```
CS 214 / 2021-05-03
===================

Final exam:
    May 10, 4:00 PM - May 11 4:00 PM
        approx. 2 hours (may change)
    offered through Sakai
    primarily multiple choice
        -> "don't know" gives you 25% credit


Review of topics
----------------

- data types
    - determining size of data: sizeof()
        sizeof(int) sizeof(struct addrinfo)
        sizeof(int *)  sizeof(struct addrinfo *)

        sizeof(variable)  <- don't be confused!

        int array[20];
        sizeof(array) == 20 * sizeof(int)

        int *array = malloc(20 * sizeof(int));
        sizeof(array) == sizeof(int *)

    - strings vs char arrays
        strings always end with a terminator ('\0')
        character arrays may have no terminator or many terminators
            the difference is how they are used

            when we call read() we get raw data; not necessarily a string

        strcpy(), strcmp(), strdup(), strlen(), strcat()
        strncpy(), strncmp(), strndup(), strnlen(), strncat()
            - the "n" variants take a maximum length
            - useful if we can't guarantee the presence of a terminator
            - useful if we aren't sure destination of strcpy() is big enough

    - creating types: struct, union, enum, typedef
        struct bundles together multiple values
        union gives us a choice between types - hard to use correctly
        enum - fancy integer constants
            enum cardinal_direction { north, south, east, west };
        typedef - abbreviate type names
            typedef void *(*thread_fun)(void *);

            now we can write
                int pthread_create(..., thread_fun, ...);

- pointers
    - size of object being pointed to
    - assigning to pointer variable vs copying data
        int *p;
        p = q;  // makes p point to the same int as q

        *p = *q;  // copies the value q points to into the place p points to

        char *msg = "Hello";
        msg = "Goodbye";  // changes which string literal msg points to
        strcpy(msg, "Goodbye"); // segmentation violation!
            // msg points to a string in the data segment (read-only memory!)

        char buf[] = "Empty";  // buf names an array of 6 chars
        buf = "Full";   // not allowed!
        strcpy(buf, "Full");  // copies characters to buf

        char *nothing;
        strcpy(nothing, "Something");  // wrong! nothing does not point to anything

        char *something = malloc(10);
        strcpy(something, "Something");  // ok, because we allocated enough space

        char *somethingelse = malloc(10);
        somethingelse = "Something";  // memory leak!
            // we no longer have a reference to the 10-byte object from malloc()
            // therefore, we have no way to free() it

    - dereferencing (*)
        left side: *p = x;  // writes to the object p points to
        right side x = *p;  // writes the value that p points to
    - address-of (&)
        &variable           - address of a variable
        &array[index]       - address of specific item in array
            array[0]  - value of first element
            array     - address of array / address of first element
            &array    - address of array / address of first element
            &array[0] - address of first element

            p[n] == *(p + n)
                - prefer array index notation to pointer arithmetic

        &struct_ptr->field  - address of field in structure pointed to by struct_ptr

    - arithmetic: incrementing/decrementing pointers
        some_type *p = malloc(sizeof(some_type) * n);

        p      - address of first item / object that p points to
        p + 1  - address of value after the one that p points to

        *p      - value of object that p points to / first item in array
        *(p + 1) - value of object after the one that p points to
            - same as p[1]
            - also the same as 1[p], if you like confusing people

    - contrast with array variables
        array variables are always static or on the stack
        pointer can point to anything

        array variables cannot be reassigned
        pointers may be freely reassigned

        compiler knows how big the array for an array variable is
        only you know how big an array a pointer points to

    - void *
        point to anything; effectively the same as char *
        C automatically casts pointers to and from void *
        we can't meaningfully dereference them
            - we cast to the right kind of pointer and then use that
            - compiler assumes that what we are doing is correct
            - no guaranteed behavior if we accidentally cast to the wrong type

- variables
    - scope/visibility
        - associated with a particular block or the top-level
        - visible after being declared
        - variables in an inner scope "shadow" variables with the same name in outer
            scopes


            int foo; // global

            void fun(void)
            {
                int foo;  // local; shadows global foo (no way to refer to global foo)

                foo = 5;  // refers to local foo, not global foo
            }

            void fun2(void)
            {
                foo = 5;  // refers to global foo
            }

    - binding with object
        top-level (global) variables bind to static (process-lifetime) objects
        local variables are usually stack (function-lifetime) objects
            static local variables have process lifetime but local scope

    - initialization vs assignment

        int a[4] = {1,2,3,4};  // array initializer
        a = {1,2,3,4};  // syntax error (cannot use initializer syntax for assignment)

        int *p = "Foo";  // pointer initializer (p points to string literal)
        p = "Foo";  // pointer assignment (p points to string literal)
        *p = "Foo";  // type error (*p expects char, but "Foo" is char *)

- objects: anything stored in memory
    - lifetime: static, function, arbitrary
        static - stored in data segment; exists while process runs
        function - stored in stack; created when function starts, destroyed when function exits
        arbitrary - stored in heap; created by malloc(), destroyed by free()
    - location: data segment, stack, heap
    - direct reference (variable name) vs indirect reference (pointer)
    - malloc(), calloc(), realloc(), free()
        - avoiding leaks
        - detecting errors

        p = realloc(p, new_size); // risky! if realloc() fails, we lose our pointer

    - memcpy(), memmove(), memset()

- file IO
    - buffered operations (FILE *, fopen(), fprintf(), etc.)
    - non-buffered operations (int, open(), write(), etc.)
    - modes: read, write, append, read/write
    - file pointer/cursor: keeps track of where we last read/wrote
        - control with lseek(), fseek()
    - use of file IO for non-files: pipe(), socket(), accept()
        - some operations restricted to specific kinds of file
    - opendir(), readdir(), closedir(), struct dirent
    - stat(), fstat()
    - dup(), dup2() - additional file descriptor for existing open file

- preprocessing, compilation, and linking
    - what goes in a header file
    - #include vs linking
    - #define macros

- processes
    - process ID
    - fork() - create a child process
    - wait() - wait for a child process to terminate
    - zombie process, orphan process, zombie-orphan process
    - execl(), execv()  - change the program a process is executing
    - many things shared between parent and child process
        - what happens if we fork() when a file is open?
    - many process attributes are preserved when we exec()

- signals
    - setting a signal handler
        - signal(), sigaction()
    - blocking signals

- threads
    - pthread_create(), pthread_join(), pthread_detach()
        - if nothing joins or detaches a thread, it becomes a zombie when it stops
    - coordination
        - mutex, condition variable, barrier, semaphore
    - deadlock: 4 necessary conditions
        - mutual exclusion
        - hold and wait
        - no preemption
        - circular wait

- file system
    - inodes
        - use of indirection to allow:
            1. constant size of inode
            2. minimal wasted space for small files
            3. possibility to represent very large files
    - directories (association of names to inodes)
    - paths
        - paths-inode relation is many-to-one!
    - file permissions/modes

- The command line
    - running programs: bare name vs path
        $ program_name
            use search path to find program
        $ path/to/program_name
        $ ./program_in_current_dir
            specifies program; no need to search

    - file globs
    - controlling stdin/stdout/stderr: pipes, redirecting
        cat file | ./ww 20
            output from first program sent as input to next program
        ./program < input
        ./program > output

    - utility programs
        - man
        - cat, more, less, head, tail
        - file, wc
        - cmp, diff
        - grep
        - ps, top, jobs
        - tee
        - echo

    - file management utility programs/commands
        - ls, cp, mv, rm
        - cd
        - mkdir, rmdir
        - chmod, chown

- networking
    - 7-layer OSI model
    - 4-layer internet model
    - addressing at various layers
        link layer: identify specific machine on local network (e.g., MAC)
        network layer: identify machine globally (IP address)
        transport layer: identify process on machine (IP address + port)
        application layer: identify things relevant to application (e-mail address, URL)
    - not all devices interact with the entire stack
        - switches and hubs live at link level; invisible to network and above
        - routers live at network layer; mostly invisible to transport and above
        - user programs live in application layer
            (sockets talk to transport layer but do not expose details)

- sockets
    - listening vs connection sockets
        listening - can use accept()
        connection - can use read(), write()
    - socket(), bind(), listen(), accept()
    - socket(), connect()
    - domain name vs IP address; getaddrinfo(), getnameinfo()
    - what can go wrong?


Remember to review the course and your recitation on SIRS!
```