```
CS 214 / 2021-02-08
===================

Announcements
-------------

* Recitations begin today
* Group Project 1 forthcoming
* Individual Homework 2 forthcoming

Memory management
-----------------

global objects - referred to using variables (direct) or pointers (indirect)
stack objects  - referred to using variables (direct) or pointers (indirect)
heap objects   - always referred to indirectly (using pointers)

Use & to get the address of a variable
        -> turn direct reference to indirect reference

Use (unary) * to follow a reference (dereference - turn indirect to direct)


Use malloc to create heap objects
        -> returns pointer to newly created object

Use free to deallocate heap objects
        -> give pointer to object we want to delete
        -> do not attempt to free global or stack objects!

Other useful functions


void *calloc(size_t num_elems, size_t elem_size);
        - allocates space and clears all the bytes (sets all bits to 0)
        - number of bytes allocated is num_elems * elem_size
        - avoids any problems with forgetting to initialize values
                - allocate an array or struct with all elements/fields set to 0

        - having two arguments may be to prevent integer overflow if you are
                allocating a lot of memory
                - but then why does malloc only take one argument? unclear

        int *A = calloc(100, sizeof(int));
                // allocate space for 100 integers and set them all to 0

        - why have calloc and malloc?
                malloc is faster because it doesn't need to write to the allocated space
                        i.e., maybe you are going to immediately initialize to non-zero values
                calloc is safer because it ensures a known value of the data

        - calloc is essentially just malloc + memset


void *memset(void *p, int c, size_t n);
        - sets the value of bytes in memory
                - p - pointer to start of object
                - n - the number of bytes in the object
                - c - actual byte (char) to write
                                only the lower 8 bits of c will be used

        - may be faster than writing a loop yourself
                - possibly takes advantage of OS operations; hardware accelleration

        - returns the same pointer it was given


void *realloc(void *p, size_t size);

        - changes the size of a heap object
                - shrinks in place
                - grows in place if possible, or allocates new space & moves data

        arguments:
                p - pointer to object we are resizing
                size - number of bytes we want the object to be

        returns pointer to object - may have moved!
                for this reason, you should not use the old pointer value after a call
                to realloc

                - realloc is only safe for objects with a single reference

recall: ArrayList in Java
        array that you can grow

We can implement ArrayList in C using a pointer and two integers

// global variables
int *list, size, used;

        // to have multiple array lists, create a struct that holds these

// set up array list: allocate space for data, set variables
// (essentially a constructor)
void init(int init_size)
{
        size = init_size;
        list = (int *)malloc(size * sizeof(int));
                // cast is unnecessary; just making sure I have the right pointer type
        used = 0;
}


// to append onto the array, we increase used
        // used: number of array elements "in use" (initially zero)

        // if used == size, we use realloc to make the array bigger

// add value onto the end of the array list
        // make additional space if necessary
void append(int i)
{
        if (used == size) {
                size = size * 2;
                list = (int *)realloc(list, size);

        }
        list[used] = i;
        used++;
}

int remove()
{
        if (used > 0) {
                --used;
                return list[used];
        }

        return 0; // or throw an error or something
}



void *memcpy(void *dest, void *src, size_t bytes);

        - copy data from one object to another
        arguments:
                dest - where data will go
                src  - where data comes from
                bytes - how many bytes to copy

        returns the destination pointer

        - you must ensure that src and dest point to objects of the correct size
        - src and dest must not overlap

        - could be implemented as on O(n) loop, or use OS/HW tools for speed

        - memcpy copies bits: does not care about types
                - does not convert values

        - you must ensure that src and dest point to appropriately typed data


        something like
                char foo[] = "Hello";

        is equivalent to doing;
                char foo[6];
                memcpy(foo, "Hello", 6);


void *memmove(void *dest, void *src, size_t bytes);

        - same as memcpy, except that source and destination may overlap
        - may be less efficient than memcpy
                - may be more efficient than writing a loop yourself

        example: removing elements from the start/middle of an array

                int a[100];
                ...
                memmove(a, &a[1], 99 * sizeof(int));
                        // copy 99 integers from a[1] .. a[99] into a[0] .. a[98]
                                // note that a[99] will retain the same value

        does a = &a[1] work?
                - no: compiler will not let us assign to an array variable

        what about pointers?

        int *p = malloc(100 * sizeof(int));

        p = &p[1];   // allowed, but now we don't have a pointer to the original object
        // unless we do free(p - 1);

        int *q = p;

        q = &q[1];
        q = q + 1;
        ++q;
                // be careful when doing pointer shenanigans!
                // if you lose the pointer to a heap object, you won't be able to free it


char *strcpy(char *dest, char *src);
        - similar to memcpy, but we don't specify a size
                - instead, we copy from src until we reach an all-0 byte ('\0')

        - still must ensure that dest and src do not overlap
        - still must ensure that dest is large enough to hold data

char *strncpy(char *dest, char *src, size_t n);
        - copies up to n bytes from src to dest
                - stops after '\0' byte, or after n bytes

        - note that strncpy may result in a non-terminated string, if src is longer
          than n bytes
                - you can explicitly set the last byte to '\0' if there is a problem

        char foo[100];

        strncpy(foo, some_string, 99);  // copy up to 99 chars
                // adds null terminator if some_string is less than 99 chars long
        foo[99] = '\0';   // ensure that foo is null-terminated
                // redundant if some_string is < 99 chars long, but that is okay



Deciding when to use memcpy, memmove, strcpy, strncpy is fairly straightforward

        memcpy and memmove specify exactly how many bytes to move
        strncpy may copy fewer bytes
        strcpy is potentially more efficient than strncpy

        in general: pay attention to how long your strings may be
                are you copying data from disjoint objects or within an object


Example: duplicating a string

char *str_dup(char *str)
{
        char *dest = malloc(strlen(str) + 1);
                // allocate space for the whole string, plus null terminator

        strcpy(dest, str);

        return dest;
}

char *str_dup(char *str)
{
        int len = strlen(str) + 1;
        char *dest = malloc(len);
                // allocate space for the whole string, plus null terminator

        memcpy(dest, str, len);
                // faster than strcpy because it doesn't need to look for '\0'
                // dest is new object, so it can't overlap with str

        return dest;
}

// or use strdup from Posix library

Use man for more details
documentation is also available on-line


C Pre-Processor & separate compilation
--------------------------------------

The C Pre-Processor (CPP) runs before the compiler starts compiling
- fancy find/replace stage that modifies your source code

- can be used for code generation or for defining constants

Lines starting with # are preprocessor directives

#include <std_file.h>
#include "my_file.h"

        #include copies the content of a file into your file
        <filename> says to look for the file in the "include" directory
                (e.g., /usr/include)
        "filename" says to look relative to the current directory

        Primarily used to include "header files"
                typically end in .h
                include function prototypes, struct declarations, typedefs, etc.

        When I write
                #include <stdio.h>
        CPP will copy the definitions in stdio.h into my file
                now I can call printf(), etc, without needing to declare them

        Typically at the top of a file, but can occur anywhere
                but included text is included at that point

        Note that this is different from linking, but related
                include brings in declarations from other files
                linking connects our code to definitions from libraries

        #include "something.txt"
                // works fine

#define MACRO optional_value

        declares a "macro", which is used for text substitution
        i.e., replacing a constant with its value

        #define MEMSIZE 100

        ...

                int mem[MEMSIZE];  // after preprocessing, MEMSIZE will be replaced with 100

        Using macros allows us to keep things consistent
                If you see 100, you might not know what it means
                MEMSIZE tells you

        If we change MEMSIZE, we can do it in one place and all references will update

        Any time you have numbers other than 1 and 0, consider defining a macro

                MEMSIZE * 2
                        will be replaced with
                100 * 2
                        and the compiler will turn that into
                200


Next time:
        macros with arguments
        conditional compilation

        also: file descriptors
```

```
CS 214 / 2021-02-08
===================

Announcements
-------------

* Recitations begin today
* Group Project 1 forthcoming
* Individual Homework 2 forthcoming

Memory management
-----------------

global objects - referred to using variables (direct) or pointers (indirect)
stack objects  - referred to using variables (direct) or pointers (indirect)
heap objects   - always referred to indirectly (using pointers)

Use & to get the address of a variable
        -> turn direct reference to indirect reference

Use (unary) * to follow a reference (dereference - turn indirect to direct)


Use malloc to create heap objects
        -> returns pointer to newly created object

Use free to deallocate heap objects
        -> give pointer to object we want to delete
        -> do not attempt to free global or stack objects!

Other useful functions


void *calloc(size_t num_elems, size_t elem_size);
        - allocates space and clears all the bytes (sets all bits to 0)
        - number of bytes allocated is num_elems * elem_size
        - avoids any problems with forgetting to initialize values
                - allocate an array or struct with all elements/fields set to 0

        - having two arguments may be to prevent integer overflow if you are
                allocating a lot of memory
                - but then why does malloc only take one argument? unclear

        int *A = calloc(100, sizeof(int));
                // allocate space for 100 integers and set them all to 0

        - why have calloc and malloc?
                malloc is faster because it doesn't need to write to the allocated space
                        i.e., maybe you are going to immediately initialize to non-zero values
                calloc is safer because it ensures a known value of the data

        - calloc is essentially just malloc + memset


void *memset(void *p, int c, size_t n);
        - sets the value of bytes in memory
                - p - pointer to start of object
                - n - the number of bytes in the object
                - c - actual byte (char) to write
                                only the lower 8 bits of c will be used

        - may be faster than writing a loop yourself
                - possibly takes advantage of OS operations; hardware accelleration

        - returns the same pointer it was given


void *realloc(void *p, size_t size);

        - changes the size of a heap object
                - shrinks in place
                - grows in place if possible, or allocates new space & moves data

        arguments:
                p - pointer to object we are resizing
                size - number of bytes we want the object to be

        returns pointer to object - may have moved!
                for this reason, you should not use the old pointer value after a call
                to realloc

                - realloc is only safe for objects with a single reference

recall: ArrayList in Java
        array that you can grow

We can implement ArrayList in C using a pointer and two integers

// global variables
int *list, size, used;

        // to have multiple array lists, create a struct that holds these

// set up array list: allocate space for data, set variables
// (essentially a constructor)
void init(int init_size)
{
        size = init_size;
        list = (int *)malloc(size * sizeof(int));
                // cast is unnecessary; just making sure I have the right pointer type
        used = 0;
}


// to append onto the array, we increase used
        // used: number of array elements "in use" (initially zero)

        // if used == size, we use realloc to make the array bigger

// add value onto the end of the array list
        // make additional space if necessary
void append(int i)
{
        if (used == size) {
                size = size * 2;
                list = (int *)realloc(list, size);

        }
        list[used] = i;
        used++;
}

int remove()
{
        if (used > 0) {
                --used;
                return list[used];
        }

        return 0; // or throw an error or something
}



void *memcpy(void *dest, void *src, size_t bytes);

        - copy data from one object to another
        arguments:
                dest - where data will go
                src  - where data comes from
                bytes - how many bytes to copy

        returns the destination pointer

        - you must ensure that src and dest point to objects of the correct size
        - src and dest must not overlap

        - could be implemented as on O(n) loop, or use OS/HW tools for speed

        - memcpy copies bits: does not care about types
                - does not convert values

        - you must ensure that src and dest point to appropriately typed data


        something like
                char foo[] = "Hello";

        is equivalent to doing;
                char foo[6];
                memcpy(foo, "Hello", 6);


void *memmove(void *dest, void *src, size_t bytes);

        - same as memcpy, except that source and destination may overlap
        - may be less efficient than memcpy
                - may be more efficient than writing a loop yourself

        example: removing elements from the start/middle of an array

                int a[100];
                ...
                memmove(a, &a[1], 99 * sizeof(int));
                        // copy 99 integers from a[1] .. a[99] into a[0] .. a[98]
                                // note that a[99] will retain the same value

        does a = &a[1] work?
                - no: compiler will not let us assign to an array variable

        what about pointers?

        int *p = malloc(100 * sizeof(int));

        p = &p[1];   // allowed, but now we don't have a pointer to the original object
        // unless we do free(p - 1);

        int *q = p;

        q = &q[1];
        q = q + 1;
        ++q;
                // be careful when doing pointer shenanigans!
                // if you lose the pointer to a heap object, you won't be able to free it


char *strcpy(char *dest, char *src);
        - similar to memcpy, but we don't specify a size
                - instead, we copy from src until we reach an all-0 byte ('\0')

        - still must ensure that dest and src do not overlap
        - still must ensure that dest is large enough to hold data

char *strncpy(char *dest, char *src, size_t n);
        - copies up to n bytes from src to dest
                - stops after '\0' byte, or after n bytes

        - note that strncpy may result in a non-terminated string, if src is longer
          than n bytes
                - you can explicitly set the last byte to '\0' if there is a problem

        char foo[100];

        strncpy(foo, some_string, 99);  // copy up to 99 chars
                // adds null terminator if some_string is less than 99 chars long
        foo[99] = '\0';   // ensure that foo is null-terminated
                // redundant if some_string is < 99 chars long, but that is okay



Deciding when to use memcpy, memmove, strcpy, strncpy is fairly straightforward

        memcpy and memmove specify exactly how many bytes to move
        strncpy may copy fewer bytes
        strcpy is potentially more efficient than strncpy

        in general: pay attention to how long your strings may be
                are you copying data from disjoint objects or within an object


Example: duplicating a string

char *str_dup(char *str)
{
        char *dest = malloc(strlen(str) + 1);
                // allocate space for the whole string, plus null terminator

        strcpy(dest, str);

        return dest;
}

char *str_dup(char *str)
{
        int len = strlen(str) + 1;
        char *dest = malloc(len);
                // allocate space for the whole string, plus null terminator

        memcpy(dest, str, len);
                // faster than strcpy because it doesn't need to look for '\0'
                // dest is new object, so it can't overlap with str

        return dest;
}

// or use strdup from Posix library

Use man for more details
documentation is also available on-line


C Pre-Processor & separate compilation
--------------------------------------

The C Pre-Processor (CPP) runs before the compiler starts compiling
- fancy find/replace stage that modifies your source code

- can be used for code generation or for defining constants

Lines starting with # are preprocessor directives

#include <std_file.h>
#include "my_file.h"

        #include copies the content of a file into your file
        <filename> says to look for the file in the "include" directory
                (e.g., /usr/include)
        "filename" says to look relative to the current directory

        Primarily used to include "header files"
                typically end in .h
                include function prototypes, struct declarations, typedefs, etc.

        When I write
                #include <stdio.h>
        CPP will copy the definitions in stdio.h into my file
                now I can call printf(), etc, without needing to declare them

        Typically at the top of a file, but can occur anywhere
                but included text is included at that point

        Note that this is different from linking, but related
                include brings in declarations from other files
                linking connects our code to definitions from libraries

        #include "something.txt"
                // works fine

#define MACRO optional_value

        declares a "macro", which is used for text substitution
        i.e., replacing a constant with its value

        #define MEMSIZE 100

        ...

                int mem[MEMSIZE];  // after preprocessing, MEMSIZE will be replaced with 100

        Using macros allows us to keep things consistent
                If you see 100, you might not know what it means
                MEMSIZE tells you

        If we change MEMSIZE, we can do it in one place and all references will update

        Any time you have numbers other than 1 and 0, consider defining a macro

                MEMSIZE * 2
                        will be replaced with
                100 * 2
                        and the compiler will turn that into
                200


Next time:
        macros with arguments
        conditional compilation

        also: file descriptors
```