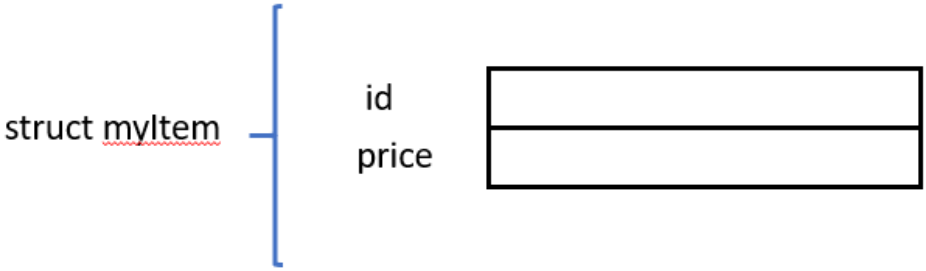
Structures and Pointers

Data Structures

In a previous lab you learned the basics of functions and variables. This lab will introduce you to some more complex concepts that, lets you make more interesting applications that are also both more readable and more efficient. Let's start with some code:

```
#include "msp.h"


struct myItem{
    int id;
    float price;
};

struct myItem test_function();

/**
 * main.c
 */
void main(void)
{

    struct myItem item1;
    struct myItem item2;

    item1.id = 1;
    item1.price = 2.30;

       WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;          // stop watchdog timer
       item2 = test_function(item1);

}

struct myItem test_function(struct myItem testItem)
{
    testItem.id = 2;
    testItem.id = 3.22;

    return testItem;
}
```

The introduction of structures allows you to organize your data, when it makes sense, even if the data is not the same type. In this case you've introduced a new data type, the struct myItem which has an integer id and a float price associated with each variable of type struct myItem. When you declare of variable type struct myItem it will contain both an integer and a float. Here is a diagram that might help:

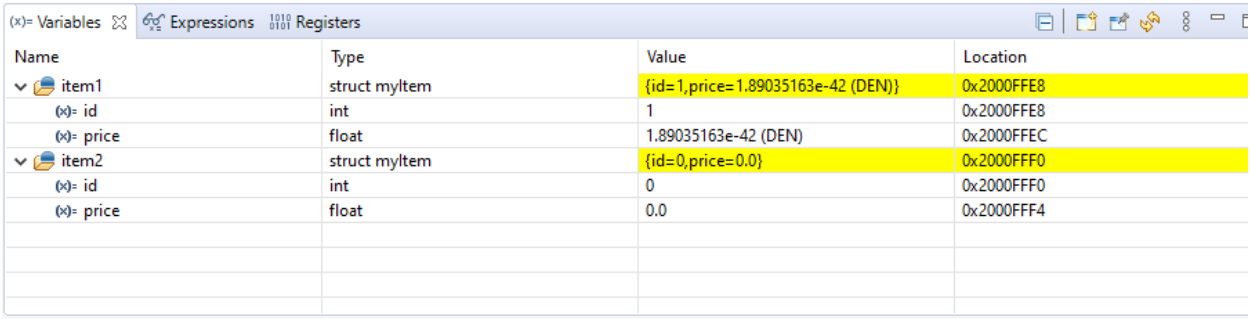


To access each individual element of the structure you'll use the dot connotation. For example:

item1.id = 1

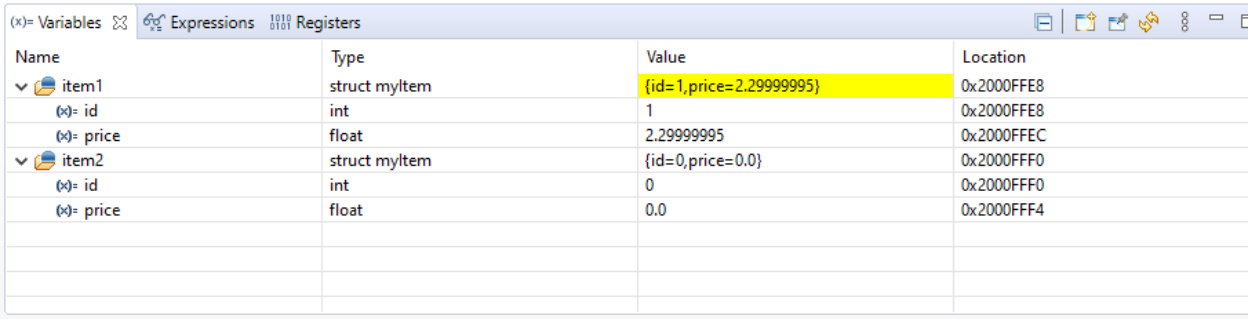
assigns the id value of the variable name item1 to 1.

Now compile the program and then debug it. Step over the two variable creations, let's look at the variable view:



Name	Type	Value	Location
item1	struct myItem	{id=1,price=1.89035163e-42 (DEN)}	0x2000FFE8
id	int	1	0x2000FFE8
price	float	1.89035163e-42 (DEN)	0x2000FFEC
item2	struct myItem	{id=0,price=0.0}	0x2000FFF0
id	int	0	0x2000FFF0
price	float	0.0	0x2000FFF4

Notice how the variables item1 and item2 have two elements. At this point they haven't been set to anything, but they do exist. Now step to the point where item1 has been set:

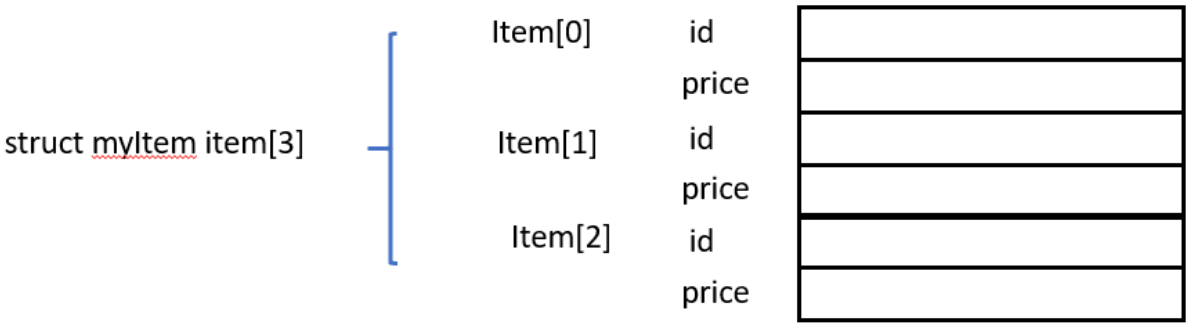


Name	Type	Value	Location
item1	struct myItem	{id=1,price=2.29999995}	0x2000FFE8
id	int	1	0x2000FFE8
price	float	2.29999995	0x2000FFEC
item2	struct myItem	{id=0,price=0.0}	0x2000FFF0
id	int	0	0x2000FFF0
price	float	0.0	0x2000FFF4

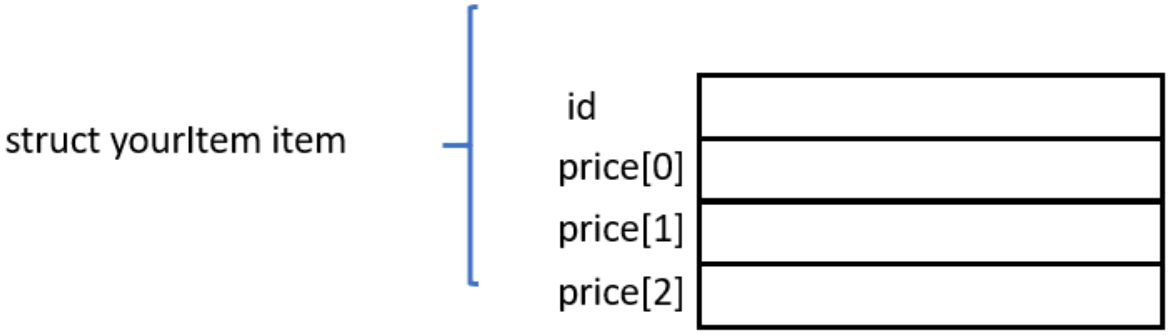
You can now see the values have been set for item1. Now step into the function test_function. As you step through the function you can see them change. When you return from the function you can also see your original item2 change.

One of the most useful ways to use structures is in Arrays. There are, however, two ways to do this, and they can be confusing, so let's look at each.

First you can create an array of the struct myItem. This would look like this in memory:



You can also create a struct that has an array for one of the elements:



Let's show some code with an example of both.

```
#include "msp.h"


struct myItem{
    int id;
    float price;
};

struct yourItem{
    int id;
    float price[3];
};


/**
 * main.c
 */
void main(void)
{

    struct myItem item1[3];
    struct yourItem item2;

    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;     // stop watchdog timer
```

```
    item1[0].id = 1;
    item1[0].price = 3.56;

    item2.id = 2;
    item2.price[0] = 5.30;

   return;

}
```

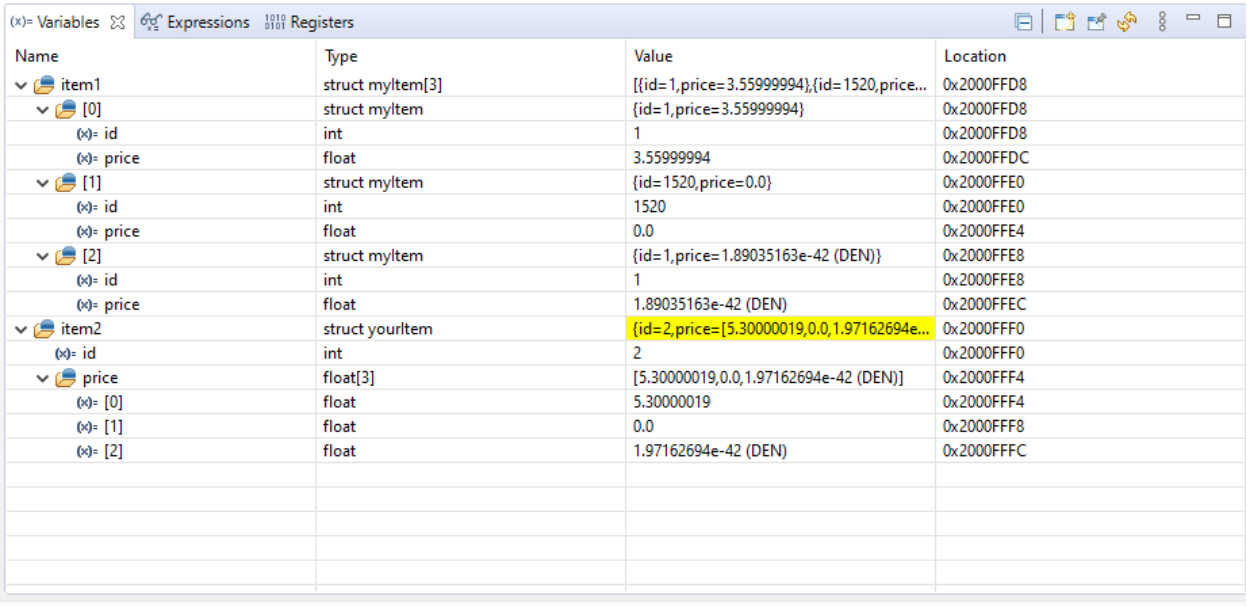
Notice for item1, which is the array of structures, to access a single item you use the syntax:

item1[0].id = 1;

For item2, which contains an array of prices, to access a single item in the array you use the syntax:

item2.price[0] = 5.30;

Now build and debug this code. When you reach the end of the program and look at the Variable viewer you should see this:



You can see the item1 is an array of structs, while item2 is a single item that contains an array of prices.

#define and ENUM

There are two other things you can do to make your C programs easier to read and debug. The first is using the pre-compiler directive *#define*. As an example you can define some text to stand in for some “magical” number you need throughout your code. This means you won’t have to remember what the number stands for, and you won’t have to use up memory in your device if you, mistakenly, chose to try to do the same thing using a variable. It also means that if you choose to change the value of the number, you wouldn’t have to change the number everywhere it’s used in your code, much like using a variable. Here is a common example.

```
#define FLASH_BASE        ((uint32_t)0x00000000)        /*!< Main Flash memory start address */
```

This statement allows you to use the term FLASH_BASE throughout your code and your compiler will replace *FLASH_BASE* everywhere it is found with the number.

The second thing you can do to make your code more readable and debugable is to create enumerations. Enumerations allow you to have multiple indicators grouped together without the overhead in memory and CPU use required by an array. Here is an example that creates an enumeration for the days of the week. Notice the use of the *enum* keyword.

```
#include "msp.h"

enum Days { SUN, MON, TUE, WED, THU, FRI, SAT };

/**
 * main.c
 */
void main(void)
{

    int dayAttendance[] = {1232,
                           500,
                           643,
                           343,
                           872,
                           900,
                           1589};
    int myDayAttendance;
    myDayAttendance = dayAttendance[MON];

    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;     // stop watchdog timer

   return;

}
```

The enum keyword and definition at the top of the code defines a set of expressions that can later be used in place of numbers. In this case SUN can be used for 0, MON can be used for 1, TUE can be used for 2, and so on. Now notice in the code that instead of using the command:

```
    myDayAttendance = dayAttendance[1];
```

you can use the command:

```
    myDayAttendance = dayAttendance[MON];
```

This makes it clearer to yourself and others that you are looking for the attendance associated with Monday.

Pointers

One of the most powerful, but also most confusing aspect of C programming is the use of Pointers. Pointers, most simply defined, are a way to access memory using the address instead of the name of a location. These are useful mainly it three situations. The first is when accessing hardware that looks like memory. The second is in passing large chunks of information. The third is when trying to dynamically (during run time) decide what function to execute. Let's go over the basics of pointers.

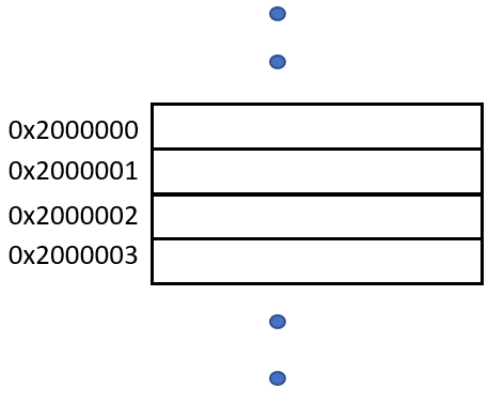
Enter this code into this week's project.

```
#include <msp.h>

void test(int *test2);

/**
 * main.c
 */
void main(void)
{
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;     // stop watchdog timer

    int test1 = 0;
    int *ptrTest = &test1;

    *ptrTest = 3;

    test(ptrTest);
}

void test(int *test2)
{
    *test2 = 4;

    return;
}
```

There are several new operators here that you will need to understand. Let's look at the line:

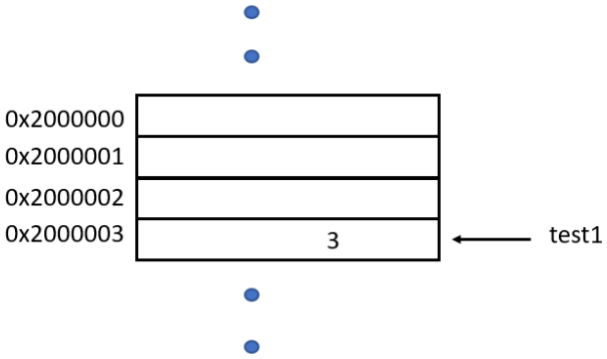
*int *ptrTest = &test1.*

The int *ptrTest is an operator you haven't seen before. When used this way it tells the compiler that the variable *ptrTest* is of type physical memory address of an integer. One of the interesting aspects of using a pointer is that you move into the space where you will be manipulating physical addresses. Let's start with a simple diagram of physical memory:



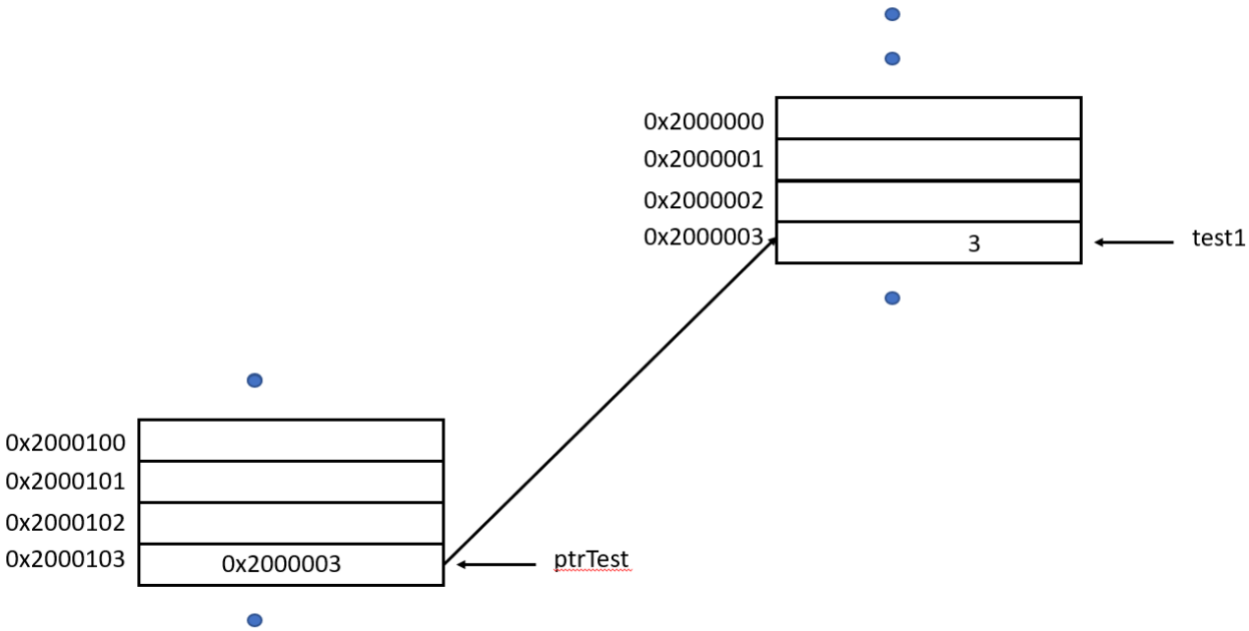
Each of the numbers represent a different physical storage location. The 0x at the start indicates that the number is a hex value, it is easiest to express large digital numbers like this in x.

When you declare a variable in C, the system sets aside some location in memory for your value. The variable name is how the program translates the address to something you can access in your program. So if you set aside a variable that was a single address location, and the name was test1, and you set the value to 3, the memory might look something like this:



During normal C programming you don't really have to know about the actual addresses, rather you use the variable names. However, there are times in C programming when you actually want access to the address. This is when you would use a pointer. When you use the * notation to declare a variable, you are telling the compiler that you want a memory space that will hold an address. To fill that memory location you can ask the compiler to return the physical address of the The ptrTest variable will hold the physical address that will "point to" an integer stored somewhere else in memory. When you want to change the value that the pointer is pointing to, you use the * command before the variable name.

Perhaps another diagram would help:



Again, you might be asking why you need such a structure, so let's review the three different usages of pointers:

1) The first is when accessing hardware that looks like memory such as a temperature sensor whose access acts like a memory address.

   The temperature sensor library provides an interface that points to the temperature sensors's address. Once the pointer is assigned to "point to" the hardware you want to access (normally done in an include file) you can then simply read information from the temperature sensor as if it were a memory address.

2) The second is in managing large chunks of information.

   Occasionally you'll need to manage large chunks of memory. You may be familiar with using the array structure built into C. But you can also use the pointer process to manage this data. This is particularly useful when you have a limited amount of memory and several different programs that may be running on the same processor for long periods of time, so you need to use dynamically allocated memory. This requires that when you need a large chunk of memory you ask the system for the memory and when you are done using the memory that you give the memory back to the system so that other programs can use it.

   In embedded systems you will often use two different areas in the dynamic memory to store information. One of those is the stack. Each process has its own stack, and the stack holds all the

   Here is some example code:

```
#include <msp.h>

void test(int *test2);

/**
 * main.c
 */
int testPort = 0;

void main(void)
{

    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;     // stop watchdog timer

    int *mem_address = 0;

    mem_address = (int *)malloc(256 * sizeof(int));
    test_function(mem_address);
    free(mem_address);
    mem_address = 0;

}

void test(int *test2)
{
    int i = 0;
    for (i = 0; i < 256; i++)
        test2[i] = 4 * i;
```

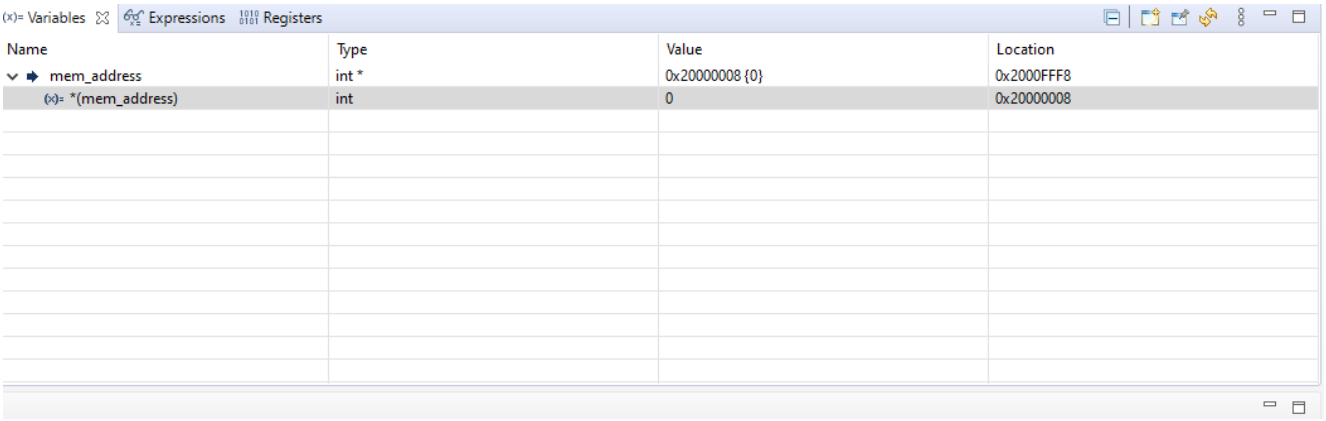
```
    return;
}
```

In this code you create a memory address that points to the start address of the block of the memory you allocated. The allocation is done using the malloc function (Memory ALLOCate). It's job is to inform the system that a certain amount of memory is needed and return a pointer to the beginning of the memory block.

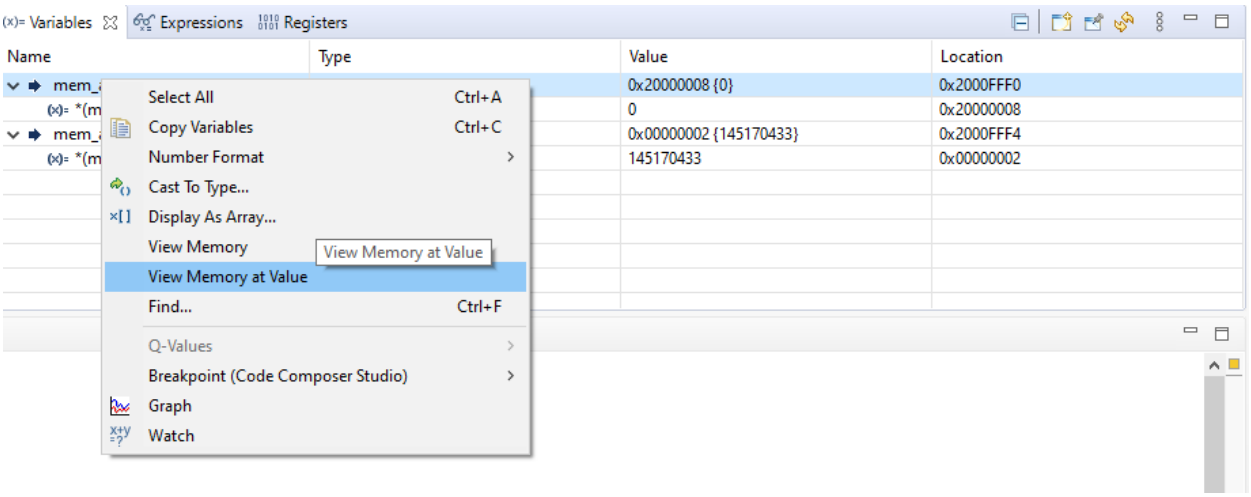
WARNING: if you request too much memory, your application will crash and may also crash the entire system.

If malloc completes successfully, the *mem_address* pointer can be passed to test function ~~subroutine~~, the *test()* function can access the block of memory as if the block was an array (since it actually is).

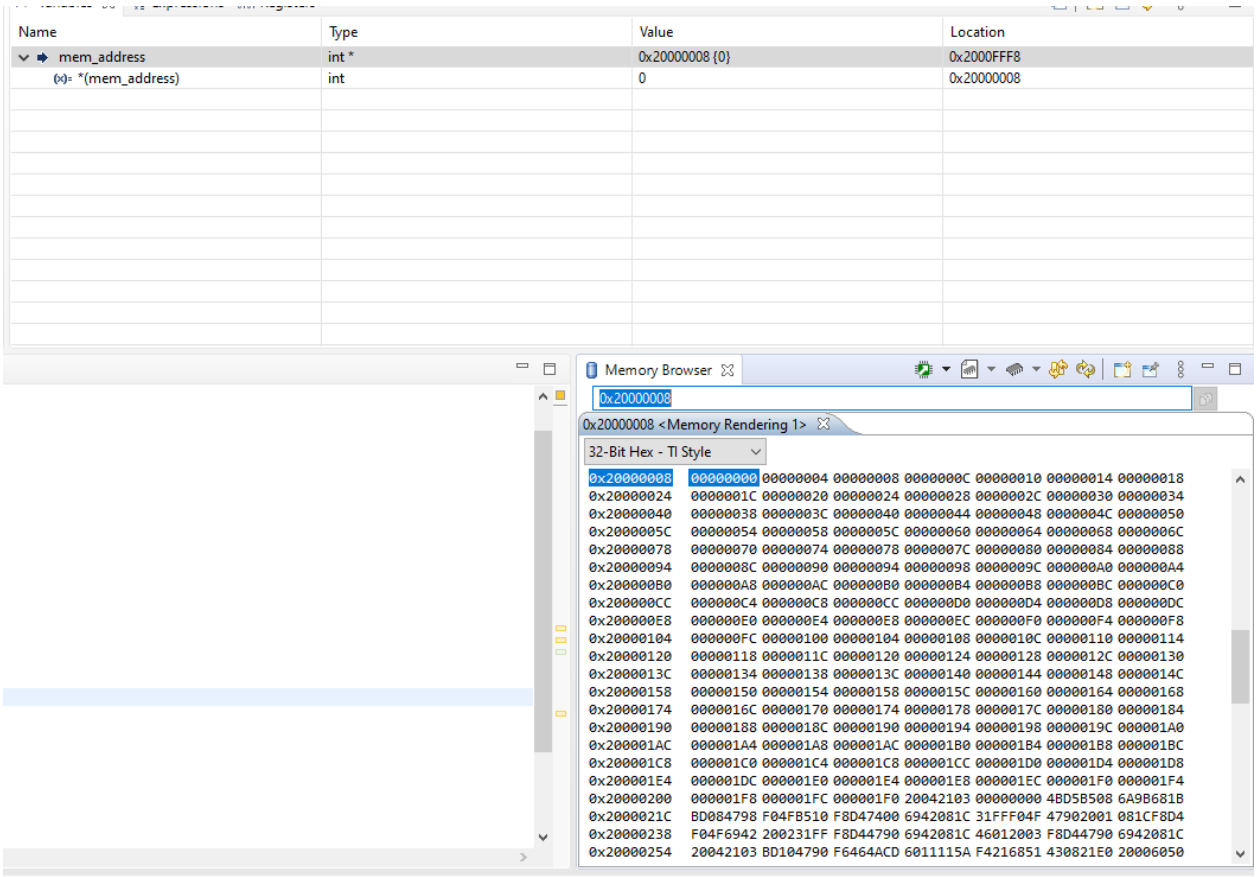
Build and Debug the code. Step until you read the ffree command. You should see something like this in the Variables monitor window:



mem_address is at 0x2000FFF8 but is holding the value 0x20000008, which is point to a value 0. If you want to see the entire block of memory, right click on the mem_address Value and then select View Memory at Value



You can now view the entire block of memory:

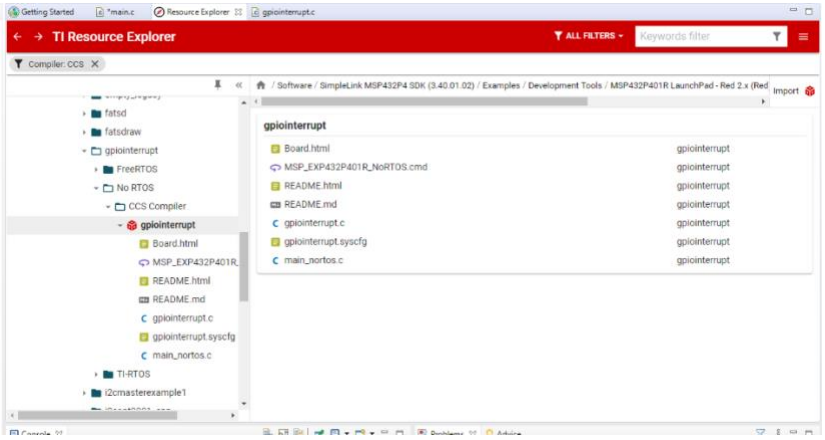


What the Memory Browser is showing you is the values, in Hex, of the data held in the memory. The values in the left hand column is the address of the first value in that row, each additional element in the row is the next data value. In this case, since you just used this particular block of memory and put each value as a multiple of 4, you can see this in the memory. The first value is 0x00000000, the next is 0x00000004, and so on.

Now step past the Free command. You will see nothing change in the Memory views, even though the Memory has been returned to the system. In the C programming language there is no attempt to set the value of memory to 0 when it is not in use, the memory retains whatever the previous values were. This is one reason to initialize variables, that way you can be assured that you don't run into a problem were your program is using data from some other program's execution.

It is important to note that there is no indication from the system that the memory allocated at mem_address has been deallocated. If you as a programmer don't realize this you can cause all kinds of defects (as I noted above, Pointers can be the cause of many serious defects.) The normal way to solve this problem is to set the mem_address to 0 after deallocating the memory.

3) The third way you may want to use a Pointer is when trying to dynamically (during run time) decide what function to execute. An excellent example of this is using hardware interrupts in your program. These hardware interrupts are used to indicate some hardware event has occurred and that your program needs to respond. So let's look at some quite different code. To access this code import the gpiointerrupt project from the Resource Explorer:



Now select to edit the gpiointerrupt.c file:

```
/*
 *  ======== gpiointerrupt.c ========
 */
#include <stdint.h>
#include <stddef.h>

/* Driver Header files */
#include <ti/drivers/GPIO.h>

/* Driver configuration */
#include "ti_drivers_config.h"

/*
 *  ======== gpioButtonFxn0 ========
 *  Callback function for the GPIO interrupt on CONFIG_GPIO_BUTTON_0.
 */
void gpioButtonFxn0(uint_least8_t index)
{
    /* Clear the GPIO interrupt and toggle an LED */
```

```
    GPIO_toggle(CONFIG_GPIO_LED_0);
}

/*
 *  ======== gpioButtonFxn1 ========
 *  Callback function for the GPIO interrupt on CONFIG_GPIO_BUTTON_1.
 *  This may not be used for all boards.
 */
void gpioButtonFxn1(uint_least8_t index)
{
    /* Clear the GPIO interrupt and toggle an LED */
    GPIO_toggle(CONFIG_GPIO_LED_1);
}

/*
 *  ======== mainThread ========
 */
void *mainThread(void *arg0)
{
    /* Call driver init functions */
    GPIO_init();

    /* Configure the LED and button pins */
    GPIO_setConfig(CONFIG_GPIO_LED_0, GPIO_CFG_OUT_STD | GPIO_CFG_OUT_LOW);
    GPIO_setConfig(CONFIG_GPIO_LED_1, GPIO_CFG_OUT_STD | GPIO_CFG_OUT_LOW);
    GPIO_setConfig(CONFIG_GPIO_BUTTON_0, GPIO_CFG_IN_PU | GPIO_CFG_IN_INT_FALLING);

    /* Turn on user LED */
    GPIO_write(CONFIG_GPIO_LED_0, CONFIG_GPIO_LED_ON);

    /* install Button callback */
    GPIO_setCallback(CONFIG_GPIO_BUTTON_0, gpioButtonFxn0);

    /* Enable interrupts */
    GPIO_enableInt(CONFIG_GPIO_BUTTON_0);

    /*
     *  If more than one input pin is available for your device, interrupts
     *  will be enabled on CONFIG_GPIO_BUTTON1.
     */
    if (CONFIG_GPIO_BUTTON_0 != CONFIG_GPIO_BUTTON_1) {
        /* Configure BUTTON1 pin */
        GPIO_setConfig(CONFIG_GPIO_BUTTON_1, GPIO_CFG_IN_PU |
GPIO_CFG_IN_INT_FALLING);

        /* Install Button callback */
        GPIO_setCallback(CONFIG_GPIO_BUTTON_1, gpioButtonFxn1);
        GPIO_enableInt(CONFIG_GPIO_BUTTON_1);
    }

    return (NULL);
}
```

You'll notice the two functions declared above the *mainThread* function. Each of these is a function you will want to execute when an interrupt occurs. Configuring this happens using this command:

```
/* install Button callback */
GPIO_setCallback(CONFIG_GPIO_BUTTON_0, gpioButtonFxn0);
```

Notice that the gpioButtonFxn0 does not have any parenthesis. This is an example of using the function name as a pointer. The system now knows that when the GPIO_BUTTON_0 is pressed (the switch on the side of the MSP432P401R), then the function gpioButtonFxn0 is to be executed. The concept of a function pointer is an extension of the concept of using a pointer to point to a variable. However, instead of the pointer pointing to a memory location that holds a data value, the pointer points to a location in memory that holds program instructions. Much like an array name "points" or holds the first location in memory that holds the data for the array, the function name "points" or holds the value of the first location in memory that holds the program instructions for that function.

You can build and Debug As this program and see the LEDs change when you press the switches, which cause the hardware interrupts.