

**CSci 350**  
**A Computing Perspective on GIS**

# Pointers

- A pointer is an address
- The `&` and `*` operators
  - Any variable `a` has an address
    - `&a`
  - Any valid address `p` has a content
    - `*p`
    - `*p` is called “dereferencing” the pointer
  - `int a;`
  - `*(&a) = a`
  - `int *p`
  - `//assume p is a valid address`
  - `&>(*p) = p`
- Dereferencing an invalid address `====>` seg fault

# Simple pointer example

```
main() {  
    int a;  
    int *p, *q;  
    //p, q are variables that point to ints; they do NOT point to anything yet  
    p = (int*) malloc(sizeof(int));  
    //now p points to a valid memory location  
    *p = 10 ;  
  
    q = (int*) malloc(sizeof(int));  
    //now q points to a valid memory location  
    *q = *p; //copy the value of *p to *q  
    ...  
    free(p);  
    free(q);  
}
```

# Static and dynamic data

- all static variables live on stack
  - the stack data is called static data; its size needs to be known at compile time
  - e.g. `int a[10]` is a static array
  - the lifetime of a stack variable is the duration when the function is active
- `malloc()` allocates space on the heap
  - the heap data is called dynamic
  - the allocated heap space needs to be freed with `free()`

# Pointers and arrays

```
int *p;  
p = (int*) malloc(n * sizeof(int));
```

- p is the same as an array (if allocating an array of unknown size were possible)
- `int[n] p;`
- an `int[]` is an `int*`
- the `[]` operator is implemented by the language on top of pointers to manipulate arrays easily
  - `p[0] = *p`
  - `&p[0] = p`
  - `*(p+sizeof(int)) = p[1]`

# Pointer arithmetic

- `int *p;`
- `float *p;`
- `double *p;`
  
- pointer arithmetic
  - `T *p;`
  - `p + 1` is the same as `p + sizeof(T);`
  
- `void *p;`
- `//p` can be thought of as the address of a chunk of bytes
- in order to be accessed it needs to be cast
- `((int*)p)[0]`

# Pointers and parameter passing

```
... fun (Type x)
```

- when calling fun(a), the value of a is copied into x
- if x is changed inside fun(), this change does not affect a
  
- if fun() wants to modify x, it needs to get a pointer to x
- ... fun(Type \*x)
  
- Examples:
  - if swap(int a, int b) wants to swap the values of two integers, it needs to take their addresses
    - void swap(int\* a, int\* b)
  - if fun wants to allocate an int\*, it needs to take as parameter &(int\*), which is an int\*\*
    - void allocateArray(int\*\* a, int n)

# Pointer programming guidelines

- **asserts**
  - malloc() returns NULL if it cannot allocate memory
  - always assert the pointer after a malloc()
  - always assert a pointer p before dereferencing
  - initialize pointers by setting them to NULL
- **write correctness checks and assert them**
  - assert(arrayIsSorted(a))
- **asserts may be expensive**
  - they can be turned off by compiling with -DNDEBUG
- **flushes**
  - printf is a library function that buffers its output
  - this buffer is emptied periodically and when the program terminates normally
  - in case the program does not finish normally (i.e. it segfaults) you will NOT see everything that you printed out
  - use fflush(stdout)



# Pointers caveats

- Accessing/dereferencing a non-valid address (pointer) may cause an error
  - seg fault
- Pointer errors do NOT always manifest
- Pointer errors do NOT always manifest the same way
  
- the space allocated with malloc() must be freed
  - there is no garbage collection
- no free(): memory leaks
- double free(): seg fault

# Pointers

## ■ Why use pointers?

- flexibility
- efficiency
  - provide a pointer to the data as parameters to functions, not a copy of the data
- build flexible, dynamic data structures
- precise control over allocation & deallocation

## ■ Why are pointers scary?

- error-prone
- pointer mistakes have wide variations on symptoms
- memory bugs hard to understand and debug