Addresses and Pointers

- Remember that:
  - all variables are data
  - all data resides in memory
  - every memory location has an address
- A **pointer variable**:
  - holds the “address" of a memory location
  - usually contains the address of a **named variable**
  - sometimes an **anonymous variable** on the heap
  - sometimes an address **within a variable**, e.g. a C string (which is a character array)
  - can be used as a formal function parameter to receive the address of a variable (an ersatz "call-by-reference" mechanism)
  - here the actual parameter (addresses) is copied to formal parameter (pointers)
  - **to obtain an address**: use the **& symbol**
  - **to use an address**: use the *** symbol** (outside of a variable declaration)

Addresses and Pointers

- Compare the following two code fragments

```
int x = 1;
int y = x;
x = 2;
printf("y is %d\n", y); /* “y is 1” */
```

```
int x = 1;
int *y = &x;
x = 2;
printf("*y is %d\n", *y); /* “*y is 2” */
```

- In other words, **`x`** is a synonym for **`*&x`**

Notation

`&x` "get the address of variable x" (**referencing**)

`*x` "get the contents of the memory location whose address is stored in variable x" (**dereferencing**)

- Note that `*` can appear in a **variable declaration**
- **However, it has a different meaning in a declaration!**

```
double f = 30.0;
double *g = &f;
printf("%lf  %lf\n", f, *g);
```

Notation

- **Pointer variables** are declared in terms of other types (scalar and nonscalar)
- Often helpful to read the simpler variable declarations **right-to-left**

```
int *a;
double *f;
char *st[10];
```

- Note: In declarations the `*` is right beside and **logically attached** to the variable name
  - Declaration syntax is meant to remind programmer of the result of **dereferencing** the variable

Pointers

- Why do we need pointers?
- **Call-by-value** works well for passing parameters into functions, but:
  - What if we want values to be modified in the call function?
  - What if want to pass a large struct as a function argument?
- Functions can only return a single value in `return` statements; what if we need multiple values changed (but don't want to write a struct for this)?
  - **Call-by-reference-like** semantics would get around the limitation of a "single return value".
  - **However, C only has call-by-value semantics!**
  - (C++ has call-by-value and call-by-reference)

Example

- swap function:

```
void swap(int a, int b) {
        int temp = a;
        a = b;
        b = temp;
}
```

```
/* ... some code here ... */

void blarg() {
    int x = 2;
    int y = 1;

    swap(x, y);
    printf("x = %d, y = %d\n", x, y); /* x = 2, y = 1 */
}
```

Example (2)

- Notice that the values in Example (1) were **not swapped**
- Integers "a" and "b" were swapped within the scope of `swap()`, but the results are not visible in to calling function
- Must use pointers to swap as shown below:

```
void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

```
/* ... some code here ... */

void blarg() {
    int x = 2;
    int y = 1;

    swap(&x, &y);
    printf("x = %d, y = %d\n", x, y); /* x = 1, y = 2 */
}
```

Invalid pointers

- **C does not implicitly check the validity of a pointer!**
  - The address could be to a region of memory holding complete and total garbage...
  - ... but C will dereference the (garbage) address if told to do so.
- **It is the programmer's responsibility** (i.e, you!) to ensure that a pointer contains a valid memory address
  - avoiding **dangling pointers**
  - avoid dereferencing a pointer when you're not sure of "where it has been"
- Example, what happens?:

```
int *x = NULL;
printf("%d\n", *x);
```

  - sometimes the runtime system reports use of null pointer
- `NULL` is defined in both "stdio.h" and "stdlib.h"

Pointers and arrays

- Recall that arrays are an aggregate data type where each data element has the same "type":

```
int grades[10];
struct date_record info[50];
char buffer[100];
```

- All elements in an array occupy contiguous memory locations
- To get the address of any data element, we can use **&**:

```
5th element of "grades": &grades[4]
1st element of info: &info[0]
last element of "buffer": &buffer[99]
```

Pointers and arrays

- an important array location is usually that of the first element
- in C, an array variable name without the subscript represents the first element; recall that each element is a character

```
char  buffer[100];
char *cursor;

cursor = &buffer[0]; /* these two lines ... */
cursor = buffer;     /* ... have the same effect. */
```

Pointers and arrays (3)

- Can use pointer variables and array names (sometimes) interchangeably to access array elements:

```
int X[4];
int *p = &X[0];
p = X; /* okay */
p++;   /* okay */
X = p; /* illegal */
X++;   /* illegal */
X[1] ~ *(p + 1);
X[n] ~ *(p + n);
```

- Declarations: the following function declarations are equivalent:

```
1.extern double func(double X[]);
2.extern double func(double *X);
```

  - Format #1 is often preferred as it does conveys more information