CS212 GBA

Introduction to Pointers

Spring 2007

A ***pointer*** is a datatype used to store a memory address. You will learn soon that pointers are particularly useful for creating and managing data structures in languages such as C, and that they are especially necessary when programming for the Game Boy Advance so that we can interact nearly directly with the hardware.

1 Basics

Most applications of pointers storing memory addresses of variables and dynamically allocated portions of memory, so we say that an instance of a pointer *points to* such a variable or section of memory.

1.1 Pointer Types

We specify that a variable is of a pointer type by prefixing the variable's identifier with the character `*`in its declaration[1]. A pointer may store the address of a variable of any type, so we also specify the type of the variable that it points to. For example, a pointer, `p`, to an `int` can be declared as `int *p`. Similarly, we can declare a `char` pointer to, `q` as `char *q`.

Of course, there's no reason to stop with just simple primative types; we can declare a variable called `r`, a pointer to a `int` pointer, as `int **r`.

We can also store pointers to structures. Suppose we also have the following (all too familiar) type definition for a struct representing a point.

```
typedef struct {
        int x;
        int y;
} point;
```

We can declare a pointer called `t` to a point as `point *t`.

1.2 `void*`

You will often see pointers to `void`. This is peculiar because `void` isn't a tangible type: one cannot create a variable of type `void`. This is particularly useful because it allows the programmer to return a pointer to variable of an unknown type, allowing the `void *` to be casted to whatever datatype is desired.

[1] In many books and web resources, you may see the `*` accompanying the identifier of the type (*eg*,`int* p`). While in simple cases this is equivalent, we strongly recommend against adopting this style because it may bring about unexpected behavior in more complicated cases. You have been warned.

1.3 Operators

1.3.1 Address-of

The ***address-of*** operator (`&`) is a unary operator that acts on a *variable* and returns its memory address[2].

For example, suppose we have the following declaration and assignment: `int a = 42;`. The expression `&a` will be a pointer to `int`, and its value will be the memory address where a is stored. Note that in most situations, you won't be particularly interested in this value except for storing it in a pointer; for example, `int *b = &a`.

What about the expression `& 42`? Remember that the `&` operator only acts on variables, so your C compiler will flag such a misuse at compile time.

1.3.2 Dereferencing

The ***dereferencing*** operator (`*`) is a unary operator that acts on *an expression resolving to a pointer* and returns the data stored at that address. It might help you to think of it, loosely, as the inverse of the address-of operator.

For instance, the expression `*b` (with `b` as defined above) will yield the value of 42: `b` stores the memory address of `a`, which stores 42. Suppose we had a declaration `int **c = &b`. How could we extract the value 42 from `c`?

Let's look at another example using dereferencing.

```
int foo;         // foo is an int
int *bar;        // bar is a pointer to an int

bar = &foo;      // bar points to foo
foo = 42;        // foo becomes 42, so *bar is also be 42
*bar = 100;      // The data pointed to by bar becomes 100.
```

What is the value of `foo` at the end of the segment?

The C Programming Language uses pointers implicitly in a number of places. For example, the implementation of arrays relies heavily on pointers. When you create an array of integers with `int foos[10]` you have allocated space in memory for 10 integers, one after the other. You could accomplish the same thing by declaring `int *foos = (int *)malloc(10 * sizeof(int));` (we will explain this line of code in depth in a moment, it is important). You could then either access the array through the normal means of `foos[0]` or through its pointer, by adding the number of the element you want to access to the pointer, and then defreferencing that. Like this:

```
*(foos+0) = 5;
*(foos+1) = 6;
*(foos+2) = 7;

printf("%d %d %d", foos[0], foos[1], foos[2]); // will print "5 6 7"
```

As you can see with proper use of memory you can do some very useful things, which is why it is good to be aware of how you are using memory, and how to use it most efficiently.

[2]Do not be confused that this operator is represented the same way as the binary bitwise-and operator with which you are already familiar; you will need to be mindful of the context in which it appears to distinguish the difference.

2 Memory Management

There are three tools that you will use with the Gameboy Advance to allocate and free memory for yourself. These are the `malloc()`, `sizeof()`, and `free()` operators.

2.1 sizeof()

The `sizeof()` operator has the prototype `size_t sizeof ( type )`, size_t being a type that defines sizes of strings and memory blocks. One generally does not use `sizeof()` outside of a call to `malloc()`.

2.2 malloc()

The `malloc()` operator has the prototype `void *malloc(size_t size);`. It takes in a value of type `size_t` which is what is returned by the `sizeof()` operator. You can also see that it returns the `void *` type we mentioned before, this allows you to cast the result to whatever pointer type you just allocated space for. Here is an example:

```
int *f = (int *)malloc(sizeof(int));
```

This allocates space for the int in memory, and returns a pointer to that memory, which is then cast to an int pointer, as that's the data type we are dealing with. `malloc()` is especially useful when using pointers to structs, which is something you will do excessively in writing your final games. If you had a structure that you wanted to make a pointer to, you could declare it like this:

```
typedef struct {
  u16 x;
  u16 y;
  OAMEntry* sprite;
  u8 visible;
} Bullet; // this creates type Bullet, which has the above fields

typedef Bullet* Bullet_p; // this creates the type Bullet_p, which is
                          // a pointer to a bullet.
```

The declaration of the `Bullet_p` type is not necessary, but it makes things clearer. To get a pointer to the alloced space in memory for an instance of the Bullet type, we use `malloc()` like so:

```
Bullet_p b = (Bullet_p)malloc(sizeof(Bullet));
```

b is now a pointer to a block of memory big enough to hold an instance of a variable of type Bullet. Note that we could also declare *b* this like:

```
Bullet *b = (Bullet *)malloc(sizeof(Bullet));
```

We have only declared the `Bullet_p` type to increase the readability of the code. You may find it useful to do so as well, or you may find it pointless. Either way you will need to do this manner of declaration of alot, otherwise you will find your GBA game behaving extremely screwy.

2.3 free()

The last operator you will need to use to manage memory appropriately is `free()`, which has the function prototype `void free(void *ptr);` This is fairly interesting because it takes a `void *` as its parameter, which means that the free operator knows nothing about the type of variable it is operating on. This is sensible when we realize that free is just freeing the memory that a given variable occupies, which does not depend on the variable type (other than that variable types occupy different amounts of space in memory). The usage of `free()` is very straightforward, you call it whenever you are done using a pointer and want the space it occupies to go back into usable memory.

```
int *f = (int *)malloc(sizeof(int)); // allocates an int

free(f); // frees the memory allocated to f, an int
```