Passing pointers to functions

Consider a very simple function, whose role is to swap two integer values:

```
#include <stdio.h>

void swap(int i, int j)
{
    int temp;

    temp = i;
    i    = j;
    j    = temp;
}

int main(int argc, char *argv[])
{
    int a=3, b=5;  // MULTIPLE DEFINITIONS AND INITIALIZATIONS

    printf("before a=%i, b=%i\n", a, b);

    swap(a, b);    // ATTEMPT TO SWAP THE 2 INTEGERS

    printf("after  a=%i, b=%i\n", a, b);
    return 0;
}

before a=3, b=5
after  a=3, b=5
```

Doh! What went wrong?

The "problem" occurs because we are not actually swapping the values contained in our variables *a* and *b*, but are (successfully) swapping *copies* of those values.

Passing pointers to functions, *continued*

Instead, we need to pass a 'reference' to the two integers to be interchanged.

We need to give the *swap()* function "access" to the variables *a* and *b*, so that *swap()* may modify those variables:

```
#include <stdio.h>

void swap(int *ip, int *jp)
{
    int temp;

    temp  = *ip;    // swap's temp is now 3
    *ip   = *jp;    // main's variable a is now 5
    *jp   = temp;   // main's variable b is now 3
}

int main(int argc, char *argv[])
{
    int a=3, b=5;

    printf("before a=%i, b=%i\n", a, b);

    swap(&a, &b);   // pass pointers to our local variables

    printf("after  a=%i, b=%i\n", a, b);

    return 0;
}

before a=3, b=5
after  a=5, b=3
```

Much better! Of note:

- The function *swap()* is now dealing with the original variables, rather than new copies of their values.
- A function may permit another function to modify its variables, by passing *pointers to those variables*.
- The receiving function now modifies *what those pointers point to.*

Duplicating a string

We know that:

- C considers null-byte terminated character arrays as strings, and
- that the length of such strings is not determined by the array size, but by where the null-byte is.

So how could we take a duplicate copy, a clone, of a string? We could try:

```
#include <string.h>

char *my_strdup(char *str)
{
    char bigarray[SOME_HUGE_SIZE];

    strcpy(bigarray, str);  // WILL ENSURE THAT bigarray IS NULL-BYTE TERMINATED

    return bigarray;        // RETURN THE ADDRESS OF bigarray
}
```

But we'd instantly have two problems:

1. we'd never be able to know the largest array size required to copy the arbitrary string argument, and
2. we can't return the address of any local variable. Once function *my_strdup()* returns, variable *bigarray* no longer exists, and so we can't provide the caller with its address.

Allocating new memory

Let's first address the first of these problems - we do not know, *until the function is called*, how big the array should be.

It is often the case that we do not know, *until we execute our programs*, how much memory we'll really need!

Instead of using a fixed sized array whose size may sometimes be too small, we must *dynamically request* some *new* memory at runtime to hold our desired result.

This is a fundamental (and initially confusing) concept of most programming languages - the ability to request from the operating system *additional* memory for our programs.

C11 provides a small collection of functions to support *memory allocation*.
The primary function we'll see is named *malloc()*, which is declared in the standard<*stdlib.h*> header file:

```
#include <stdlib.h>

extern void *malloc( size_t nbytes );
```

- *malloc()* is a function (external to our programs) that returns a pointer.
  However, *malloc()* doesn't really know what it's returning a pointer to - it doesn't know if it's a pointer to an integer, or a pointer to a character, or even to one our own user-defined types.

  For this reason, we use the *generic pointer*, pronounced *"void star"* or *"void pointer"*.

  It's a pointer to "something", and we only "know" what that is when we place an *interpretation* on the pointer.

- *malloc()* needs to be informed of the amount of memory that it should allocate - the *number of bytes* we require.

  We use the standard datatype *size_t* to hold an integer value that may be 0 or positive (we obviously can't request a negative amount of memory!).

  We have used, but skipped over, the use of *size_t* before - it's the datatype of values returned by the **sizeof** operator, and the pedantically-correct type returned by the *strlen()* function.

Checking memory allocations

Of course, the memory in our computers is finite (even if it has several physical gigabytes, or is using virtual memory), and if we keep calling *malloc()* in our programs, we'll eventually exhaust available memory.

Note that a machine's operating system will probably not allocate *all* memory to a single program, anyway. There's a lot going on on a standard computer, and those other activities all require memory, too.

For programs that perform more than a few allocations, or even some potentially large allocations, we need to *check* the value returned by *malloc()* to determined if it succeeded:

```
#include <stdlib.h>

  size_t bytes_wanted   = 1000000 * sizeof(int);

  int    *huge_array    = malloc( bytes_wanted );

  if(huge_array == NULL) {   // DID malloc FAIL?
      printf("Cannot allocate %i bytes of memory\n", bytes_wanted);
      exit( EXIT_FAILURE );
  }
```

Strictly speaking, we should check *all* allocation requests to both *malloc()* and *calloc()*.

Duplicating a string *revisited*

We'll now use *malloc()* to dynamically allocate, at runtime, exactly the correct amount of memory that we need.

When duplicating a string, we need enough new bytes to hold every character of the string, *including a null-byte* to terminate the string.
This is 1 more than the value returned by *strlen*:

```
#include <stdlib.h>
#include <string.h>

char *my_strdup2(char *str)
{
    char *new = malloc( strlen(str) + 1 );

    if(new != NULL) {
        strcpy(new, str);  // ENSURES THAT DUPLICATE WILL BE NUL-TERMINATED
    }
    return new;
}
```

Of note:

- we are not returning the address of a local variable from our function - we've solved *both* of our problems!
- we're *returning a pointer* to some *additional* memory given to us by the operating system.
- this memory does not "disappear" when the function returns, and so it's safe to provide this value (a pointer) to whoever called *my_strdup2*.
- the new memory provided by the operating system resides in a reserved (large) memory region termed *the heap*. We never access the heap directly (we leave that to *malloc()*) and just use (correctly) the space returned by *malloc()*.

Allocating an array of integers

Let's quickly visit another example of *malloc()*. We'll allocate enough memory to hold an array of integers:

```
#include <stdlib.h>

int *randomints(int wanted)
{
    int  *array = malloc( wanted * sizeof(int) );

    if(array != NULL) {
        for(int i=0 ; i<wanted ; ++i) {
            array[i] = rand() % 100;
        }
    }
    return array;
}
```

Of note:

- *malloc()* is used here to allocate memory that we'll be treating as integers.
- *malloc()* does not know about our eventual use for the memory it returns.
- how much memory did we need?

  We know how many integers we want, *wanted*, and we know the space occupied by each of them, **sizeof(int)**.
  We thus just multiply these two to determine how many bytes we ask *malloc()* for.

Requesting that new memory be cleared

In many situations we want our allocated memory to have a known value. The C11 standard library provides a single function to provide the most common case - clearing allocated memory:

```
#include <stdlib.h>

extern void *calloc( size_t nitems, size_t itemsize );
    ....
    int  *intarray = calloc(N, sizeof(int));
```

It's lost in C's history why *malloc()* and *calloc()* have different calling sequences.

To explain what is happening, here, we can even write our own version, if we are careful:

```
#include <stdlib.h>
#include <string.h>

void *my_calloc( size_t nitems, size_t itemsize )
{
    size_t  nbytes  = nitems * itemsize;

    void *result = malloc( nbytes );

    if(result != NULL) {
        memset( result, 0, nbytes );  // SETS ALL BYTES IN result TO THE VALUE 0
    }
    return result;
}

    ....
    int  *intarray = my_calloc(N, sizeof(int));
```

Deallocating memory with *free*

In programs that:

- run for a long time
  (perhaps long-running server programs such as *web-servers*), or
- temporarily require a lot of memory, and then no longer require it,

we should *deallocate* the memory provided to us by *malloc()* and *calloc()*.

The C11 standard library provides an obvious function to perform this:

```
extern void free( void *pointer );
```

Any pointer successfully returned by *malloc()* or *calloc()* may be *free*d.

Think of it as requesting that some of the allocated *heap memory* be given back to the operating system for re-use.

```
#include <stdlib.h>

    int   *vector = randomints( 1000 );

    if(vector != NULL) {
        // USE THE vector
        ......
        free( vector );
    }
```

Note, there is no need for your programs to completely deallocate all of their allocated memory before they exit - the operating system will do that for you.

Reallocating previously allocated memory

We'd already seen that it's often the case that we don't know our program's memory requirements until we run the program.

Even then, depending on the input given to our program, or the execution of our program, we often need to allocate more than our initial "guess".

The C11 standard library provides a function, named *realloc()* to grow (or rarely shrink) our previously allocate memory:

```
extern void *realloc( void *oldpointer, size_t newsize );
```

We pass to *realloc()* a pointer than has previously been allocated by *malloc()*, *calloc()*, or (now) *realloc()*.
Most programs wish to extend the initially allocated memory:

```
#include <stdlib.h>

  int  original;
  int  newsize;
  int  *array;
  int  *newarray;

  ....
  array = malloc( original * sizeof(int) );
  if(array == NULL) {
      // HANDLE THE ALLOCATION FAILURE
  }
  ....

  newarray = realloc( array, newsize * sizeof(int) );
  if(newarray == NULL) {
      // HANDLE THE ALLOCATION FAILURE
  }
  ....
```

```
#include <stdlib.h>

  int  nitems = 0;
  int  *items = NULL;

  ....
  while( fgets(line ....) != NULL ) {
      items = realloc( items, (nitems+1) * sizeof(items[0]) );
      if(items == NULL) {
              // HANDLE THE ALLOCATION FAILURE
      }
      ....    // COPY OR PROCESS THE LINE JUST READ
      ++nitems;
  }
  ....
  if(items != NULL) {
      free( items );
  }
```

Of note:

- if *realloc()* fails to allocate the revised size, it returns the *NULL* pointer.
- if successful, *realloc()* copies any old data into the newly allocated memory, and then deallocates the old memory.
- if the new request does not actually require new memory to be allocated, *realloc()* will usually return the same value of *oldpointer*.
- a request to *realloc()* with an "initial" address of NULL, is identical to just calling *malloc()*.