Pointers

Pointers Variables

- & is the address operator. It produces the address of the variable
  - printf("This is the address of x: %p\n", &x);
- Pointer variable (or just pointer)
  - Can contain the address of a memory location that holds a particular kind of data
  - NULL is a special value for pointers (numerically 0)
- Declarations
  - float x = 9.1;
  - float *xPtr;
  - int a = 7;
  - int *aPtr;
- These 2 statements place an address in each pointer variable
  - xPtr =             // stores the address of x
  - aPtr =            // stores the address of a

Indirection Operator

- * is the indirection or dereferencing operator
  - It is always followed by the name of the pointer and it returns the value in the location whose address is the pointer
- * is used in 2 ways
  - * is used when pointers are declared
    - int *p; // p will "point to" an int
  - * is used as an operator
    - *p = 9; // the location "pointed to" by p is assigned 9

Pointer Pitfalls

- Remember: Declaring a pointer variable does NOT place a meaningful address in that variable
  - int *x;
  - *x = 7; // ERROR - x doesn't have a meaningful address // and can't store 7
- If a pointer is NULL it holds no address
Meaningful Address

• How does C place a meaningful address in a pointer variable?
  o Assignment – give the pointer an address that is already allocated
    int x; int *p1;
    p1 = &x;

• Grab some unused memory and give its address to the pointer
  int *p2;
  p2 = malloc(sizeof(int));
  o If there is no available memory, p2 would be set to NULL

Pointer Assignment

• What happens when as assignment between pointers takes place?
  o Suppose *p1 contain 56 and the assignment p2 = p1; is executed?

Draw memory locations and their contents

1. int *a, *b, i = 14;
   a = &i;
   b = a;

2. float *x, *y;
   x = malloc(sizeof(float));
   *x = 4.5;
   y = malloc(sizeof(float));
   *y = *x + 2;
   printf("%1.1f %1.1f
",
         *x, *y);

3. int *aPtr, *bPtr, *cPtr;
   aPtr = malloc(sizeof(int));
   *aPtr = 4;
   bPtr = malloc(sizeof(int));
   *bPtr = *aPtr;
   cPtr = bPtr;
   *cPtr = *cPtr + *aPtr;
   printf("%d %d %d
",
          *aPtr, *bPtr,
          *cPtr);
Functions that Return Pointers

- WARNING: make sure the function returns a meaningful pointer

```c
int* doIt (int num){
    int *p;
p = calloc(num, sizeof(int));
p[0] = 99;
    return p;
}

char* find(char s[]){
    char *p;
p = s;
    while (*p != '\0')
        p++;
    return p;
}
```

Pointer Arithmetic

- Should you add pointers to pointers?

```c
float a[5] = {1.1, 2.2, 3.3, 4.4, 5.5};
printf("%1.1f\n", *(a+1));
```

- Incrementing a pointer changes the address to the next address of the same type (here, the next address for a float)
- Decrementing a pointer changes the address to the previous address of the same type

Examples

- int *p = calloc(5, sizeof(int));
  *p = 4;
p++;  // but what’s the name of the array???

- int a[5] = {1, 2, 3, 4, 5};
  int *p;
  a is the address of a[0]
  a+1 is the address of so *a is the same as
  a+2 is the address of so *(a+1) is the same as
  p = a+3;         // Could we say a = a + 3;   ?
  printf("%d\n", *p);
Questions

• What are synonyms for
  o &(*a)
  o &a[0]
  o &a[1]

• What does this code do? Trace it.

```
int i;
float x[6], *q;
q = x;
for (i = 0; i < 6; i++) {
  *q = 5 * i;
  q++;
}
```

const

- If we want a variable to remain unchanged, we can prefix its declaration with const.
  ```
  const float PI = 3.14;
  ```
- This can be done in parameter lists also
  ```
  int find(char *s) could be int find(const char *s)
  ```
- const parameters are used in many functions in `string.h`
- The use of const can enforce the principle of least privilege, i.e. the function should not change values it has no right to change!

Pointers in structs

```c
typedef struct account Account;
struct account {
  char *name;
  long number;
  double balance;
};
```

```c
struct node1 {
  Account acct;
  struct node1* next;
};
```

Or

```c
struct node2 {
  Account *acct;
  struct node2* next;
};
```