INFO1910 S1 2023 Week 3 Tutorial

Pointers and Strings

Question 1: Limited Strings

A string is a collection of characters. Let's try to use another type containing multiple bytes to construct a "string". In particular we will be using an eight byte integer to store up to eight characters. Some function headers have been provided, use this to store and print the 'string' "dinosaur".

Don't forget to provide a makefile!

```
#define bad_string size_t
// size_t is eight bytes long!

/*
 * str_print
 * Prints our "bad" string
 * :: const BAD_STRING str :: The "string" to print
 * Returns nothing, but prints to standard output
 */
void str_print(const bad_string str);

/*
 * byte_place
 * Places a character at a given byte position in our "string"
 * :: bad_string str :: Our current string
 * :: const char character :; The character to place
 * :: size_t position :: the position to place the character
 * Returns a new "bad_string" with the character placed
 * in the correct position
 */
bad_string byte_place(
  bad_string str,
  const char character,
  const size_t position);
```

For example:

```
bad_string str = 0;
str = byte_place(str, 'a', 0);
str = byte_place(str, 'b', 1;
str_print(str);
str = byte_place(str, 'c', 2);
str_print(str);
```

Should print `ab` then `abc`.

Consider what would happen in your solution if you tried to call `byte_place` twice on the same position

Pointers

Another type in C is the pointer type. Pointers are variables that store an address in memory. For this we add another two unary operators to our list from last week:

- `&` Returns the address of a variable
- `*` Dereferences an address (returns the data at that location)

From our discussion of types, you might determine that when dereferencing an address, we need a type with which to represent it. For these purposes, every data type has an associated pointer type, for example:

```
int x = 5; // Four bytes
int* x_ptr = &x; // A pointer to the address of x
*x_ptr = 3; // Assign to the location pointed to by x_ptr
printf("%d\n", x);
```

Here `x_ptr` takes the type of a pointer to an integer, when we dereference, it is interpreted as an integer.

Pointers allow us to modify varibles across scopes, as we pass the address to the original version of the variable, rather than a copy of the data, we can modify values between functions

The NULL pointer points to the address 0 and should always throw a segmentation fault when read to or written from.

Question 2: Pointer Mod

Write a function that modifies the value of an integer variable stored within the main function

```c
/*
 * var_mod
 * Modifies the value stored at a pointer address
 * :: int* mod_int :: The address of the value to be modified
 * :: const int value :: The value to store at that address
 * Returns nothing
 */
void var_mod(int* mod_int, const int value);

int main()
{
  int x = 5;
  var_mod(&x, 7);
  printf("%d\n", x); // Should print 7
}
```

How does this differ from the previous problem where you returned a new double variable?

Question 3: Pointer Swap

Write a function that swaps the memory between two integers. The function will take two arguments, the addresses of the two integers to be swapped.

You should assert that neither of the pointers is a NULL, if it is the function should end.

```c
/*
 * mem_swap
 * Swaps the memory between the two specified integer pointers
 * :: int* a :: The first pointer to swap
 * :: int* b :: The second pointer to swap
 * Returns nothing, swaps occur in place
 */
void mem_swap(int* a, int * b);
```

Question 4: Pointer Array Equivalence

An array is defined by:

```
int x[10];
```

This creates a 40 byte block of memory (on modern processors), with x being a 'label' for the initial value of that block. Here our dereference operator comes in handy.

```
*x
```

Returns the value at the first four byte block (depending on the size of an integer), similarly

```
*(x + 1)
```

Will return the value at the second four byte block. You may notice that the incrementation of the address is tailored to the type of the pointer, if x was a `char*` it would act over one byte blocks, while `double*` would act over eight byte blocks.

The above representation is a bit verbose, so we can adopt the following syntatic sugar

```
x[n]
```

Which is exactly equivalent to

```
*(x + n)
```

Indeed, it is so equivalent that the following syntax is also legal (note using the following syntax **will** result in an automatic fail in any assesment you use it in)

```
2[x]
```

It's noteworthy that the length of these arrays can **only** be defined at compile time. You should not leave these brackets empty, or attempt to have a variable length array.

Basics of Recursion

We haven't yet covered control flow, and as that's a week and a topic in and of itself we will instead be making do with recursion and ternary operators for now.

Recursion is the concept that a function can call itself; proofs exist that demonstrate that this property is as computationally powerful as iteration.

```
int pow_recurse(int base, unsigned int exp)
{
        return (exp == 0) ? 1 : pow_recurse(base, exp - 1) * base;
}
```

This may not be the neatest method of solving problems in C, but it's certainly an interesting method.

Question 5: Finding the length of a string

A C string is a block of memory that is interpreted as a series of one byte ascii characters. The final byte is a 'NULL' terminator byte, which consists of only zero bits.

Write a function that finds the length of a C string; as we have not yet covered loops you should not use them here.

```
size_t c_strlen(char* str);
```

Question 6: Find the length of an array

Write a function that finds the length of a generic array.

```
size_t array_len(void* arr);
```

Would specifying the type of the array (for example integer or double) make this task easier?

void*

Last week we briefly discussed the `void` type as the 'None' type. This time around we will discuss `void*` as the 'UR' type, or the generic type. Type information primarily conveys the number of bytes an object occupies along with what operators can be applied to it. The size of a void object is undefined. As a result a `void*` typed variable may point to memory of any size.

The only problem that arises is that as `void` typed variables are not permitted that `void*` pointers must be cast to another pointer type before they may be dereferenced. In cases were operations are being performed on raw memory the size of the underlying objects must be passed as a variable.

Question 7: Generic Pointer Swap

Write a function that swaps the memory between two arrays. The function will take two arguments, the addresses of the two integers to be swapped.

You should assert that neither of the pointers is a NULL, if it is the function should end.

```
/*
 * mem_swap
 * Swaps the memory between the two specified integer pointers
 * :: void* a :: The first pointer to swap
 * :: void* b :: The second pointer to swap
 * :: size_t n_bytes :: The number of bytes in each elenment
 * :: size_t len :: The number of elements in a and b
```

```c
 * Returns nothing, swaps occur in place
 */
void mem_swap(void* a, void* b, size_t n_bytes, size_t len);

int main()
{
        int a[5] = {1, 2, 3, 4, 5};
        int b[5] = {2, 4, 6, 8, 10};

        double c[7] = {1, 4, 9, 16, 25, 36, 49};
        double d[7] = {1, 2, 3, 4, 5, 6, 7};

        mem_swap(a, b, sizeof(int), 5);
        mem_swap(c, d, sizeof(double), 7);

        return 0;
}
```

Reading Command Line Arguments

Command line arguments are passed as an "array of pointers", though this is a somewhat inaccurate description. Two arguments are passed, the first is an integer that dictates the number of arguments, the second is an array of pointers to character addresses that contain the strings of each of the arguments, split by whitespace.

```c
int main(int argc, char** argv);
```

Here the first argument, which by convention is called argc, dictates the number of arguments, while argv contains the pointers to pointers to C strings.

Question 8: Hear and Respond

Write a program that reads each command line argument and prints the string stored there, again we don't yet have loops so you'll have to make do without them.

Question 9: Reverse a string

Write a function that reverses a string, then prints it, once again we cannot use loops.

```
./reverse Yellow
wolleY
```

Function Pointers

Given that functions are stored in memory it holds that pointers may contain the address of these functions.

```
void print_a()
{
        printf("Hello");
        return;
}

int main()
{
        void (*func)() = print_a;
        func();
        return 0;
}
```

Of course it then follows that you may have arrays of functions, may pass functions as arguments to other functions and other such intersting situations.

You may find this property very useful for the next question.

Question 10: Calculator

Write a simple command line calculator.

- You should take three command line arguments, two integers **followed** by an operation
- If not enough arguments are provided, print 'Not enough arguments!'
- You should support addition, multiplication, subtraction, division and modulus
- Pay attention for when the output might require a floating point representation

You will find the `atoi` or `sscanf` function useful here, be sure to check their man pages.

For example:

```
./calc 14 7 +
21

./calc 8 3 *
24

./calc 5 2 /
2.5
```

```
./calc 5 2
Not enough arguments!
```

You might find a switch statement much more useful here rather than a large amount of if statements!

Question 11: Fun with Function Pointers

The map operation is a useful abstraction which performs an operation on every element in a collection. For example: map$(+1, [1, 2, 3]) \to [2, 3, 4]$

Your goal is to write your own implementation of the map function in C that acts on an array in place. Remember that loops are still prohibited.

```c
void map(void (*fn), void* elements, size_t bytes, size_t length)
{
        // TODO

        return;
}

int plus_one(int val)
{
        return val + 1;
}

int times_two(int val)
{
        return val * 2;
}

int main()
{
        int x[3] = {1, 2, 3};

    map(plus_one, x, 3, sizeof(int));
        printf("%d\n", x[0]);

    map(times_two, x, 3, sizeof(int));
        printf("%d\n", x[0]);

        return 0;
}
```

Would the map function work on functions with more than one argument?

Question 12: (Extension) The Stack list

Keen eyed observers may have noticed that all problems posed to date deal with a static allocation of memory. Try to construct a dynamic allocation of memory using the call stack. If you are confused by the concepts then shelve this problem until we revisit it in later weeks.

```
void* stacklist(
        void (*fn)(void*),
        void* first,
        void* last,
        size_t n_bytes)
{
        // Stacklist structure here

        return (0 == n_bytes)
                ? fn(stacklist)
                : stacklist(fn, first, new_last, n_bytes);
}
```

You will also need to write your own function to query elements in the stacklist.