Today’s Goals

- Pointers as function arguments
- Pointers as function return value
- Arrays in terms of pointers
  - Single
  - Multi-dimensional
  - Pointer arrays

The NULL Pointer

- C guarantees that zero is never a valid address for data
- A pointer that contains the address zero known as the NULL pointer
- It is often used as a signal for abnormal or terminal event
- It is also used as an initialization value for pointers

Pass by Value

- All functions are pass-by-value in C
  - A copy is made of each parameter’s value and then the copy is passed
- Variables supplied as parameters to a function call are protected against change
  - i.e. impossible to write a \texttt{swap(x, y)} function
- Only way to modify a variable through a function is to assign the return value to that variable

Pass by Value and Pointers

- All functions are pass-by-value in C
- Pass-by-value still holds even if the parameter is a pointer
  - A copy of the pointer’s value is made – the address stored in the pointer variable
  - The copy is then a pointer pointing to the same object as the original parameter
  - Thus modifications via de-referencing the copy STAYS.

Function Arguments

- \texttt{x} and \texttt{y} are copies of the original, and thus \texttt{a} and \texttt{b} can not be altered.

```c
void swap(int x, int y) {
    int tmp;
    tmp = x; x = y; y = tmp;
}
int main() {
    int a = 1, b = 2;
    swap(a, b);
    return 0;
}
```

Wrong!

Pointers and Function Arguments

- Passing pointers – \texttt{a} and \texttt{b} are passed by reference (the pointers themselves \texttt{px} and \texttt{py} are still passed by value)

```c
void swap(int *px, int *py) {
    int tmp;
    tmp = *px; *px = *py; *py = tmp;
}
int main() {
    int a = 1, b = 2;
    swap(&a, &b);
    return 0;
}
```
Use Pointers to Modify Multiple Values in a Function

```c
void decompose(double d, int *i, double *frac) {
    *i = (int) d;
    *frac = d - *i;
}

int main() {
    int int_part;
    double frac_part, input;
    scanf("%lf", &input);
    decompose(input, &int_part, &frac_part);
    printf("%f decomposes to %d and %f\n", *int_part, *frac_part);
    return 0;
}
```

Pass by Reference

- Do not equate pass-by-reference with pass-by-pointer
- The pointer variables themselves are still passed by value
- The objects being pointed to, however, are passed by reference
- In a function, if a pointer argument is dereferenced, then the modification indirectly through the pointer will stay

Pointers are Passed by Value

```c
void f(int *px, int *py) {
    px = py;
}

int main() {
    int x = 1, y = 2, *px;
    px = &x;
    f(px, &y);
    printf("%d", *px); // will print 1
}
```

Modification of a Pointer

```c
void g(int **ppx, int *py) {
    *ppx = py;
}

int main() {
    int x = 1, y = 2, *px;
    px = &x;
    g(&px, &y);
    printf("%d", *px); // will print 2
}
```

Pointer as Return Value

- We can also write functions that return a pointer
- Thus, the function is returning the memory address of where the value is stored instead of the value itself
- Be very careful not to return an address to a temporary variable in a function!!

Example

- `x` and `y` are copies of the original, and thus what is `&x` and `&y`?

```c
int max(int *x, int *y) {
    if (*x > *y)
        return x;
    return y;
}

int main() {
    int a = 1, b = 2, *p;
    p = max(&a, &b);
    printf("%d", *p);
}
```
Arrays

- Declaration – int a[5]; a
- Assignment – a[0] = 1;
- Reference – y = a[0]; a
- Schematic representation

```
0 1 2 3 4
```

Pointers and Arrays

- Arrays are contiguous allocations of memory of the size: sizeof(elementType) * numberOfElements
- Given the address of the first byte, using the type (size) of the elements one can calculate addresses to access other elements

Name of an Array

- The variable name of an array is also a pointer to its first element.
- a == &a[0]
- a[0] == *a

Multi-Dimensional Array

```
int a[2][3];
a[0][1] = 5;
y = a[0][1];
```

Pointer Arithmetic

- One can add/subtract an integer to/from a pointer
- The pointer advances/retreats by that number of elements (of the type being pointed to)
  - a+i == &a[i]
  - a[i] == *(a+i)
- Subtracting two pointers yields the number of elements between them

Pointers Arrays: Pointer to Pointers

- Pointers can be stored in arrays
- Two-dimensional arrays are just arrays of pointers to arrays.
  - int a[10][20]; int *b[10];
  - Declaration for b allows 10 int pointers, with no space allocated.
  - Each of them can point to an array of 20 integers
  - int c[20]; b[0] = c;
  - What is the type of b?
Ragged Arrays

Arrays as Arguments

- Arrays are passed by reference
- Modifications stay

#define SIZE 10
void init(int a[])
{
    int i;
    for(i = 0; i < SIZE; i++)
        a[i] = 0;
}

int main()
{
    int a[SIZE];
    init(a);
    return 0;
}

/* equivalent pointer alternative */
void init(int *a)
{
    int i;
    for(i = 0; i < SIZE; i++)
        *(a+i) = 0;
}

Combining * and ++/--

- ++ and -- has precedence over *
  - a[i++] = j;
  - p=a; *p++ = j; <=> *(p++) = j;
  - *p++; value: *p, inc: p
  - (*p)++; value: *p, inc: *p
  - ++(*p); value: (*p)+1, inc: *p
  - *++p; value: *(p+1), inc: p

Summary

- Understand the relationship between arrays and pointers
- Understand the relationship between two-dimensional arrays and pointer arrays
- Arrays are passed by reference to functions
- Pointer arithmetic is powerful but dangerous!