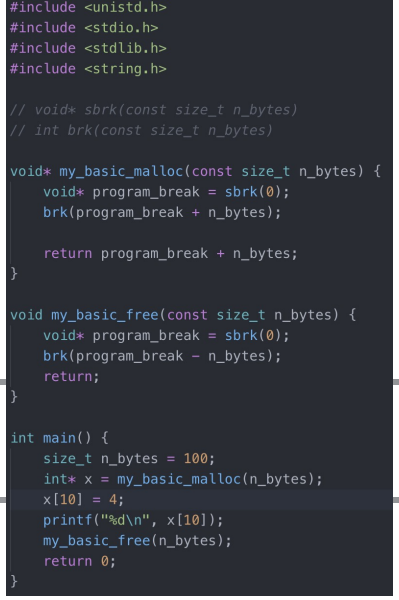


THE UNIVERSITY OF SYDNEY

COMP2017 Week 5 Tutorial

Dynamic Memory

One of the most important uses of dynamic memory is to create a dynamically sized array at run time. Up until `C99` it was impossible to do this without calling malloc. For an array with some length 'length', we can create a dynamically sized array as follows:

```
int* my_arr = (int*) malloc sizeof(int) * length);
```

As you should be aware by now, arrays are effectively pointers to the first element of the array, we can use the pointer returned by malloc to access any element in the array.

```
my_arr[10];
```

And in time honoured C tradition, you can of course run dangerously over the ends of the array where no pointer has gone before, into uncharted memory space. You can also allocate memory for structs, pointers, function pointers and any thing else that can be put to memory. If you want your memory all to be initialised to 0, then you can call 'calloc'.

```
int* my_zeros = (int*) calloc(sizeof(int), length);
```

Notice that the syntax for calloc differs from malloc in that it takes two arguments, the first the size of the object, and the second the length of the array. Of course all of these objects need to be *freed* to prevent memory leaks.

Once finished with a memory allocation from `malloc`, `calloc` or `realloc` it will need to be `free`'d from our program's clasps. The `free` function will accept pointer with a value to a heap allocation and message the operating system to reclaim the allocation.

```
int* my_arr = (int*) malloc(sizeof(int) * length);

//Other parts of the program

free(my_arr);
```

Why `malloc` over VLAs?

Dynamicly allocated arrays on the stack are typically bad practice since stack space is small, it does not handle negative values, it generates a substantial amount of code to dynamically allocate the array at run time and the allocation requested can be larger than the stack space itself.

Header Files

Before we start with pointers and array we will cover header files. In the previous tutorial you will have used the #include preprocessor directive. When writing multiple source files you will want to expose those functions to other source files.

If we have functions we want to expose to other source files we can specify them in the header

```
//.. in myheader.h

void my_function();

void another_function();
```

This also includes `structs`, `enums` and `unions` as well

Header Guards

With any good header file we will want to ensure that we guard against unnecessary inclusion (we only want to include the header file once).

```
#ifndef MY_HEADER_FILE_H
#define MY_HEADER_FILE_H
        //Your declarations and includes can go here
#endif
```

Without the header guard we may have a cyclic dependency where the header may try to be included infinite times.

Once we have created our own header file we can then include it in our source file or we could even include it in other header files. For locally referenced header files we use "" instead of <>

```
#include "myheader.h"
//.. rest of the code
```

a.c

```
#include "b.h"

int main()
{
    func_b();
    return 0;
}
```

b.c

```
#include "b.h"

void func_b()
{
    printf("qwop\n!");
}
```

b.h

```
#ifndef B_H
#define B_H 1

#include <stdio.h>


void func_b();

#endif
```

Pre-Tutorial Questions

Question 1: Lifetimes of Variables

For each of the variables in the code below, determine both where they are stored, what initial values they take, and what their lifetimes are.

lifetime of global: whole program
lifetime of stack: main function
lifetime of heap: until you free

```c
#include <stdio.h>
#include <stdlib.h>

int var_a;

int sum (const int* var_b) {
        static int var_c;
        var_c += var_b;
        return var_c;
}

int main (int argc, char** argv) {
        int var_d = 0;

        int* var_e = malloc(sizeof(int));
        if (NULL == var_e)
        {
                perror("Malloc Failed!");
                return 1;
        }
        *var_e = 2;
        sum(&var_d);
        sum(&var_a);
        sum(var_e);
        char* var_f = calloc(sizeof(char), 100);
        if (NULL == var_f)
        {
                perror("Calloc Failed");
                return 1;
        }

        free(var_e);
        return 0;
}
```

global

stack

static

stack

heap

var_e value is in stack, the pointer is in heap

heap

- What is the lifetime of `var_a`,`var_c`, `var_d` and `var_e`?
- What would happen if we did not free var_f — memory leak
- When and where might memory leaks occur in this code? — forget free var_f

Question 2: Linked Lists

Implement a singularly linked list in C. The end of the list should be indicated by the next element pointing to `NULL`. You will want to create your own node struct containing any appropriate types.

```
/*
        list_init
        Creates a new list by creating and head node
        :: int value :: The value to be stored in the head node
        Returns a pointer to the newly created node
*/
struct node* list_init(int value);
/*
        list_add
        Adds a node containing a specified value to an existing list
        :: int value :: The value to be stored in the new node
        Does not return anything
*/
void list_add(struct node* head, int value);

/*
        list_delete
        Removes the specified node from the list and updates the list accor
        :: struct node* n :: The pointer to the node to be deleted
        Does not return anything
*/
void list_delete(struct node** head, struct node* n);

                              make a copy if  n is head
/*
        list_next
        Returns a pointer to the next node in the list
        :: const struct node* n :: The node
        Returns a pointer to the next node
*/
struct node* list_next(const struct node* n);

/*
        list_free
        Frees all existing nodes in a list
        :: struct node* n :: The pointer to the head node of the list
        Does not return anything
*/
void list_free(struct node* head);
```

A different approach to this problem is to get the programmer to manage their own memory rather than encapsulating them in methods. Outline the advantages and disadvantages of each method?

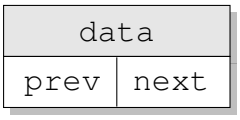
Question 3: Circular linked list

Update your linked list implementation to a circular linked list that. This will have a reference to the starting node if there is a node present in the current list.

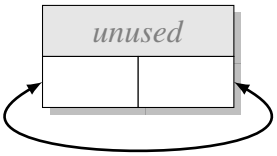
The circular linked list will be made up of nodes like so:

```
typedef struct node node;
struct node {
        int data;
        node* prev;
        node* next;
}
```

Recall that in circular linked list, the last element links back to the first element. In a circular linked list of zero elements, the first node simply links back to itself, the data stored in the head element is unused.

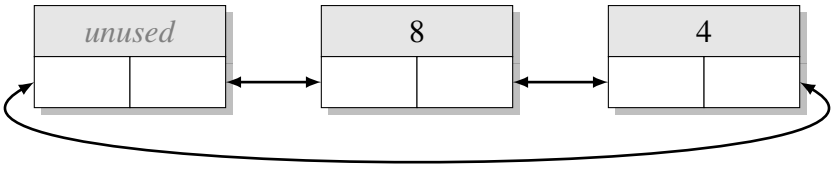


node



An empty circular linked list

The last element of the circular linked list joins back to the first element:



A circular linked list containing two elements: 8, 4

Memory Layout:
- Stack: local variables, function arguments, return addresses, temporary storage
- Heap: dynamically allocated memory
- Global/Static: golbal variables, static variables
- Code: program instructions

Tutorial Questions

Question 4: Dynamic array

You can use `malloc` to allocate a buffer and then use `realloc` to grow the size of your buffer.
A simple and efficient strategy of resizing your buffer is to double its size everytime it becomes full.

Compile your program with debugging symbols and address sanitizer using `clang`.

```
$ clang -g -std=c11 -Wall -Werror -fsanitize=address dynamo.c -o dynamo
```

Alternatively you can use `valgrind` instead of compiling with address sanitizer.

```
$ clang -g -std=c11 -Wall -Werror dynamo.c -o dynamo
$ valgrind ./dynamo
```

These tools detect common memory errors such as invalid memory access as well as memory leaks caused by not calling `free`. The `-g` generates debugging symbols and allows these tools to give you more helpful output.

The typical usage of `malloc, calloc, realloc, free` are shown below, note the use of the cast since these functions return a `void *` that needs to be converted to the proper type.

```c
// Allocate 100 bytes of memory
char* buffer = (char *) malloc(100 * sizeof(char));

// Allocate memory for 10 integers
int* values = (int *) malloc(10 * sizeof(int));

// Allocate memory for 10 integers, setting all elements to zero
int* zeroes = (int *) calloc(sizeof(int), 10);

// Returns a pointer to a new block of memory that contains space
// for 200 characters. The values of the elements are preserved.
buffer = (char *) realloc(buffer, 200 * sizeof(char));

// Free allocated memory
free(buffer);
free(values);
free(zeroes);
```

Question 5: Dynamic Array Structure

After familiarising yourself with the pattern involved with resizing a dynamic array. Create a dynamic array structure that will hold the capacity, size and contents of the array. Your structure should also have a set of functionality accompanying it that will allow programmers to get, set, add, remove and lastly delete the entire allocation.

```
/*
        dyn_array_init
        Creates a new dynamic array with a default allocation
        :: int value :: The value to be stored in the head node
        Returns a pointer to the newly created node
*/
struct dyn_array* dyn_array_init();
/*
        dyn_array_add
        Adds an element to the list
        :: struct dyn_array* dyn :: The pointer to the dynamic array
        :: int value :: The value to be stored in the new node
        Does not return anything
*/
void dyn_array_add(struct dyn_array* dyn, int value);
/*
        dyn_array_delete
        Removes an element from the list and updates the list accordingly
        :: struct dyn_array* :: The pointer to the dynamic array
        :: int index :: index of element to remove
        Does not return anything
*/
void dyn_array_delete(struct dyn_array* dyn, int index);

/*
        dyn_array_get
        Returns a pointer to the next node in the list
        :: struct dyn_array* dyn :: The dynamic array
        :: int index :: the index at the specific array
        Return the int at the specified index
*/
int dyn_array_get(struct dyn_array* dyn, int index);

/*
        dyn_array_free
        Frees the current array allocation
        :: struct dyn_array* dyn :: The dynamic array
        Does not return anything
*/
void dyn_array_free(dyn_array* dyn);
```

Question 6: Seg Trek

Fix the following code, valgrind, -fsanitize and gdb may be very useful. **man** these commands, or ask your tutor if you are unsure as to their use.

```
int main() {
                                                        8
        void* a = (int*)malloc(sizeof(int*) * (1 << 3));
        void* b = (int*)malloc(sizeof(int*) * 9);

        for (size_t i = 0; i <= 9; i++)
        {
                ((int*)a)[10] += i >> 1;   out of bound for a
                b[i] = i + ~i;             out of bound for b
                if (i == 8)
                {
                        free(a);
                }
        }
        free(a);                           not free(b)
        return ((int*)a)[0];               heap free after use
}
```

Question 7: Into the Void

Genericise your data structures by replacing the type of the values in each node or element with `void*` and then cast back to the required type when you get an object from the list. You should consider an efficient way of mallocing and memcpying the correct amount of data for your `void*` to point to. How might this benefit for a hybridised approach where mallocing the object is handled by the user and then passed to the list?

Can this hybridised approach also be encapsulated by some appropriate function?

Consider the advantages and disadvantages to casting everything to `void*`. If you do cast everything to `void*` what can you do to make your code more readable to your beleaguered tutors?

Try rewriting your data structures to allow any data structure to be stored.

- Linked List
- Circular Buffer
- Dynamic Array