worry about it.)  

*Hint: although doit only has four arguments, reserve room for five arguments when you write main. This will help you in a possible exercise later.*

● the function doit calculates the difference of each element of A and B placing the result in the array pointed to by Result. It generates the difference at one index position by calling the function  

int diff(int a, int b)  
to do the difference of one element, like this  

Result[i] = diff(A[i],B[i])  

● the function diff returns the absolute value of (a-b)

● When doit returns, the main program outputs the Result array using a supplied function  

void outarray(int *Result, int nelems)  
That function is supplied for you. It uses the function itoax() from util.s to output the result as hexadecimal strings.

NOTE: outarray() uses the function PrintString() to output the hexadecimal strings to the MARS
messages window rather than to a dialog box. You will know if your program is working if the resulting output is multiples of 0x10: 0,10,20,30,...

Keep the program diff.s. You will be using it in the next exercise. (There is no solution for diff.s.)

**Version 2 - diffhalf.s**

Take a copy of diff.s and modify it as follows:

- add a function `unsigned half(unsigned u, unsigned l)` that treats its two arguments as `unsigned int`s. It returns an `unsigned int` that has the least significant 16-bits of each of its arguments combined in one `unsigned int` like this: the most-significant 16-bits of the result is the least-significant 16-bits of `u` and the least-significant 16-bits of the result is the least-significant 16-bits of `l`. Then modify `doit` to use the `half` function instead of the `diff()` function like this: `Result[i]=half(A[i],B[i])`

- Leave the `diff` function in your program, even though it is unused.

Name the new version diffhalf.s

There is no solution for diffhalf.s available.

**A modification to diffhalf.s**

Make a copy of diffhalf.s. Call it whatever you like. Then make the following changes:

- add a fifth argument to `doit()`. This fifth argument is a function pointer that is used to call the function to apply to each pair of elements in the array. `main` simply passes the address of the function that `doit` should use by placing the address of the function on the stack where the fifth argument should go
- in your `doit` function, load the fifth argument into a t-register. Then use the instruction `jalr $reg` to call the function.

Begin with your program passing the address of `diff` to `doit` and see if it works as before. Then edit your program and change the address passed to `doit` to `half` and test it.

Congratulations! You have just used a function pointer! And you are significantly on the way to doing Assignment Three!

**revstring.s**

This program starts with the program strdup.s that you wrote in Exercises-Procedures

Take the strdup.s program that you wrote in Exercises-Procedures and modify the `strdup` function so that when it copies the string to the newly-allocated buffer, it reverses it. You should also change the name of the function and the program to revstring. (If your version of strdup.s used `strcpy()`, you will have to change it to copy the bytes in a loop.)

This is a good exercise, as `revstring()` requires you to use pointers going in opposite directions.

There may be a complete or partial solution for `revstring()` in the answers directory as revstring.s if I have time to get to it. Otherwise, share your code on the group.