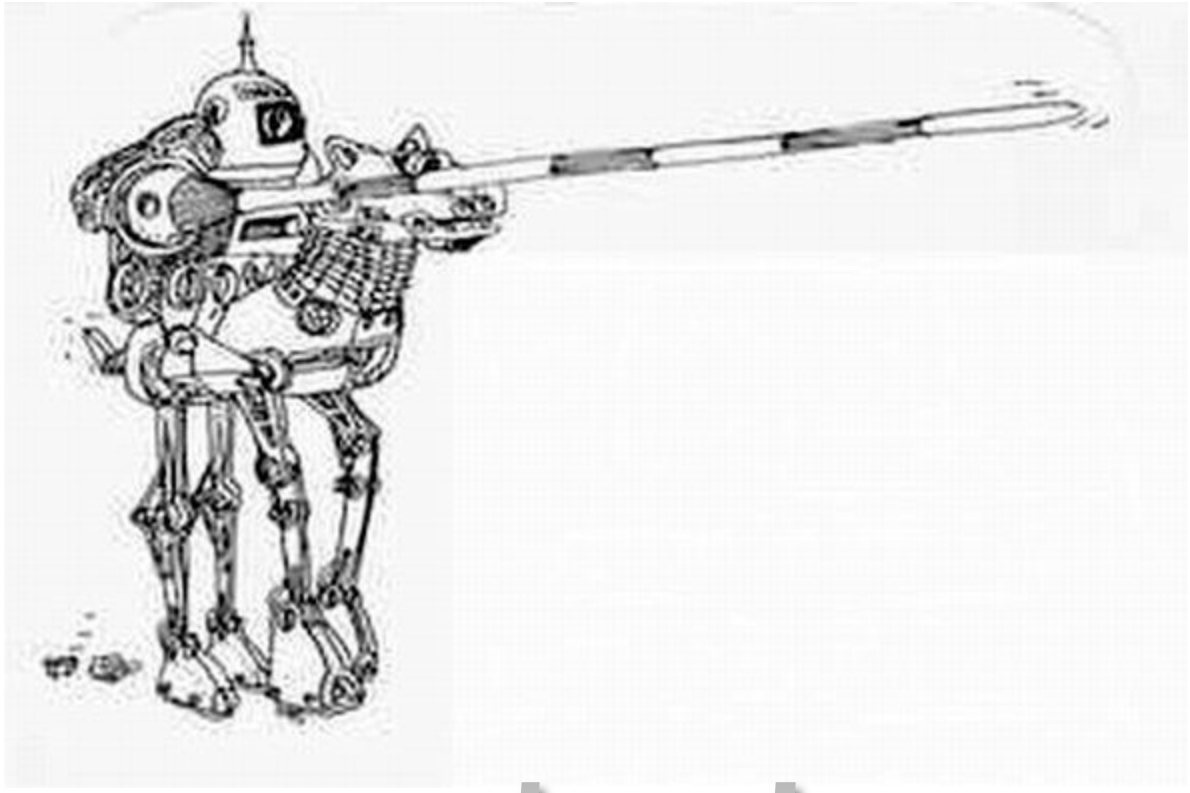


C Programming Language

Objectives

- Basic knowledge of C and C++ should have been taught in an introductory class on programming
- In the following we focus on some finer points of C:
  - External declarations
  - Pointers
  - Dynamic data structures using pointers
  - Function pointers

Extern Declarations

my_incl.h

```
extern int my_var;
```

prog1.c

```
#include “my_incl.h”
void f()
{
        my_var = 1;
}
```

prog2.c

```
#include “my_incl.h”
int my_var;
void g()
{
        my_var = 0;
}
```

External Declarations

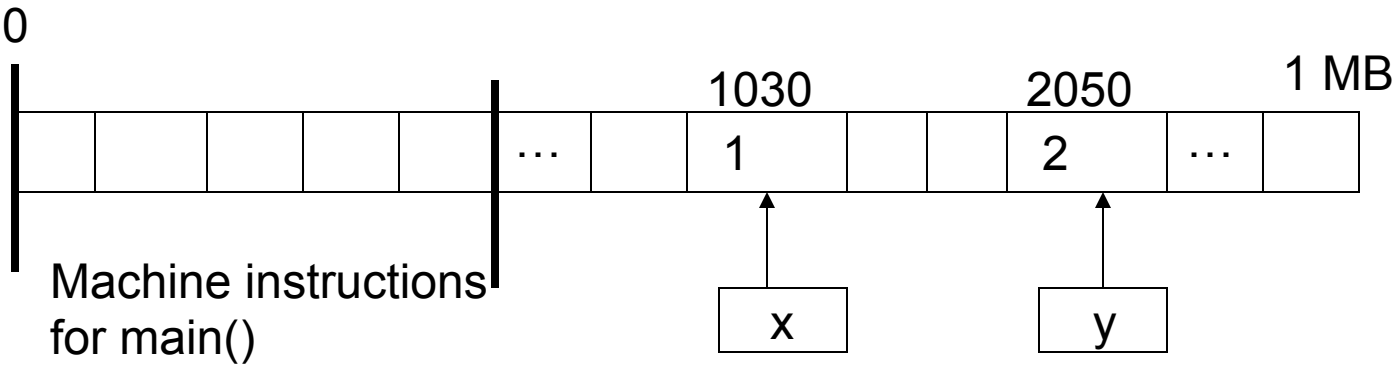
- `my_var` is an external identifier
- It is introduced to the compiler by `extern int my_var;`
- The compiler does not yet allocate memory for `my_var`, but programs can start using it (e.g. in `prog1.c`)
- Variable must be declared exactly once in one of the translation units (e.g. `prog2.c`). With the declaration the compiler allocates memory
- The linker resolves all external references (e.g. `my_var` in `prog1.c` refers to the declaration in `prog2.c`)
- Note that there is a difference in C and C++: in C it is permissible to have a global symbol declared more than once. The linker will *not* complain about this.

Memory allocation

```
int x;
int y;

void main()
{
        x = 1;
        y = 2;
        …
}
```

- for each variable declaration, the compiler allocates memory
- Variable `x` is stored at address 1030
- Variable `y` is stored at address 2050
- The address operator "`&`" determines the address. E.g. `&x` is 1030

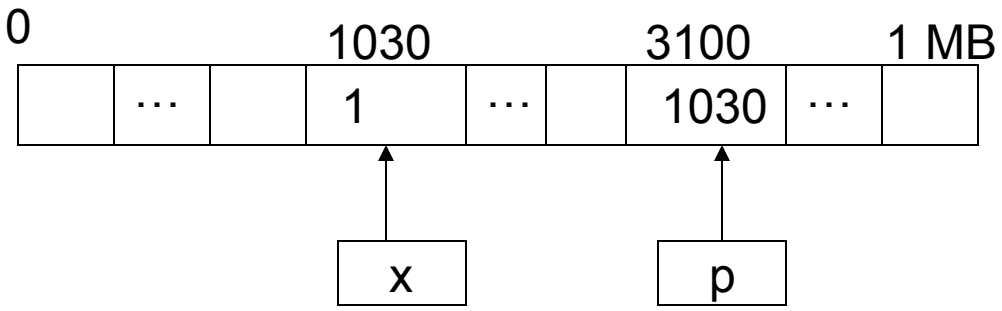


Pointers

```
int x;
int *p;

void main()
{
        x = 1;
        p = &x;
        …
}
```

- `p` is a pointer to an integer
- `p` as a global variable is also allocated memory by the compiler (e.g. `&p` is 3100)
- `p = &x;` means that p is assigned the address of `x`
- Content of memory location 1030 is 1
- Content of memory location 3100 is 1030

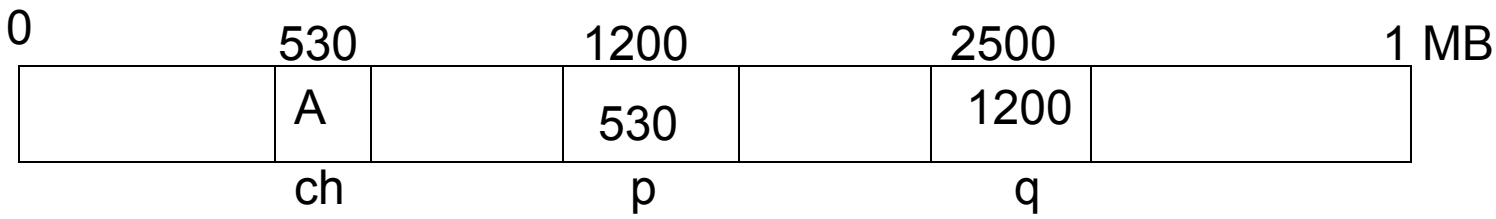


Pointers

```
/* unix */
void main()
{
    char ch = 'A';
    char *p = &ch;
    char **q = &p;
    printf("%c\n", ch);
    printf("%p\n", p);
    printf("%p\n", q);
    printf("%c\n", *p);
    printf("%p\n", *q);
    printf("%c\n", **q);
}
```

Output

```
A
530
1200
A
530
A
```



Casting

- During an assignment `x=y;` the type of `x` has to be compatible with the type of `y`
- If this is not the case, the C-compiler will issue an error.
- E.g.
```
char* c;
c = 1000;
```
will result in a compile time error, because the type of `c` is `char*` and the type of 1000 is `int`.
- When you know what you do, you can short-circuit type checking.
- This is called a cast.
- E.g.
```
char* c;
c = (char*) 1000;
```
- This tells the C-compiler to assume that the type of 1000 is `char*`

Using Pointers

```
void kernel_main()
{
        char* screen_base = (char *) 0xB8000;
        *screen_base = 'A';
}
```

We can use pointers to write to memory-mapped I/O devices. The program above will write an 'A' to the top left corner of the screen. 0xB8000 is the address of the top-left corner of the video screen. More on this later. Note that this program will very likely not work under Unix or Windows because of memory protection (I.e., you cannot just write to address 0xB8000).

Dynamic Data Structures

- Dynamic data are used when the number of data items are not known *a priori*
- Program usually uses `malloc()/free()` in C and `new/delete` in C++ to create sufficient number of data items at runtime
- As will be discussed later, in OS hacking we don't have those functions!!!
- We have to solve the problem by doing a pseudo dynamic memory management
- Solution #1: do an array where each element has a 'used' flag. If a new element is needed, look for one where `used==FALSE`
- Solution #2: use a single-linked list (see next slides)

Part 1: Data Structures

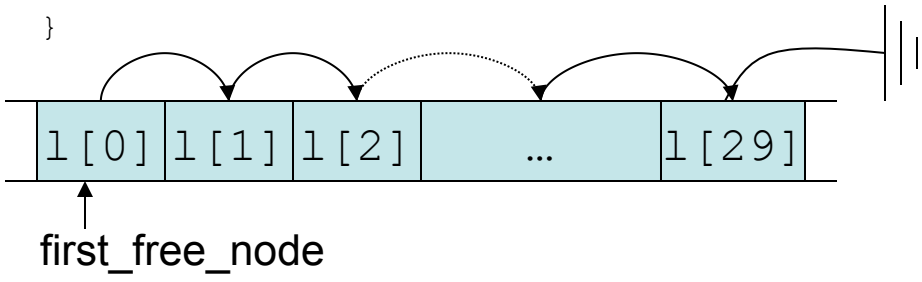
```
typedef struct _Node
{
      int x;
      struct _Node* next_free;
} Node;

Node l[30];
Node* first_free_node;
```

- Compiler allocates array with 30 data items (this is how we get around using `malloc()`)
- I.e., there can be a maximum of 30 data items at any given time
- Each data item contains a pointer to the next available data item

Part 2: Initialization

```
void init ()
{
    int i;
    /* Create single linked list
       of the first _29_ elements */
    for (i = 0; i < 29; i++)
        l[i].next_free = &l[i+1];
    /* 0 terminates the list at
       the last element */
    l[29].next_free = (Node *) 0;
    /* Initialize the pointer to the
       first free node */
    first_free_node = &l[0];
}
```



- Before the data structure can be used, it needs to be initialized once
- Function `init()` creates a single linked list of the first 29 elements of the list.
- Global variable `first_free_node` is initialized to point to the first array element
- Only available (I.e., free) elements are on the linked list.

Part 3: Allocating a Data Item

```
Node* alloc_data_item ()
{
    Node* tmp;
    tmp = first_free_node;
    first_free_node = tmp->next_free;
    return tmp;
}
```

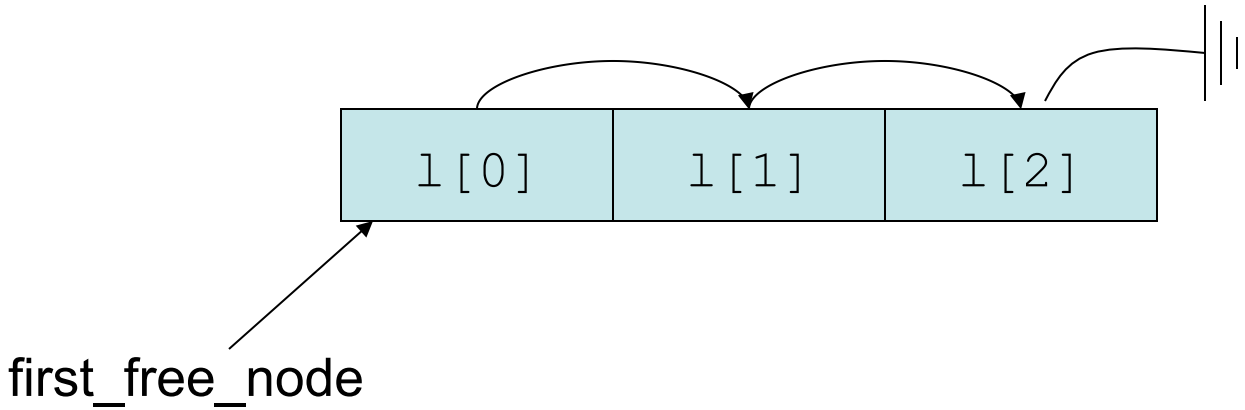
- Function `alloc_data_item()` returns a free data item from the list of available data items
- The next free data item is determined via global variable `first_free_node`
- I.e., the next free element is taken from the head of the single linked list
- If no more data items are available, the function returns 0
- The caller is responsible to check that the value returned is not 0

Part 4: Deleting a Data Item

```
void delete_data_item (Node* elem)
{
    elem->next_free = first_free_node;
    first_free_node = elem;
}
```

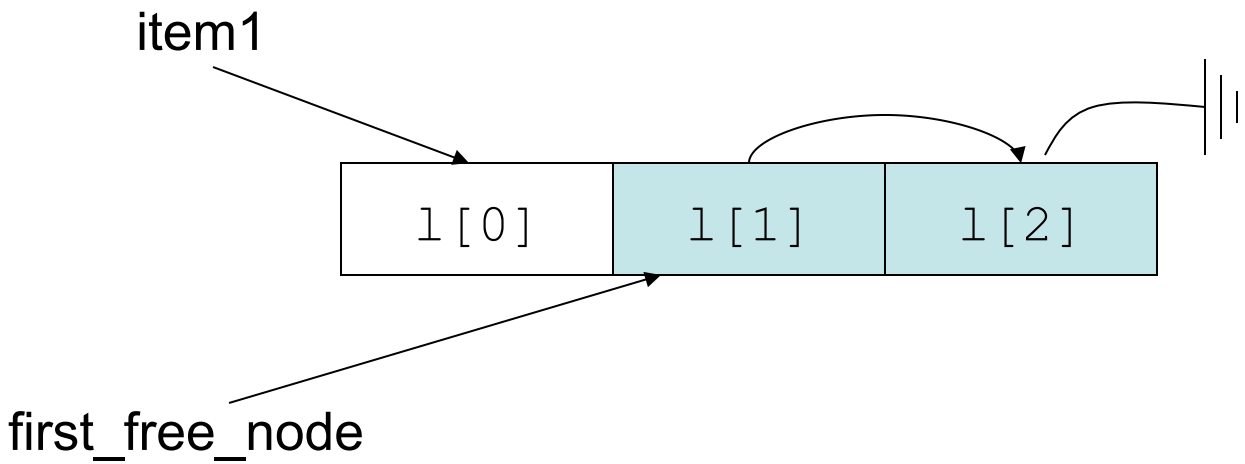
- A data item is deleted by returning it to the free list
- This is done by adding the data item to be deleted to the beginning of the single linked list

Example



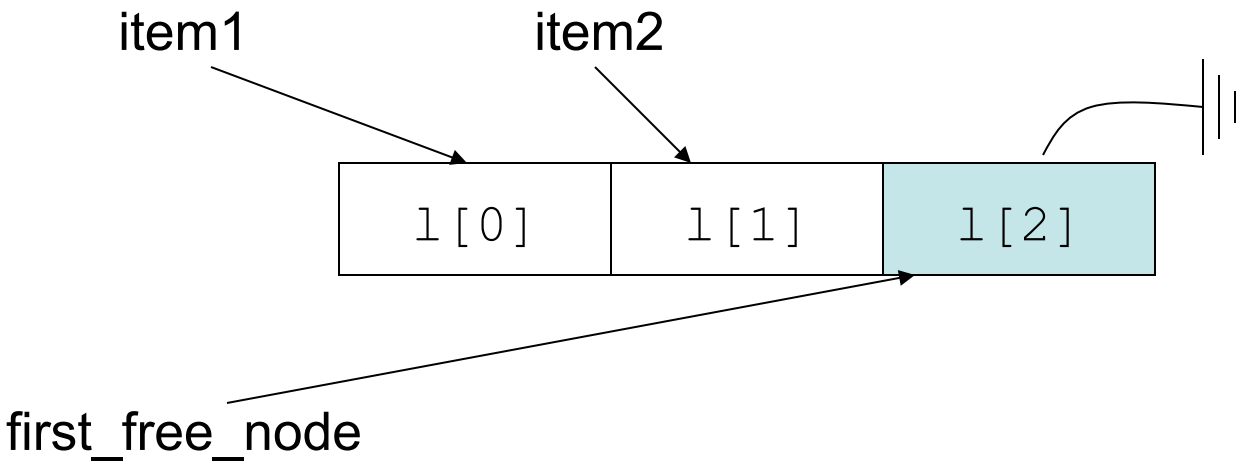
Initial setup: all elements are on free list.

Example



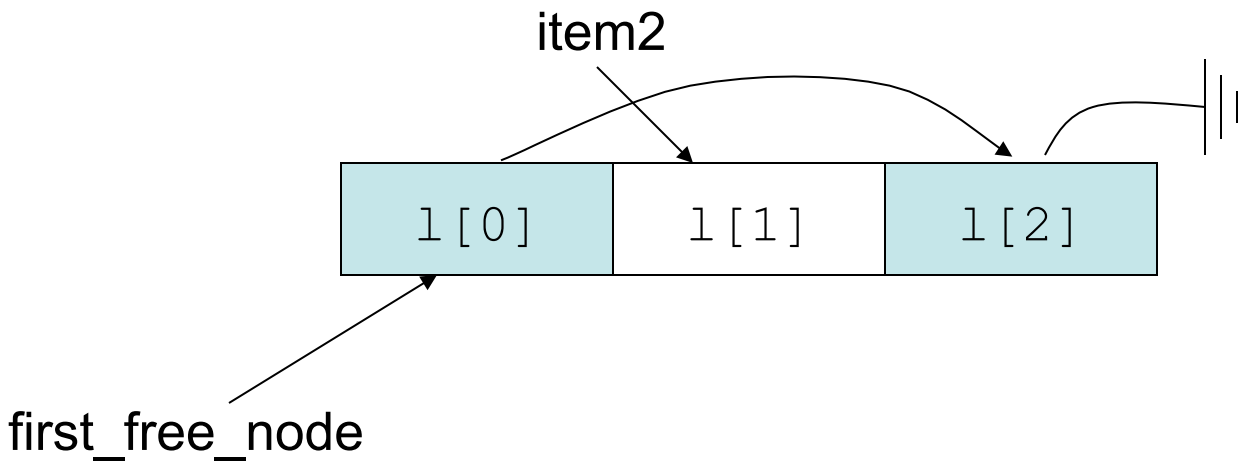
Step 1: request a data item.

Example



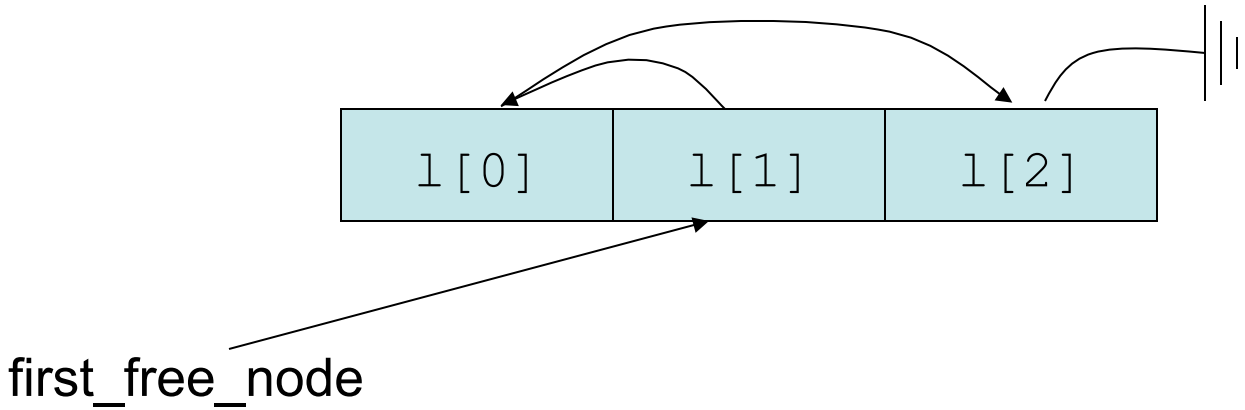
Step 2: request another data item.

Example



Step 3: release item1.

Example



Step 4: release item2.

Pointers to Functions

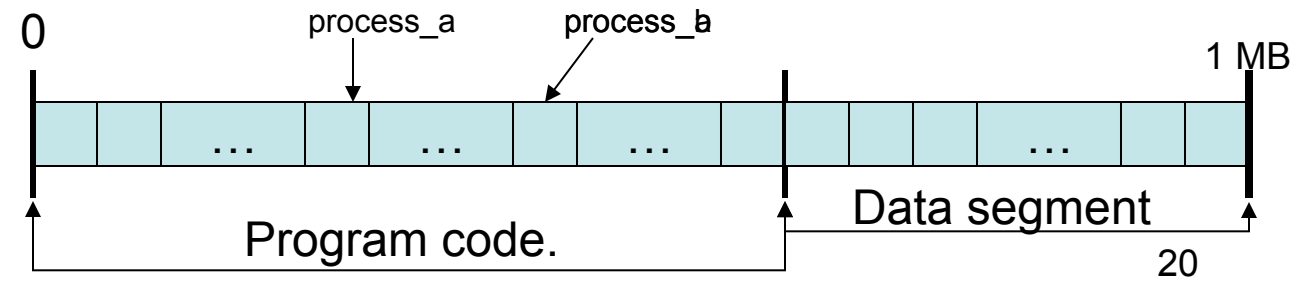
```
void process_a (int x)
{
    printf ("Process a got %d\n", x);
}

void process_b (int x)
{
    printf ("Process b got %d\n", x);
}

void call (void (*func)  (int), int arg)
{
    (*func) (arg);
}
void main()
{
    call (process_a, 10);
    call (process_b, 20);
}
```

Produces output:

```
Process a got 10
Process b got 20
```



Pointer to Functions

- `void call (void (*func)(int), int arg)`
  - `func` and `arg` are formal parameters.
  - `void (*func)(int)` is the first argument
  - `int arg` is the second argument

- The first argument to `call` expects a pointer to a function with return type void and one input parameter of type integer.
- The pointer to a function is actually the address of the first machine instruction that belong to the implementation of the function.
- `call (process_a, 10)` passes two actual parameters:
  - pointer (i.e. address) of function `process_a` (note: `process_a` is not called at this time)
  - 10
- `(*func )(arg)` actually calls the function pointed to by `func`. The function to be called is passed the actual parameter `arg`.

Pointers revisited

```
void say_hello()
{
    printf("Hello!\n");
}

void main()
{
    char *p = (char *) say_hello;
    say_hello();
    *p = 0xc3;
    say_hello();
}
```

Output:

```
Hello!
```

- `0xc3` is the Intel x86 machine instruction for `RET` (Return from Subroutine)
- Overwriting the first instruction of `say_hello()` with `RET` will cause the function to exit immediately
- This program will cause a security violation under Unix, but it would work under TOS.

Assignment 1

- Detailed explanation on course web site
- Get TOS source code
- Implement the functions located in `tos/kernel/stdlib.c`:
  - `k_strlen()`
  - `k_memcpy()`
  - `k_memcmp()`
- Run test cases by calling make target
  `make host-tests`
- Not too much coding -- get familiar with TOS source code, review pointers
- Hint: the above functions should behave identical to the standard C-library functions `strlen()`, `memcpy()`, and `memcmp()`. Use the Unix man-pages to understand their behavior.