Today's lecture

- **Continue exploring C-strings**
  - **Under the hood: sequence of chars w/ terminating null**
    - Review implementation of strcpy, strncpy
  - **As abstraction: client of string.h functions**
    - Write pig latin conversion program
- **Pointer mechanics**
  - **Pointer types**
  - **& and * operators**
  - **Pointer arithmetic**
- **C arrays**
  - **Array declaration**
  - **Array indexing**
  - **Relation to pointers**

/afs/ir/class/cs107/samples/lect4

```
char *my_strcpy(char *dst, const char *src)
{
    char *result = dst;
    while ((*dst++ = *src++)) ;
    return result;
}

char *my_strncpy(char *dest, const char *src, size_t n)
{                                              // from man page
    size_t i;

    for (i = 0; i < n && src[i] != '\0'; i++)
         dest[i] = src[i];
    for ( ; i < n; i++)
        dest[i] = '\0';
    return dest;
}
```

Let's code!

`/afs/ir/class/cs107/samples/lect4/pig.c`

Pig latin: be => ebay
trash => ashtray
one => oneway

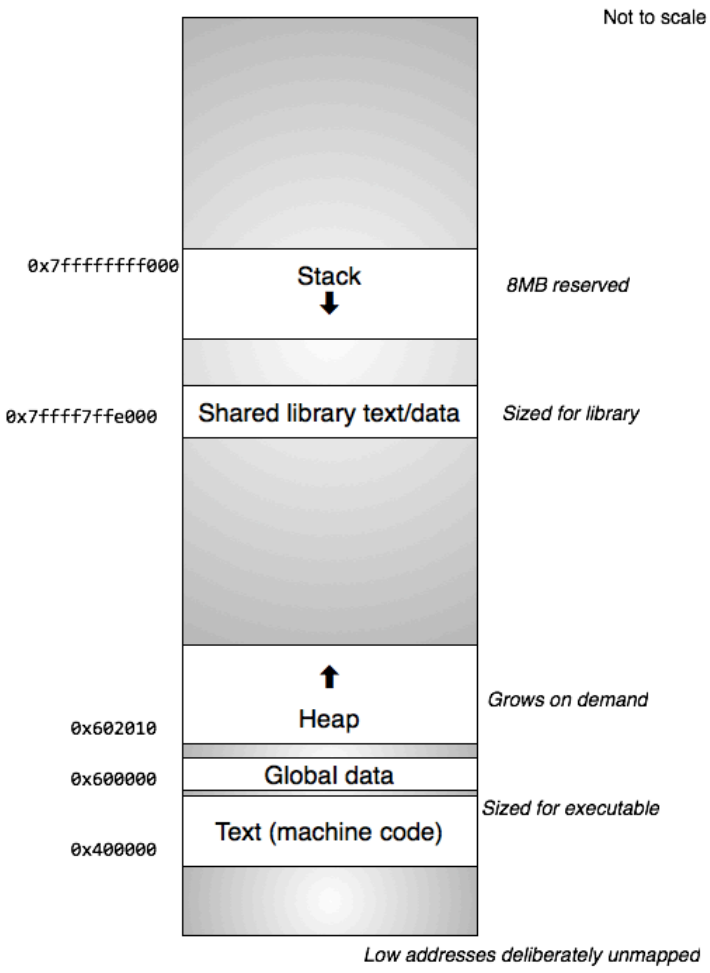
Helpful string.h functions to consider:

- strcspn
- strcat, strncat

Why pointers?

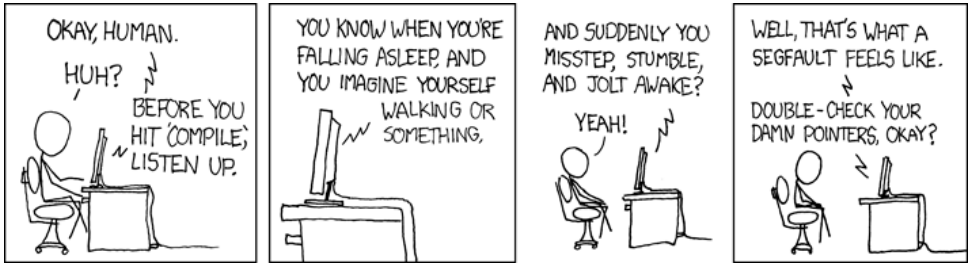
- **Pointers facilitate sharing, efficiency**
  - Linked data structures (lists, trees, graphs)
  - Efficiently pass/return without making additional copies
  - Avoid redundancy in data structures, link to one version of truth
- **In C, pointers are ubiquitous**
  - Access specific memory location by address (e.g. system peripherals)
  - Use for pass by reference (manual, not automatic)
  - Arrays are implemented as pointers
    - C-strings are arrays of char
  - Dynamic memory accessed via pointer
  - Function pointers

Meet your address space



Divided into segments by purpose

Access to invalid memory location results in "segmentation fault"



Pointer basics

- Address: is a memory location, represented as unsigned long
- Pointer: is a variable that holds an address
- Data stored at address is called the "pointee"

  (my made-up word; placeholder for actual type such as char/int/struct student)

- & address-of

  Apply to variable to get its address, i.e. &var is location in memory where var is being stored
- * dereference

  Apply to address/pointer to access pointee

  Same syntax to read and write, *p = *q
- Can manipulate address numerically

  Limited to equality/relational ops and add/subtract offset

C pointer types

- **C type system**
  - Each variable declared with type
  - Type determines size of storage and valid operations
- **Operations required to respect that type**
  - Can’t multiply two char *s, can’t deference an int
  - Co-mingle distinct types accepted if "sensible" automatic conversion exists
- **Pointer variables distinguished by type of pointee**
  - Dereferencing int* yields an int, dereferencing char * yields a char
  - Pointer arithmetic on int* scales by sizeof(int), on char * scales by sizeof(char)
- **Types are compile-time (no runtime checking)**
  - If use typecast to subvert CT check, no RT error on mismatch

C arrays

- **Array is sequence of elements, homogenous type**

  `int arr[5];`

  Allocates space for 5 ints, contiguous memory, indexed from 0 to 4

- **Subscript to access individual elements**

  ```
  arr[0] = 72
  arr[1] = 45
  ```

- **What happens if subscript invalid?**

  ```
  arr[99] = 10
  arr[-1] = 3
  ```

- **Can assign array to pointer — what does this do?**

  `int *ptr = arr;`

  Use of array name "decays" to address of first element, e.g. `arr` is equivalent to `&arr[0]`

  Array contents not copied on assignment, `ptr` assigned address in memory where `arr` stored

  ptr and arr are now "aliases", refer to same memory

Pointer arithmetic, array indexing

- **Array indexing is "syntactic sugar" for pointer arithmetic**

```
ptr + i      <=>    &ptr[i]
*(ptr + i)   <=>    ptr[i]
```

- **Arithmetic scaled by sizeof(pointee)**
  - ptr + 1 adds one if ptr is char *, what if ptr is int *?
  - What happens if you cast to different size pointee before arithmetic?
- **Either syntax on either pointer or array**
  - Can use subscript on pointer variable or pointer arithmetic on array
  - Access to nth element in either always takes into account size of pointee

Pointer versus array

- **Similar…. but not identical**
  - Consider C type system & draw pictures to visualize how underlying reality is same/different
- **Operations in common**
  - Dereference, pointer arithmetic, array indexing
- **Difference in declaration**
  - What space is allocated and what does memory diagram look like?
  - Array declaration set aside space for N elements
  - Pointer declaration is single variable to hold address
- **Difference in operations**
  - Can reassign the pointer to hold a different address, not so with array
    - arr = NULL doesn't even compile — why not?
  - What is sizeof(ptr)? what is sizeof(arr)?