Lecture 05 - Pointers (last updated: 2023-09-25)

Pointers hold addresses

Locations in memory are indexed by memory addresses. We call variables that hold memory addresses "pointers," because they "point" to a location in memory. The type of a pointer that points to some type T is written "T *". For example:

```
int *p;             // p is a pointer to an integer
```

We can assign p the location of integers in memory. For example:

```
int x = 1;

p = &x;             // p now holds the address of x
```

We can obtain the address of x (i.e., a pointer to x) using the prefix "address-of" operator, &. You may hear this colloquially referred to as the "reference" operator, since the pointer "refers" to x. However, we will insist on calling it the "address-of" operator because the term "reference" refers to something else in C++.

You can read from the address held by a pointer using the "dereference" operator, *:

```
int y = *p;         // read 1 from x (pointed to by p) and assign it to y
```

We can also write to the address held by a pointer by dereferencing it on the left-hand side of an assignment:

```
*p = 0;             // x is now 0
```

We can of course reassign p to point at something else:

```
p = &y;             // p now points to y

*p = 2;             // y is now 2; x is still 0

++*p;               // y is now 3

(*p)++;             // y is now 4
```

Note the necessary parentheses in (*p)++, which means "dereference p, and increment what p points to". In contrast, *p++ would be grouped as *(p++), which means "dereference the value of expression (p++), which is p before it gets incremented". We will talk more about incrementing pointer variables later.

Pointers are typed

Note that you can’t mix pointers to different types without casting:

```
int    i = 1234;
double d = 3.14;

int    *pi = &i;
double *pd = &d;
```

```
    // pi = pd;         // compiler error

    pi = (int *) pd;    // compiles, but you better know what you're doing
```

Casting can be very unsafe, because you are telling the compiler that you want to reinterpret the bits of one type as another. At this point, if you tried to print the value of *pi, you would find a very surprising value.

There is an exception to C's pointer type rules: void pointers (i.e., void *). Void pointers can point to any type, but do not tell us anything about what they point to (i.e., the size of the data they point to). We can freely convert to and from void pointers without casting:

```
    void *pv = pd;      // no cast necessary; no compiler error
```

However, C does not allow us to dereference void pointers, because it does not know how many bytes we mean to read or write:

```
    double d2;

    // d2 = *pv;         // compiler error: can't dereference a void pointer
```

Knowing that pv currently points to a double value, we can assign pv to a pointer to a double and then deference it:

```
    double *pd2 = pv;

    d2 = *pd2;
```

Or equivalently:

```
    d2 = *(double *)pv;
```

Simulating pass-by-reference with pointers

C is "pass-by-value" language, meaning that the values of function parameters are copied from the caller to the callee. For example, consider this broken implementation of the swap() function:

```
    void bad_swap(int x, int y) {
        int temp = x;
        x = y;
        y = temp;
    }
```

Here is a snippet of code that calls our broken bad_swap():

```
    int x = 1, y = 2;
    bad_swap(x, y);     // Didn't work: x is still 1, y is still 2
```

bad_swap() fails to swap the values of x and y in the caller's context because bad_swap() only swaps its local variables.

In order to make swap() work, we need "pass-by-reference" semantics, which we can fake in C using a level of indirection, i.e., pointers:

```
    void swap(int *px, int *py) {
```

```
        int temp = *px;
        *px = *py;
        *py = temp;
    }
```

Now we can call swap() with pointers to x and y:

```
    int x = 1, y = 2;
    swap(&x, &y);       // Now x is 2 and y is 1
```

Note that we are still passing arguments to swap() by value, except this time the values are pointers.

NULL pointers

Pointer values are just numbers that happen to be memory addresses. The actual values are usually irrelevant, with the exception of the zero value. A pointer containing the zero address is called a NULL pointer.

Normally, the compiler will issue a warning if you try to assign some random integer to a pointer variable, but 0 is an exception:

```
    int *pi = 0;              // pi is a NULL pointer; compiles without warning
```

But it is considered a better style to use NULL instead:

```
    double *pd = NULL;
```

The C standard library provides the following definition:

```
    #define NULL  ((void *) 0)
```

Dereferencing a NULL pointer is considered undefined behavior, and usually leads to a type of runtime error known as a "segmentation fault" (colloquially, a "segfault"). For example, running this program:

```
    // segfault.c
    int main(void) {
        int *i = NULL;
        return *i;            // dereferencing a NULL pointer
    }
```

will lead to this error message:

```
    $ ./segfault
    Segmentation fault (core dumped)
```

Just like integer variables, pointer variables can be used in a boolean expression. Just like integers again, 0 evaluates to false and anything else evaluates to true. In other words, only NULL pointers evaluate to false and all other pointers evaluate to true.

Do not confuse, however, a NULL pointer with a pointer to a variable holding 0:

```
    char c = 0;
    char *p = &c;             // pointer to zero
    char *n = 0;              // NULL pointer
```

```
    if (p) {
        // reached
    }

    if (n) {
        // not reached
    }

    if (*p) {
        // not reached
    }

    if (*n) { // segfault!
        // not reached
    }
```

NULL pointers are usually used to signify the lack of something to point to, i.e., "this pointer doesn't point to anything." For example, a function that usually returns a pointer to some useful data may return a NULL pointer to indicate an error.

Dangling pointers

Recall that variables in C belong to one of the two storage classes:

- Static variables live for the entire duration of the running program.
- Automatic variables (aka local or stack variables) are newly allocated every time we call a function, and their lifetime is limited to the duration of that function call.

Consider a function that returns a pointer to a stack variable:

```
    int *alloc_int(void) {
        int i = 0;
        return &i;
    }

    int *p = alloc_int();     // dangling pointer!
```

After alloc_int() returns, p points to where alloc_int()'s stack frame used to live on the stack, but that stack frame is no longer in use! We call p a "dangling pointer," because it points to an address that should no longer be pointed to. If the program calls another function, p may end up pointing to somewhere unpredictable in that other function's stack frame.