IAP 2012

6.270 Advanced C Course Notes

Isaac Gutekunst

Some Terminology
Compiler - A computer program that reads c code sequentially, and converts into code that can be run by a computer, and in our case, the HappyBoard.

Basic Variables - How are they actually stored?

```
uint8_t a = 4;
```

What is really going on?
Memory can be thought of as a giant table, a contiguous region of space that can be used to store values. It is organized into "boxes" that can each store one byte (8 bits) of information. Each byte in memory has an "address", or a numerical value that corresponds to its position in memory.

In the example:
```
uint8_t a = 4;
```

The compiler associates the name a with some space in memory to store a value. If you look closely, at the definition, you will notice three things. One: the u means that a stores unsigned values. Two the variable uses 8 bits of storage. That means that you can store values from 0 to 255. The t is just a naming convention. In fact, the entire variable type is just a naming convention, where the different parts to have meaning. You can in fact us a int variable type, which corresponds to a signed 16 bit int, which is also named int16_t. There are more variables types available, and you can read about those in the basic C lecture slides. Figure 2.1 shows how the variable a will be stored in memory.

Mapping	Value	Address
		...
		4
		3
		2
		1
a ---->	4	0

Table 2.1: Memory layout for simple variable
What if in addition we create a new variable of type uint16_t? In this case the compiler will allocate 16 bits of memory (2 bytes) to store the variable.

If our code looks like

```
uint8_t a = 4;
uint16_t b = 17;
```

Our memory map will look like table 2.2. Notice that be occupies to bytes in the memory table. The "higher order" byte is zero for our example value of 17. A value of 256 for example will have the MSB (most significant bit) equal to 1, and the LSB (lest significant bit) equal to 1 as well. If you are confused by this, consult Wikipidia (http://en.wikipedia.org/wiki/Binary_numeral_syste

Mapping	Value	Address
		...
		7
		6
		5
		4
		3
also b->	0	2
b----->	17	1
a ---->	4	0

Table 2.2: Two variables

Arrays

Arrays let you store a collection of identical variables and let you access them with a subscript (square brackets in c). I will start off with an example.

```
#include <joyos.h>
int usetup(){

}

void printMines(uint8_t mines[]){

  uint8_t i;
  for (i = 0; i < 6; i++) {
    printf("Mine #%d has %d resources \n",i,resources[i]);
  }
}

int  umain(void){
  uint8_t resources[6] = {10,10,5,4,2,7}; //A way to "initialize" an array
  printMines(resources);

}
```

OUTPUT

```
Mine #0 has 10 resources
Mine #1 has 10 resources
Mine #2 has 5 resources
Mine #3 has 4 resources
Mine #4 has 2 resources
Mine #5 has 7 resources
```

Code Sample 2.1

Arrays are stored in memory in sequentiall memory locations. That is, the first item in the array will go in, for example, the 0_{th} box (assuming it is an 8 bit variable). The i_{th} item will be stored in the i_{th} location in memory (box number i). Table 2.1 shows this.

Note: C does not keep track of the size of arrays, unlike Java and python for example.

Mapping	Value	Address
		...
		7
		6
mines[5]	7	5
mines[4]	2	4
mines[3]	4	3
mines[2]	5	2
mines[1]	10	1
mines[0]	10	0

Table 2.3: Array of mines

If instead of uint8_t, we declared mines as uint16_t, the array would, not surprisingly take twice as much space. In this case, each element in the array would be mapped to two bytes in memory, each offset by the size of the individual element, multiplied by its index.

Arrays can be passed as arguments to functions as well. Refer back code sample 2.1 to see the syntax. There is one gotcha, however: in c, arrays are passed by reference. If you don't know what the means, keep reading and learn about pointers.

Pointers

Pointers are simply variables that instead of storing a number or character, they "point" to another variable. Their actual value is the address of another variable in memory, or if you are not careful, some arbitrary location in memory that you probably shouldn't access. Why would one ever want to use pointers in the first place? How about an example? Let's say you have two variables, a and b, which each store the mine number you want to go mine next. Now let's say that you discover that one of the mines is depleted, so you want to swap a and b, so you will visit a different mine first. If you think about this for a while, you might come up with a solution like this:

```
#include <joyos.h>

int usetup(){
}

int umain(){
	uint8_t mine_a = 4;
	uint8_t mine_b = 7;

	printf("Mine a: %d\n", mine_a);
	printf("Mine b: %d\n", mine_b);

	//swapping part
	uint8_t temp = mine_b;
	mine_b = mine_a;
	mine_a = temp;



	printf("Swapped:\n");
	printf("Mine a: %d\n", mine_a);
	printf("Mine b: %d\n", mine_b);
}
```

OUTPUT:

```
Mine a: 4
Mine b: 7
Swapped:
Mine a: 7
Mine b: 4
```

Code Sample 2.2: Basic Swapping

This works fine, and swaps the values of a and b. What if you want to perform swaps in several parts of your code. Good coding practice says you shouldn't repeat yourself, and that you should group your code into functions. Let's go ahead and do that. Let's throw the swapping code into a function and call it from umain.

```
#include <joyos.h>

int usetup(){
}

void swap(int a, int b){
	//swapping part
	uint8_t temp = mine_b;
	mine_b = mine_a;
	mine_a = temp;
}

int umain(){
	uint8_t mine_a = 4;
	uint8_t mine_b = 7;

	printf("Mine a: %d\n", mine_a);
	printf("Mine b: %d\n", mine_b);
	swap(a,b);



	printf("Swapped:\n");
	printf("Mine a: %d\n", mine_a);
	printf("Mine b: %d\n", mine_b);
}
```

```
OUTPUT:
Mine a: 4
Mine b: 7
Swapped:
Mine a: 4
Mine b: 7
```

Code Sample 2.3: Incorrect swap, and example of local variables.

Woah! It didn't work? Why not? Well, this would be a good time to introduce the idea of pass-by-value. When a function is called, the arguments (in this case a, and b) are actually copied into local copies assessable only to the function (in this case swap). Changing them has no affect to the variables of caller, or area that called the function. This is because they scope of the variables is different. In umain, a maps to a particular address. Once the program is running swap, a is now pointing to a different location in memory. A good way to remember this is that every nested block of curly bracket has the ability to make new mappings to variables. For example:

```
#include <joyos.h>

int usetup(){
}
int umain(){
	int a = 4;

	if(1){ //always true
		int a = 7;
		printf("A: %d\n", a);
	}
	printf("A: %d\n",a);
}
```

```
OUTPUT:
A: 7
A: 4
```

Code Sample 2.4

What is going on here? The body of the if statement is a new "block". That means that new variables created in this block mask variables created in a parent block. Once the program leaves the body the if statement, any variables created inside it are destroyed.

What does this have to do with pointers? Nothing really, but it emphasize the need for them for the contrived example of the swap function.

Here's a simple example of using pointers:

```
#include <joyos.h>

int usetup(){
}
int umain(){
      int a = 4;
      int * a_ptr;      // pointer names are suffixed with _ptr by convention
      a_ptr = &a;       //Point a_ptr at a. (Set a_ptr to the "address of" a)

      printf("a: %d\n",a);    //print a

      *a_ptr = 7;             //set "the value of what" a_ptr pointer points
                              //at to 7
      printf("a: %d\n",a);    //print out a again
}
```

```
OUTPUT:
A: 4
A: 7
```

Code Sample 2.5

What is going on here?

The first unfamiliar line is the declaration of the pointer a_ptr:

```
int * a_ptr;      // pointer names are suffixed with _ptr by convention
```

This line creates an int pointer. Here the asterisk is the only difference from creating an int variable.

The next line is also unfamiliar:

```
a_ptr = &a;       //Point a_ptr at a. (Set a_ptr to the "address of" a)
```

This line can be read as "set the value of a_ptr to the "address of" the variable a. The ampersand is easier to understand if you read it out-loud as "the address of." The ampersand operator is sometimes called the referencing operator, as it gets a reference, or pointer to a a variable.

The next new line is:

```
*a_ptr = 7;             //set "the value of what" a_ptr pointer points
```

This line is interesting. It uses the * syntax again, but in this case, not in quite the same way. The * here is used to access the value of the variable it proceeds. It can be read as "the value of", or "the value of the variable" ptr "points to".

Note: The defining a pointer variable can be a bit counter-intuitive. A good mnemonic is that that a_ptr is of type int*, or by mentally grouping the asterisk with a_ptr, you can think that *a_ptr is an int.

Mapping	Value	Address
		...
		7
		6
		5
		4
		3
		2
a_ptr	0	1
a	10	0

Table 2.4: Memory map for Simple Pointers example (Code sample 2.5)

Examining the table more closely reveals how a seemingly mysterious pointer actually works. It doesn't have some kind of arrow that "points" at the other variable. It just stores it's address. Additionally, the compiler knows that since, in our example, the size of the variable it points to is 8 bits, or one byte. This will become useful later.

Pointers can be used as arguments to functions. Let's go back and try to make the infamous swap function again, this time with pointers.

```
#include <joyos.h>

int usetup(){
}

void swap(int *a, int *b){
      //swapping part
      uint8_t temp = *mine_b; //temp = the "the value pointed to by" mine_b
      *mine_b = *mine_a;      //"the value of" mine_b = "the value of" mine_a
      *mine_a = temp;         //"the value of" mine_a = temp
}

int umain(){
      uint8_t mine_a = 4;
      uint8_t mine_b = 7;

      printf("Mine a: %d\n", mine_a);
      printf("Mine b: %d\n", mine_b);

      swap(&a,&b);



      printf("Swapped:\n");
      printf("Mine a: %d\n", mine_a);
      printf("Mine b: %d\n", mine_b);
}
```

OUTPUT:

```
Mine a: 4
Mine b: 7
Swapped:
Mine a: 7
Mine b: 4
```

Code Sample 2.6: Successful Swap. Using pointers as arguments.

Most of this code should be familiar to you. One line should jump out as a bit suspicious:

```
swap(&a,&b);
```

It's simple to understand once you use the mnemonics. You are calling the function swap, and passing it the "address of" a, and the "address of" b. This makes sense, since the swap function takes in pointers to uint8_t, which inherently are addresses.

Pointer Arithmetic

If you have been paying a lot of attention to what is going on, you might realize that since pointers really store the addresses of other variables, it is conceivable to modify the value of a pointer, "by hand", to make it point at a new address. For example, you could make it point to address 0xFFEEBB, which is a critical joyos variable. You can then modify it, cause a horrible crash, and lose the competition. Or you can do some clever things.

Let's say you create an array of uint8_ts, and create a pointer to the first item, like so:

```
uint8_t mines[5] = {1,2,3,4,5};

uint8_t * first_mine = &mines[0]; // pointer to the first mine
```

Now let's increment the value of first_mine.

```
first_mine +=1; //increment first_mine
```

Keep in mind that we are adjusting fist_mine's value, not the value of the variable it points to (notice the lack of asterisks). This has the effect of changing what first_mine points to.

Mapping	Value	Address
		...
		7
		6
first_mine	0	5
mines[4]	5	4
mines[3]	4	3
mines[2]	3	2
mines[1]	2	1
mines[0]	1	0

Mapping	Value	Address
		...
		7
		6
first_mine	1	5
mines[4]	5	4
mines[3]	4	3
mines[2]	3	2
mines[1]	2	1
mines[0]	1	0

Table 2.5: Pre-incremented layout **Table 2.6: Post incremented layout**

As you can see above, incrementing the pointer makes it point to the next value in the array. Note: If the datatype of the pointer is two bytes, incrementing the pointer by one make it point two bytes ahead in memory, to the next value. For example, if our pointer was a uint16_t, incrementing it by one would make it point two bytes ahead in memory.

```
#include <joyos.h>
void printMines(uint8_t mines[]){


  uint8_t i;
  for (i = 0; i < 5; i++) {
    printf("Mine #%d has %d resources \n",i,resources[i]);
  }
}


int usetup(){


}
int umain(){
      uint8_t mines[5] = {1,2,3,4,5};
      printMines(mines);
      uint8_t * first_mine = &mines[0]; // pointer to the first mine
      first_mine +=1; //increment first_mine
      *first_mine = 17;                    //change value of the 2nd mine to 17
      printf("-------\n");
      printMines(mine);

}
```

OUTPUT

```
Mine #0 has 1 resources
Mine #2 has 2 resources
Mine #2 has 3 resources
Mine #3 has 4 resources
Mine #4 has 5 resources
-------
Mine #0 has 1 resources
Mine #2 has 17 resources
Mine #2 has 3 resources
Mine #3 has 4 resources
Mine #4 has 5 resources
```

Code Sample 2.8: Pointer Arithmetic

Function Pointers

Pointers can also point at functions. This can be useful if you want to create a function that will call another function at some point. The best way to explain this is with an example, but first, the syntax:

```
return_type (* name) (args)
```

So to make a function that has one argument that is a function returning a uint8_t, and taking in a uint8_t as an argument, you would do so as follows:

```
void foo( uint8_t (* bar) (uint8_t) ){
      bar(5);     //calls the function provided to foo with the argument 4


}
```

A joyos example, from Jesse Moeller's 2010 lecture is a perfect example of a good use of function pointers.

```
void DoUntil(void (*Action)(),
               uint8_t (*StopCondition)(),
               uint16_t end_time)
{
      while (time < end_time && !StopCondition()){
      Action()
      }
}
```

This function could be called in a variate of ways to perform useful robotic activities. For example:

```
DoUntil(&Forward, &HitWall, 500);
DoUntil(&Backward, &HitWall, 500);
DoUntil(&Forward, &NearRobot, 500)
```

These three calls pass in the "address of" Forward, HitWall, Backwards, and NearRobot, which are all regular functions, allowing DoUntil to call them.

Structs

Structs, short for structures are an advanced data type available in c. They enable group several variables together into a single structure for ease of code clarity, as well as getting around the problem of a single return value.

To create a new struct data type, you use the following syntax:

```
struct struct_name{
      var_type var1;
      var_Type var2;
      ///etc
};          //notice the semicolon. It's important
```

For example:

```
struct point{
      double x;
      double y;
};
```

To create a variable of your newly created type, treat `struct struct_name` as any other variable type. For example:

```
struct point origin;
```

To access members of a struct, use the dot operator. For example:

```
origin.x = 0;
origin.y = 0;
```

Internally, structs are just a contiguous[1] area in memory with symbolic names that point to the various members of the struct. For our example above, the memory map would look as follows:

Mapping	Value	Address
		...
		7
		6
		5
origin.y		4
		3
		2
	0	1
origin.x	0	0

Table 2.7: Struct memory layout

Notice that origin.y starts at location 0, but occupies 4 bytes, leaving origin.y to start at location 4. That is because doubles take up 4 bytes of memory.

Structs can be used as arguments to functions. A common usage would be to make a distanceBetween function that take two point structs, and returns the Cartesian distance.

```
#include <joyos.h>
#include <math.h>
struct point{
```

[1] Ask an organizer about this. It's not exactly true.

```
    double x;
    double y;
};

typedef struct point Point;

double distanceBetweenPoints(Point a, Point b){
  double d2 = (a.x - b.x) * (a.x - b.x) +
           (a.y - b.y) * (a.y - b.y);

  return sqrt(d2);
}

int usetup(){

}
int  umain(void){
  Point origin;
  origin.x = 0;
  origin.y = 0;

  Point mine;

  mine.x = 300;
  mine.y = 400;

  double distance = distanceBetweenPoints(origin,mine);
  printf("Distance is %.2f",distance);


}
```

OUTPUT

```
Distance is 500.02      //Estimated answer. Will vary slightly depending on
                        //size of the x,y coordinates in the struct
```

Code Sample 2.9: Using structs in functions.

If you examine the code above, one thing might jump out at you. What is this typdef statement? Typedefs, or Type Definition let's the programmer create a symbolic representation of a new data type. Without it, the distanceBetweenPoints function would need the begin like this:

```
double distanceBetweenPoints(struct point a, struct point b){
//etc
```

By using a typedef, we are assigning Point to the value of "struct point", and from there on, using typing Point is the same as typing struct point.

Note: The capitalization is a stylistic choice and does not matter technically, but consistency is important.

Similarities Between Arrays and Pointers

Arrays are really just pointers to blocks of memory large enough to store all the values. Read that previous sentence again. Why would this be so? The main reason C was designed this way was to improve performance. If we have an array of 100 items and need to move that data around all over the place, for example, when calling a function, our code can get pretty slow.

This similarity can be explained with a few examples.

```
uint8_t data[2] = {10,10};
```

From before, the understand is that each name array_name[index] points to the corresponding element. This is true, but a bit misleading. What is happening internally is that the variable, in this example, data, is really a pointer to the first item in the array. Accessing `data[1]`, is equivalent to looking at the the pointer to the first item,jumping forward by one item, and then accessing the value. In fact, data[1] is internally the same as a *(data+1).

Note: A compiler might complain if you do this, as it is a dangerous path you are walking on. It's important to be aware of what you are doing, and it is the compiler's job to warning you when you are doing risky things.

Mapping	Value	Address
		...
		7
		6
		5
	10	4
	10	3
var_foo		2
lols		1
data	3	0

Table 2.9: Real memory layout

Note: If this is just really hurting your brain, don't worry about it too much. It's not vital to successful code. Notice that data contains the value 3, that is it points to memory location three, the start of the array. `data[1]` accesses the data at an offset of 1 from the start of the array, since the pointer/array data type is one byte wide, and we are accessing index 1. If it were an array of `unint16_t,` accessing `data[1]` will access data at an offset of data_width * 1, or 2 * 1, since `uint16_t` variables take up two bytes of storage. Keep in mind that the first location of an array may be located anywhere in memory, either right after the pointer (data), or some arbitray location that the compiler found convenient.

```
#include <joyos.h>
int usetup(){

}

void printMines(uint8_t mines[]){

  uint8_t i;
  for (i = 0; i < 2; i++) {
    printf("Mine #%d has %d resources \n",i,resources[i]);
  }
}

int  umain(void){
      uint8_t resources[6] = {10,10,5,4,2,7}; //A way to "initialize" an array
      printMines(&resources[0]);
      printf("----\n);
      printMines(&resources[1]);
      uint8_t * third_mine = &resources[2];
      printf("----\n);
      printMines(third_mine);

}
```

OUTPUT

```
Mine #0 has 10 resources
Mine #1 has 10 resources
----
Mine #0 has 10 resources
Mine #1 has 5 resources
----
Mine #0 has 5 resources
Mine #1 has 4 resources
```

Code Sample 2.10: Arrays are really pointers?

Take some time to examine this code. If the ampersands int asterisks are throwing you for a loop, go back a few pages and review pointers. What this example demonstrates is that arrays, in fact are just pointers, and you can do cleaver and dangerous things with them.

Pointers to Structs

Advanced C is not advanced without some convoluted uses of data types. Pointers can point to structs as well. The syntax follows almost identically from other pointers, with a few gotchas. Let's look at an example:

```
#include <math.h>
#include <joyos.h>
struct point{
    double x;
    double y;
};

typedef struct point Point;

double distanceBetweenPoints(Point *a, Point *b){
  double d2 = (*(a).x - *(b).x) * (*(a).x - *(b).x) + //<--Gross!
           (b->y - a->y) * (b->y - a->y);             //<-- nicer
  return sqrt(d2);
}

int usetup(){
}
int  umain(void){
  Point origin;
  origin.x = 0;
  origin.y = 0;

  Point mine;
  mine.x = 300;
  mine.y = 400;

  double distance = distanceBetweenPoints(&origin,&mine);
  printf("Distance is %.2f",distance);



}
```
Code Sample 2.10: Pointers to structs

Almost everything in this code should look familiar, except for one statement that spans two lines:

```
double d2 = (*a).x - (*b).x) * (*a).x - (*b).x) + //<--Gross!
           (b->y - a->y) * (b->y - a->y);             //<-- nicer
```

The first part can be explained using the usual pointer mnemonic. (*a).x can be read as:
Take the value pointed to by a (a struct point), and then access element x in it. The parenthesis are necessary because in C, the order of operations, or precedence rules dictate that the . operator will be executed first. If you ommited them, you would be trying to access the x field on a pointer, which is not possible, since pointers only store addresses to memory, not actual numerical values, in this case. Fortunately for us, the designers of C decided to make a slightly nicer way to access members of structs pointed to by pointeres. The syntax is as follows:

If a is a pointer t a struct point, then `a->x` is equivalent to `(*a). x` .

Threading

In a sentence, threading lets you run multiple functions at the same time. It is useful if, for example, you want to have one function responsible for driving straight and doing the PID loop, and another for deciding where to navigate to, and you want these function to run at the same time.

The Hapyboard can really only run one bit of code at the same time, so threading is a way to have joyos chop your functions into little bits, and run a few bits of one, then switch over, and run a few bits of another function, and so on. This is amazingly useful, and sometimes a bit dangerous.

Let's put danger aside, and learn the basics of threading. Threading, as mentioned earlier, lets you run two functions at the same time. Each concurrent function is called a thread. In joyos, you create a thread by calling the create_thread function

```
#include <math.h>
#include <joyos.h>
int navigation_loop(){
      for(;;){
            printf("Amazing Navigation goodness\n");
            delay(500);       //wait 500 ms
            yield();          //tell joyos this thread is done for now
      }
}
int usetup(){
}
int  umain(void){
      create_thread(&navigation_loop, STACK_DEFAULT, 0, "pid_thread");
      for(;;){
            printf("Umain!!!\n");
            delay(500);
            yield();
      }

}
```

OUTPUT: (Results may vary, as joyos may chop up your functions differently)

```
Umain!!!
Umain!!
Amazing Navigation goodness
Amazing Navigation goodness
Umain!!!
etc.....
```

Code Sample 2.11: Basic Threading:

That's pretty much all there is to it. At the moment, the only part you would change as a contestant is the pointer to the function you want to run, and its name. If you are confused about the ampersand, flip back a few pages, and review pointers and function pointers. What you are doing is telling the `create_thread` function you want it to make a thread with your function, in this case `navigation_loop`, telling it to use the normal amount of memory for the stack[2], giving it a name, and setting its priority to zero.

Sharing data between threads

Once you have written a few threads, you probably want to allow them to communicate. The simplest way to do this is what is called shared memory. It might sound a bit scare, but it's easy as pie. All you need to do is create a variable outside, and "above" the functions you want to share data. This variable is now "global" to the two functions. Now you can change this value in one function, and have the other function act on this changed value. Here is a really simple example:

```
#include <math.h>
#include <joyos.h>
unsigned long counter = 0;
int updater(){
      for(;;){
            counter = counter +1;   //update counter
            delay(1000);                  //wait one second
      }
}
```

[2] Ask an organizer about the stack. It's not important that you understand it, but if you are interested in computer architecture, it's a cool thing to learn about.

```
int usetup(){
}
int  umain(void){
      create_thread(&updater, STACK_DEFAULT, 0, "updater thread");
      for(;;){
            printf("Value: %ld\n",counter); //print the value of the counter
            delay(500);                   //wait 1/2 a second
            yield();
      }

}
```

OUTPUT: (Results may vary, as joyos may chop up your functions differently)

```
Value: 0
Value: 0
Value: 1
Value: 1
Value: 2
Value: 2
Value: 3
Value: 3
....etc
```

Code Sample 2.12: Sharing Data
Locks

Locks are a feature of joyos that makes it easier for multiple threads to not clobber shared data. The main problem that can occur when you share data without locks, is that one thread can write some data to a large variable (an array of resource quantities for example), be halfway though writing, when it is interrupted by joyos, which decided it was time to run another thread. Now the array contains some new values, and some old values, which can be a problem. Another example is when two threads share a point struct that contains an x and y member, as follows:

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

Lets say that one thread (navigation) is responsible to pick targets to drive to, and another thread (driving) is responsible to control the motors to actually get there.Imagine that vigation thread has just decide to update the target. Its code might look somewhat like this:

```
...
target.x = new_x;
target.y  = new_y;
…
```

Right after it updates the value of x, it might be interrupted by joyos, leaving a completely incorrect coordinate in the target variable. This problem is known as a race condition.

Note: Joyos doesn't necesary chop code into bits, line by line. Sometimes a line will be broken up into several smaller pieces, and therefor, the execution of a line might be interrupted. The smallest, indivisible, and uninterpretable operations are called atomic.[3]

So how can we solve this problem?

How about a far fetch metaphor: Five MIT students share an apartment with one bathroom that can NOT be physically locked. To avoid the inevitable embarrassing situation when one student walks in on another, they agree on special protocol. They decide to leave a padlock in a box on the door, and take it when entering the bathroom. They all agree that when the padlock has been acquired, someone is using, and will not enter. When the occupant leaves the bathroom, they release control of the padlock, and place it back in its box.

Joyos has a feature that behaves almost identically. Interestingly enough, this features is known as locks. To use them, you first must create and initialize them. After that, the function `acquire` takes the lock, and `release` releases the lock. When a lock is acquired, no other thread can acquire it. An attempt to call `acquire` when it is already acquired will block, that is, just sit there until the lock is released by the other thread.

Syntax:

```
struct lock lock_name;  //create the lock object
```

[3] If you are interested in this, ask an organizer.

```
init_lock(struct lock * k, const char * name);  //initialize the struct
acquire(struct lock * k);                       //acquire the lock
release(struct lock * k);                       //release the lock
```

For example:

```
struct lock bathroom_lock;                      //create the lock object

init_lock(&bathroom_lock, "Bathroom Padlock");  //initialize the struct
acquire(&bathroom_lock);                        //acquire the lock
release(&bathroom_lock);                        //release the lock
```

This solves the problem, but only if we are careful about using locks. Keep in mind that locks are not magical. They don't prevent reading and writing of variables, but are a convenient way to read and write variables safely from multiple threads. However, locks present some new challenges. Let's switch back the the building of MIT students. Let's say one student picks up the padlock, walks into the bathroom, pulls out a parachute and jumps out the window, and forgets the padlock in the bathroom. The bathroom is no longer occupied, and is safe to enter, but the lock is missing. Other students will not enter the bathroom, thinking it's occupied. Even the parachuter will not enter the bathroom (he is a little thick-skulled, he did just jump out a bathroom window), as he to thinks it is occupied. This is a type of deadlock.

In C, this kind of deadlock will occur if you forget to release a lock before returning from a function, for example.

There is one other thing to watch out for. Let's return once again to the building of crazy students. They also have only one telephone line, but multiple phones. Excited by the success of their bathroom lock system, and terrified of listening in on other's conversations, they decide to implement one for the telephone as well. They place a small plastic phone model in a box, and call it the phone lock. Some students like to talk on the phone will in the bathroom (yeah, weirdos), and for a while this system works. Unfortunately, one day one student goes to have a phone/shower, grabs the phone lock, and starts walking towards the bathroom. Another student has a similar idea, but grabs the bathroom lock, and then starts walking to pick up the phone lock. They meet in the hallway, and come to a standoff, each refusing to give up his/her respective lock.

This example seems a bit contrived, but is a very common mistake when dealing with multiple locks. The quick takeaway is to always acquire and release multiple locks in the same order. Otherwise the threads may become deadlocked.