

COMP1511/1911 Programming Fundamentals

Week 5 Lecture 2

Pointers

Last Lecture

- Was a public holiday - I hope you watched the recording
- Part 1
 - A larger array program (to help with assn1)
- Part 2
 - Strings recap
 - Array of strings and Command Line Arguments (new content)
 - Multi-file Programs (new content).

Public Holiday

Tuts/labs

- Students in monday classes please book and attend another class for week 5.
- [Link to book here](#) COMP1511 pass key
- [Timetable](#)

Next Week is Flex Week

There are no lectures or tut/labs next week.

But there is your assn1
and lab 5 to do

So there are:

- help sessions!
- revision sessions!



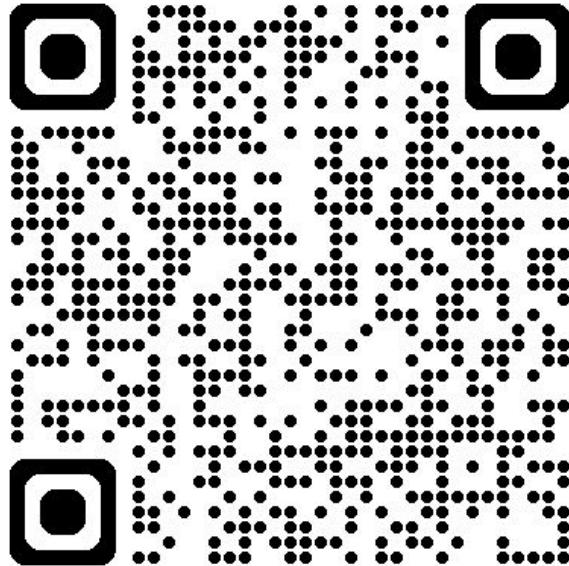
Today's Lecture

- Pointers!!
- Memory and the stack

Please keep the noise down in the lecture hall so everyone can hear properly here and online!

Link to Week 5 Live Lecture Code

https://cgi.cse.unsw.edu.au/~cs1511/24T3/live/week_5/



Memory and Addresses

- **Memory** is effectively a gigantic array of bytes.
- Memory **addresses** are effectively an index to this array of bytes.
- They are usually written in hexadecimal
- Real addresses on our system would be 8 bytes and look something like
 - 0x7ffcaa98655c

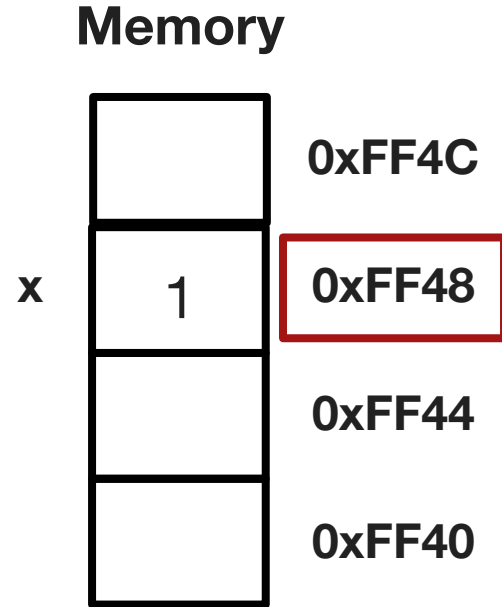
Memory



Memory and Variables

- During execution program **variables** are stored in memory.
- Each variable is stored at a particular address.

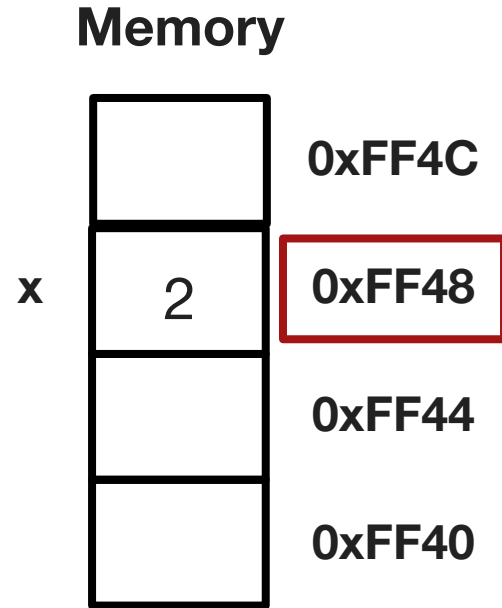
```
// In this scenario,  
// x is stored at address 0xFF48  
int x = 1;
```



Memory and Variables

- During execution program **variables** are stored in memory.
- Each variable is stored at a particular address.

```
// In this scenario,  
// x is stored at address 0xFF48  
int x = 1;  
x++;
```

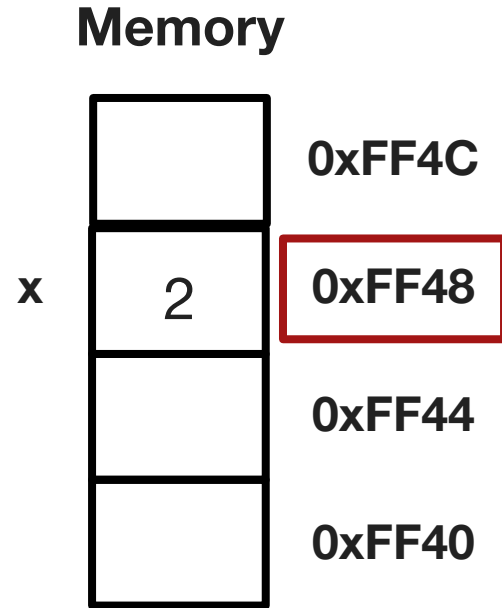


Even though the value in x has changed, the address is the same

The Address of Operator

- We can get the address of a variable using the address of operator &

```
int x = 2;
// Print the address of x
// In this scenario it would print 0xFF48
printf("%p", &x);
```

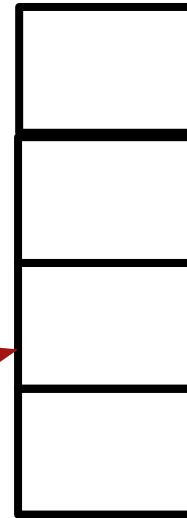


Addresses

- We have seen the address of operator before
- We tell scanf the address of our variable so it can go and put the data into the correct memory location for us
 - Like giving your address to pizza shop so they know where to deliver your food to.

```
int y;  
scanf("%d", &y);
```

Memory



0xFF4C

0xFF48

0xFF44

0xFF40

Is there a way to store an address in a variable?

Declaring a Pointer

- Pointers are variables that can store memory addresses
- To declare a pointer variable you specify what type the pointer points to and use an asterisk to indicate it is a pointer.
- E.g `type_pointing_to *pointer_variable_name;`

```
int *number_ptr;  
double *real_ptr;  
char *my_ptr;  
struct person *student_ptr;
```

Initialising a Pointer

- To initialise a pointer, we assign it the address of a variable

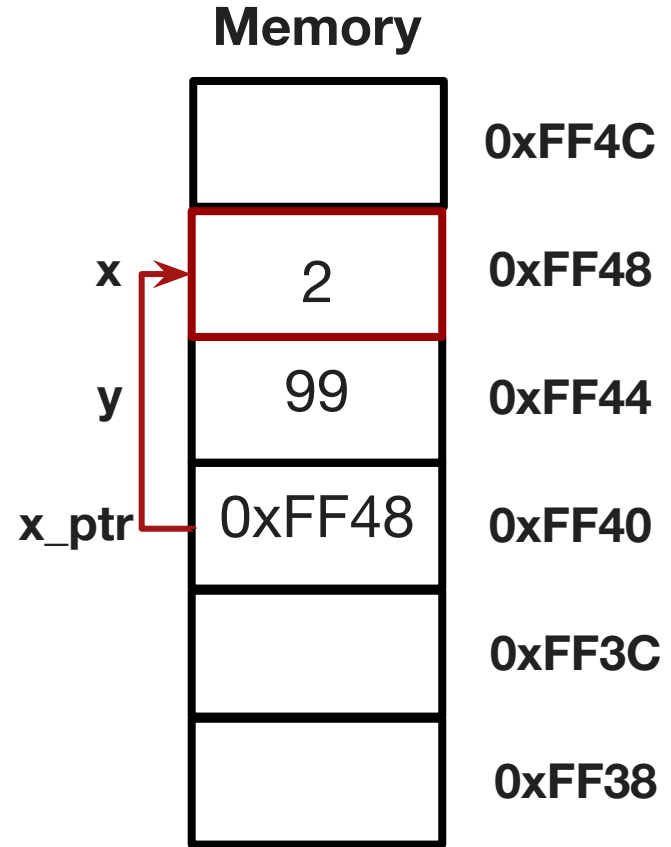
```
int x = 2;
// number_ptr is declared
// and initialised and
// contains the address
// of int variable x
int *x_ptr = &x;
```

```
double y = 1.5;
// real_ptr declared
double *real_ptr;
// real_ptr is initialised
// and contains the
// address of double
// variable y
real_ptr = &y;
```

Declaring and Initialising Pointers

```
int x = 2;
int y = 99;
// x_ptr now contains address of x
// which in this scenario is
// 0xFF48
int *x_ptr = &x;
```

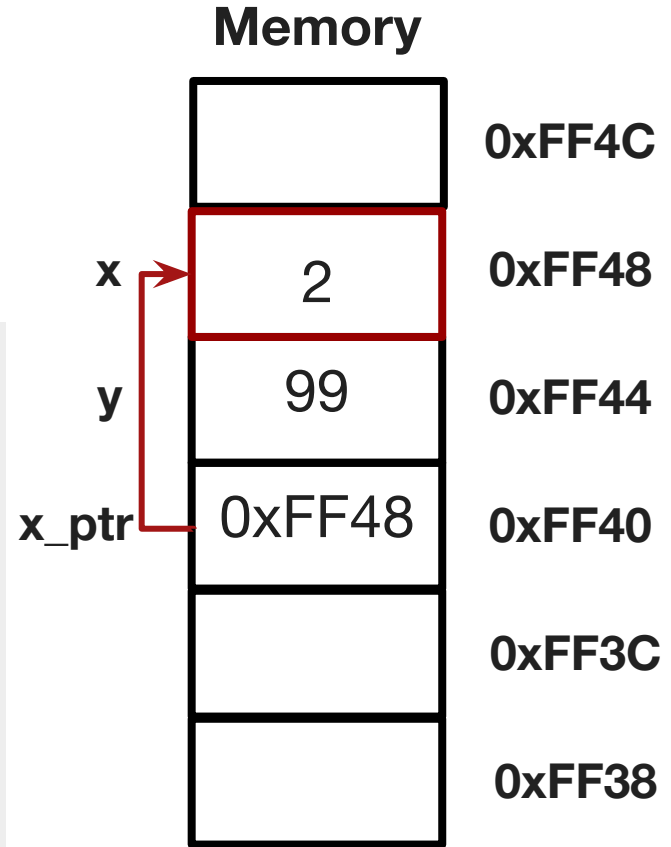
We say `x_ptr` references `x` or
`x_ptr` points to `x`



Dereference operator

- The dereference operator is *
 - This accesses the value at the address that the pointer variable holds

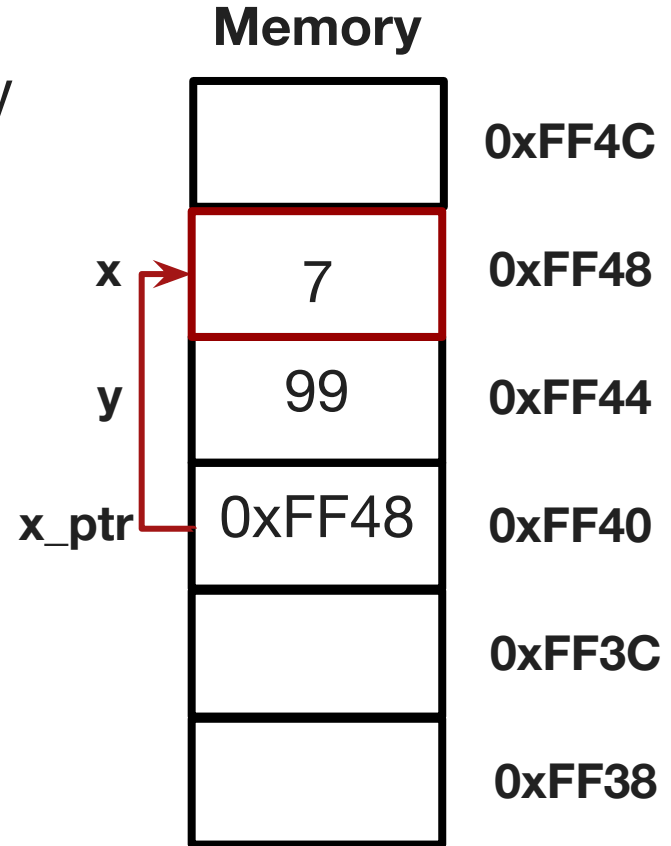
```
int x = 2;  
int y = 99;  
int *x_ptr = &x;  
// *x_ptr will go to address  
// 0xFF48 and get the value 2  
printf("%d\n", *x_ptr); //prints 2
```



Indirectly modify a variable

- We can use pointers to indirectly modify variables

```
int x = 2;
int y = 99;
int *x_ptr = &x;
// goes to address 0xFF48 and
// sets the value to 7
// x now has the value 7!
*x_ptr = 7;
```



Pointers: Putting it all together

1. Declare a pointer with a *
 - this is where you specify what type the pointer points to and get a chunk of memory for your pointer variable

```
int x = 42;
// Declare a pointer
int *number_pointer;

// Initialise pointer
number_pointer = &x;

//dereference pointer to get
//42 so z is equal to 43
int z = *number_pointer + 1;
```

Pointers: Putting it all together

2. Initialise pointer

- assign the address to the variable potentially using the **address of operator &**

```
int x = 42;
// Declare a pointer
int *number_pointer;

// Initialise pointer
number_pointer = &x;

//dereference pointer to get
//42 so z is equal to 43
int z = *number_pointer + 1;
```

Pointers: Putting it all together

3. Dereference a pointer
 - using the dereference operator *
 - go to the address that this pointer variable is assigned and access what is at that address

```
int x = 42;
// Declare a pointer
int *number_pointer;

// Initialise pointer
number_pointer = &x;

//dereference pointer to get
//42 so z is equal to 43
int z = *number_pointer + 1;
```

Pointer Coding Demo.

pointer_intro.c

pointer_exercise.c

What will this print out?

```
int x = -7;
int y = 5;

int *ptr1 = &y;
int *ptr2 = &x;

int z = *ptr1 + y;
*ptr2 = z - 1;
printf("%d %d %d\n", x, y, z);

ptr2 = ptr1;
printf("%d %d\n", *ptr1, *ptr2);
```

The NULL Pointer

- Sometimes we initialise our pointer variables with a special value meaning that they don't point to anything yet.
 - We use the special value `NULL` to do this
- You will get a run time error if you dereference a `NULL` pointer

```
int *my_ptr = NULL;
// Dereferencing a NULL
// pointer will cause a
// run time error
printf("%d\n", *my_ptr);
```

```
int *my_ptr = NULL;
// Check for NULL first if
// it might be NULL
if (my_ptr != NULL) {
    printf("%d\n", *my_ptr);
}
```

Pointers to structs

Remember that when we access members of a struct we use a `.`

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

```
int main(void) {  
    struct point p;  
    p.x = 10;  
    p.y = 9;  
}
```


Pointers to structs

Accessing pointers to structs with . gets messy.

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

```
int main(void) {  
    struct point p;  
    struct point p_ptr = &p;  
    (*p_ptr).x = 10;  
    (*p_ptr).y = 9;  
}
```

Pointers to structs

Instead we can use -> notation

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

```
int main(void) {  
    struct point p;  
    struct point p_ptr = &p;  
    (*p_ptr).x = 10;  
    (*p_ptr).y = 9;  
    // The same but easier  
    p_ptr->x = 10;  
    p_ptr->y = 9;  
}
```

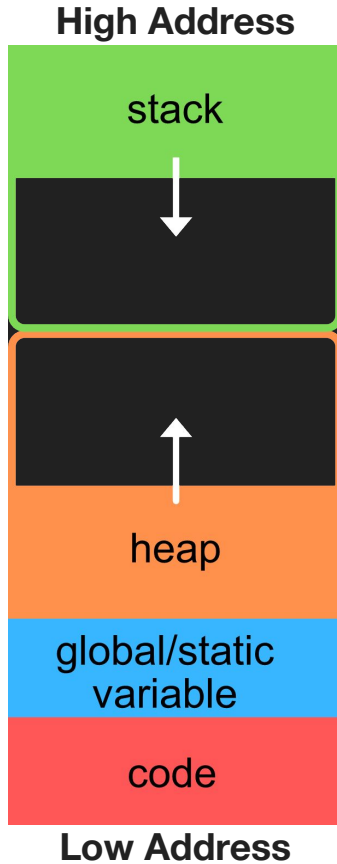
What is the point of all of this?

What will this print?

```
int main(void) {  
    int x = 2;  
    int y = 5;  
    printf("%d %d\n", x, y);  
    update(x,y);  
    printf("%d %d\n", x, y);  
    swap(x, y);  
    printf("%d %d\n", x, y);  
    return 0;  
}
```

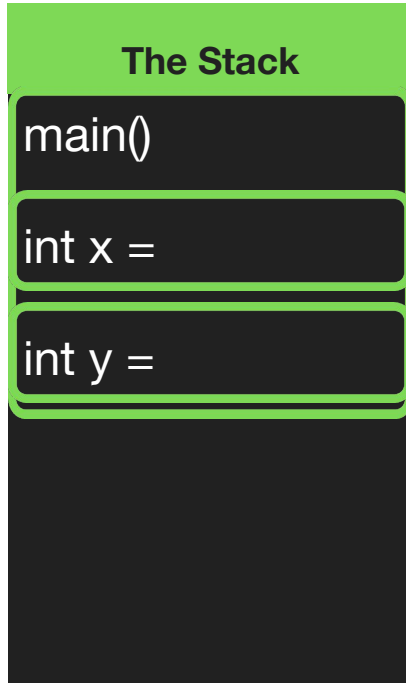
```
void update(int x, int y) {  
    x = x + 1;  
    y = y - 1;  
}  
  
void swap(int x, int y) {  
    int tmp = x;  
    x = y;  
    y = tmp;  
}
```

More about Memory: The Stack



- Stack memory stores data about each function your program calls.
- When a function is called, data gets pushed onto the stack such as
 - local variables
 - where to return to when the function finishes
- Once your function finishes, its data including variables will **automatically** be removed from the stack

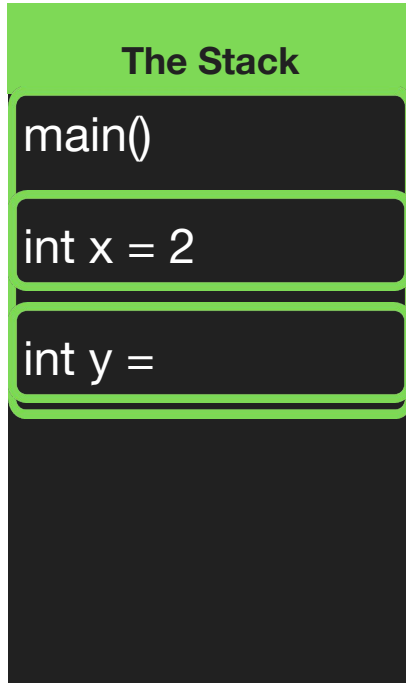
More about Memory: The Stack



```
void update(int x, int y) {  
    x = x + 1;  
    y = y - 1;  
}
```

```
int main(void) {  
    int x = 2;  
    int y = 5;  
    update(x, y);  
    printf("%d %d\n", x, y);  
    return 0;  
}
```

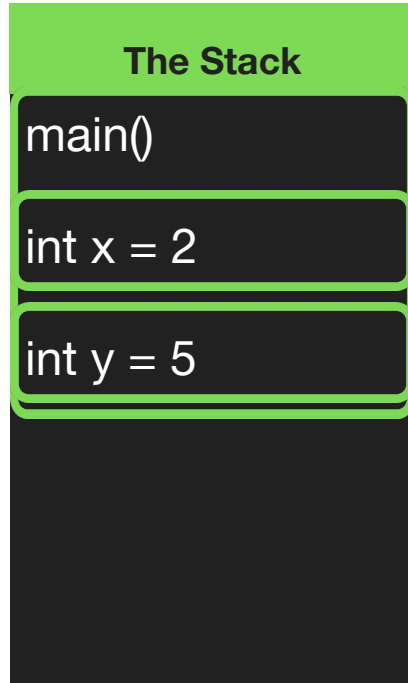
More about Memory: The Stack



```
void update(int x, int y) {
    x = x + 1;
    y = y - 1;
}
```

```
int main(void) {
    int x = 2;
    int y = 5;
    update(x, y);
    printf("%d %d\n", x, y);
    return 0;
}
```

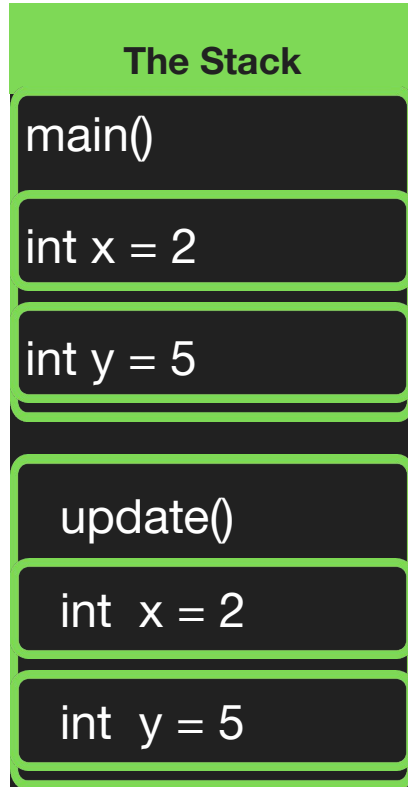
More about Memory: The Stack



```
void update(int x, int y) {
    x = x + 1;
    y = y - 1;
}
```

```
int main(void) {
    int x = 2;
    int y = 5;
    update(x, y);
    printf("%d %d\n", x, y);
    return 0;
}
```


More about Memory: The Stack



```
void update(int x, int y) {
```

```
    x = x + 1;
```

```
    y = y - 1;
```

```
}
```

```
int main(void) {
```

```
    int x = 2;
```

```
    int y = 5;
```

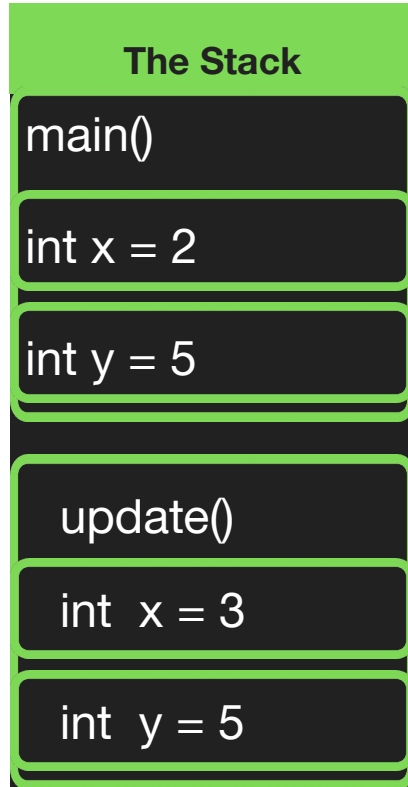
```
    update(x, y);
```

```
    printf("%d %d\n", x, y);
```

```
    return 0;
```

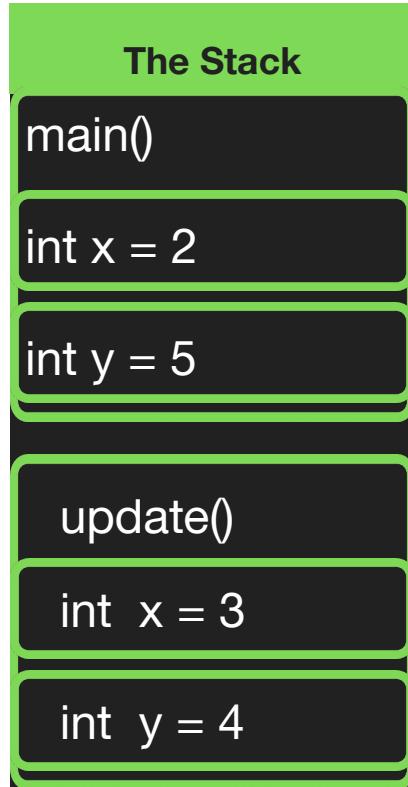
```
}
```

More about Memory: The Stack



```
void update(int x, int y) {  
    x = x + 1;  
    y = y - 1;  
}  
  
int main(void) {  
    int x = 2;  
    int y = 5;  
    update(x, y);  
    printf("%d %d\n", x, y);  
    return 0;  
}
```

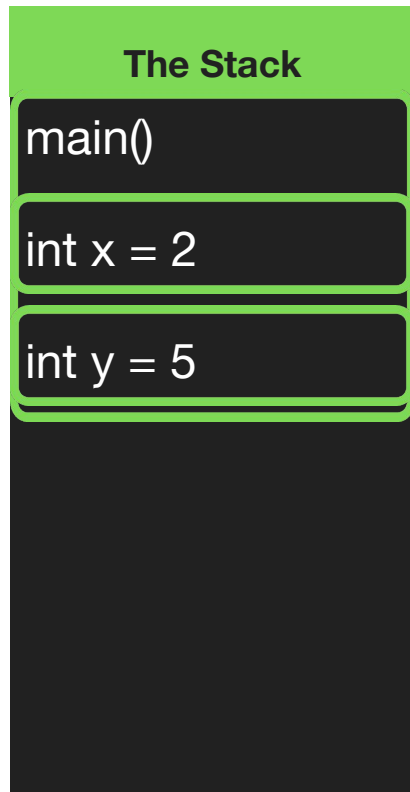
More about Memory: The Stack



```
void update(int x, int y) {
    x = x + 1;
    y = y - 1;
}

int main(void) {
    int x = 2;
    int y = 5;
    update(x, y);
    printf("%d %d\n", x, y);
    return 0;
}
```

More about Memory: The Stack



```
void update(int x, int y) {
    x = x + 1;
    y = y - 1;
}

int main(void) {
    int x = 2;
    int y = 5;
    update(x, y);
    printf("%d %d\n", x, y);
    return 0;
}
```

2 and 5 get printed

More about Memory: The Stack

The Stack

```
void update(int x, int y) {
    x = x + 1;
    y = y - 1;
}

int main(void) {
    int x = 2;
    int y = 5;
    update(x,y);
    printf("%d %d\n", x, y);
    return 0;
}
```

Functions and Pointers

- Variables and data are passed by value into functions (note: arrays are a special case we will discuss separately)
 - The function gets passed copies of the values
 - We can't change the original values from inside the function
 - The modified copies don't even exist once the function ends
- Is there anyway around this?

Functions and Pointers

- Can we pass in the addresses of variables into our functions like we do with scanf so we can modify them?
 - Yes! Then the function can go to the memory address and access and modify the original values
 - Note, we are still passing in copies of the addresses

So now we have a way of letting functions we call modify out local variables, even if they are not arrays!!

Functions and Pointers

```
int main(void) {  
    int x = 2;  
    int y = 5;  
    printf("%d %d\n", x, y);  
    update(&x, &y);  
}
```

```
void update(int *x, int *y) {  
    *x = *x + 1;  
    *y = *y - 1;  
}
```

To do this:

- Our main function would have to pass in the addresses of x and y
- Our update function would need to change to have pointer parameters since pointers can store addresses!

Functions and Pointers

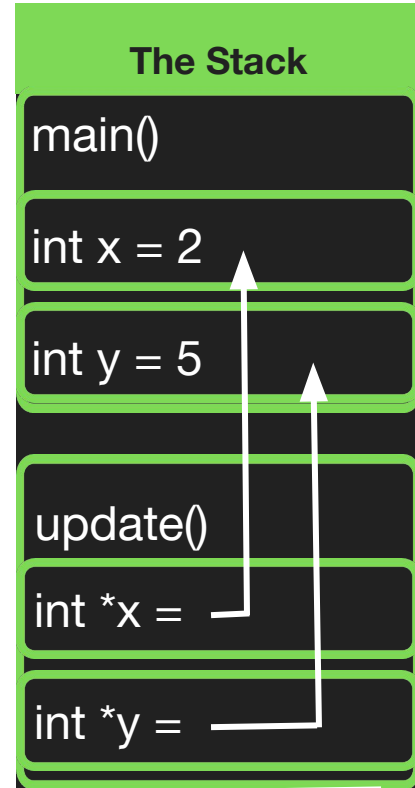
```
int main(void) {  
    int x = 2;  
    int y = 5;  
    printf("%d %d\n", x, y);  
    update(&x, &y);  
}
```

```
void update(int *x, int *y) {
```

```
    *x = *x + 1;
```

```
    *y = *y - 1;
```

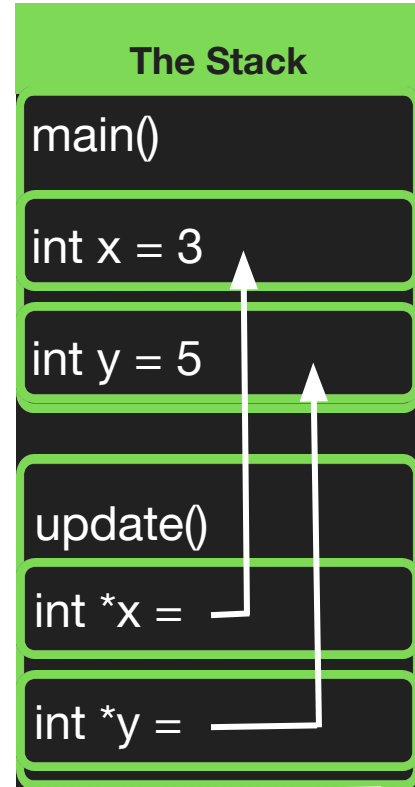
```
}
```



Functions and Pointers

```
int main(void) {  
    int x = 2;  
    int y = 5;  
    printf("%d %d\n", x, y);  
    update(&x, &y);  
}
```

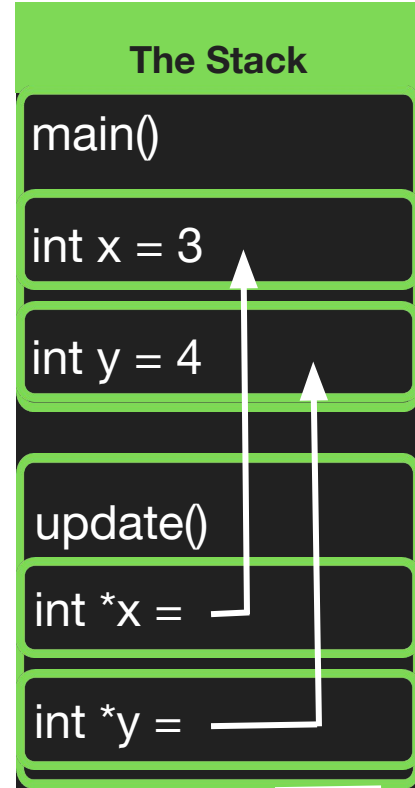
```
void update(int *x, int *y) {  
    *x = *x + 1;  
    *y = *y - 1;  
}
```



Functions and Pointers

```
int main(void) {  
    int x = 2;  
    int y = 5;  
    printf("%d %d\n", x, y);  
    update(&x, &y);  
}
```

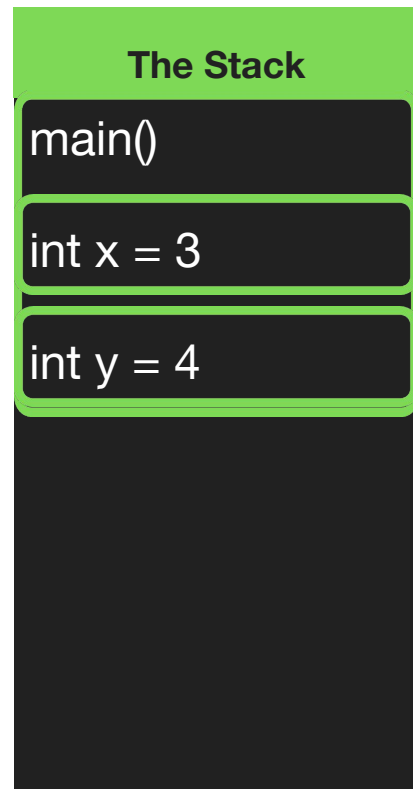
```
void update(int *x, int *y) {  
    *x = *x + 1;  
    *y = *y - 1;  
}
```



Functions and Pointers

```
int main(void) {  
    int x = 2;  
    int y = 5;  
    printf("%d %d\n", x, y);  
    update(&x, &y);  
}
```

```
void update(int *x, int *y) {  
    *x = *x + 1;  
    *y = *y - 1;  
}
```



Exercise: Now how can we modify swap?

```
int main(void) {  
    int x = 2;  
    int y = 5;  
    printf("%d %d\n", x, y);  
    update(&x, &y);  
    printf("%d %d\n", x, y);  
    swap(x, y);  
    printf("%d %d\n", x, y);  
    return 0;  
}
```

```
void update(int *x, int *y) {  
    *x = *x + 1;  
    *y = *y - 1;  
}
```

```
void swap(int x, int y) {  
    int tmp = x;  
    x = y;  
    y = tmp;  
}
```

Exercise: Pointers to structs

```
void update(struct point p) {
    p.x = p.x + 1;
    p.y = p.y + 1;
}

int main(void) {
    struct point p;
    p.x = 10;
    p.y = 9;
    update(p);
    printf("(%d,%d)\n", p.x, p.y);
}
```

What will this do?
How can we fix it?

Pointers to structs

```
struct point update(struct point p) {  
    p.x = p.x + 1;  
    p.y = p.y + 1;  
    return p;  
}  
  
int main(void) {  
    struct point p;  
    p.x = 10;  
    p.y = 9;  
    p = update(p);  
    printf("( %d, %d) \n", p.x, p.y);  
}
```

An option without pointers could be to return the updated point.

Pointers to structs

```
void update(struct point *p) {
    p->x = p->x + 1;
    p->y = p->y + 1;
}

int main(void) {
    struct point p;
    p.x = 10;
    p.y = 9;
    update(&p);
    printf("(%d,%d)\n", p.x, p.y);
}
```

We could also pass
in a pointer and
update the original
copy

Functions and Arrays

- When we pass an array into a function, the address of the start of the array gets passed in by default!
 - It does not send in a copy of all of the data
 - Just a copy of the address of the first element!
 - This is why we can modify the contents of our array arguments

```
// This WILL modify the contents of the num array
void increment_all(int nums[], int length) {
    for (int i = 0; i < length; i++) {
        nums[i] = nums[i] + 1;
    }
}
```

Coding time Arrays and Pointers

array_addresses.c

array_arguments.c

Can we return a pointer from a function?

We can return Pointers from Functions

But we can't do this? Why?

```
int *f(void) {  
    int x = 3;  
    return &x;  
}
```

And we can't do this? Why?

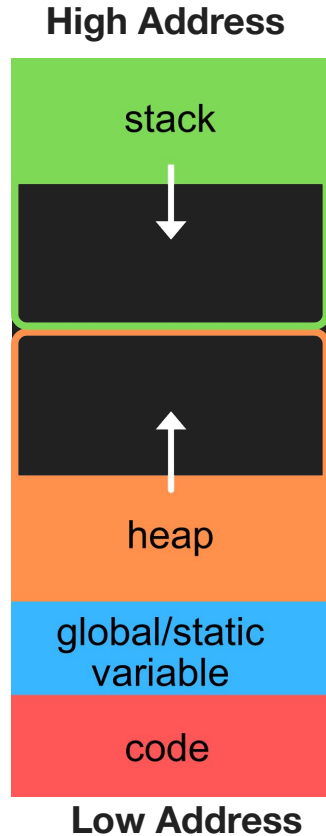
```
int *f(void) {  
    int numbers[] = {1, 2, 3};  
    return numbers;  
}
```

We can't return the address of a local variable

Local variables live on the stack

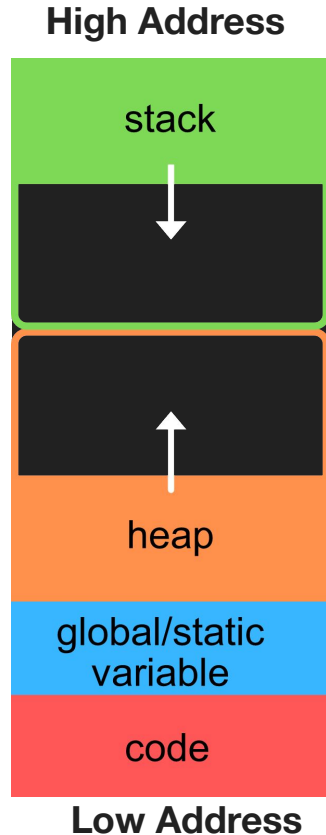
When the function returns it does not exist any more!

The Heap



- We would like to be able to create arrays within functions and return them
- We would also like to create arrays whose sizes are not know until runtime
- Can the heap allow us to do this?
 - Yes!!!!

The Heap



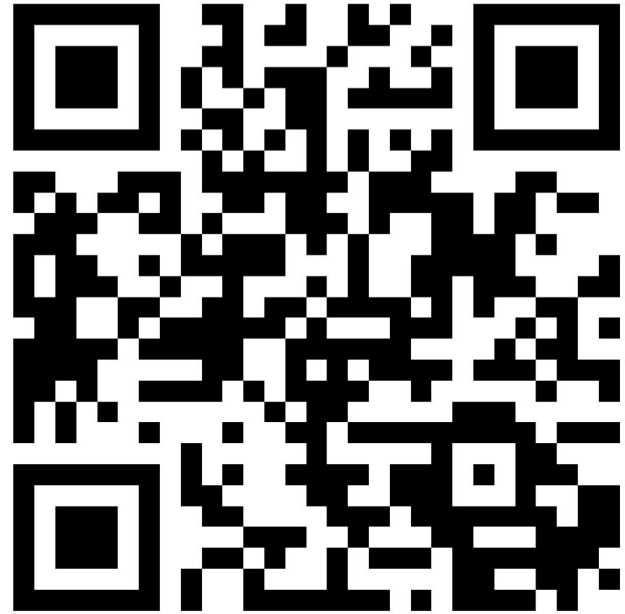
- Unlike stack memory, heap memory is allocated by the programmer
- It won't be deallocated until it is explicitly freed by the programmer
- You now have the power to control memory on the heap!
- With power comes heaps of responsibility

Feedback Please!

Your feedback is valuable!

If you have any feedback from today's lecture, please follow the link below or use the QR Code.

Please remember to keep your feedback constructive, so I can action it and improve your learning experience.



<https://forms.office.com/r/0SvCZ5LDq2>

What did we learn today?

- Pointers
- Pointers to structs with ->
- Pointers and Functions
- Pointers and Arrays
- Memory and the Stack

Have an amazing Flex week.

See you back in week 7 where we will learn about the Heap, malloc and dynamic arrays and...

Linked Lists!



Reach Out

Content Related Questions:
Forum

Admin related Questions email:
cs1511@unsw.edu.au

