

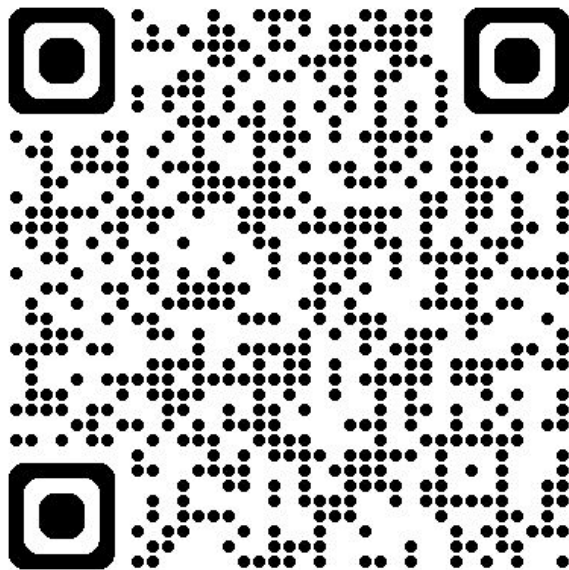
COMP1511/1911 Programming Fundamentals

Week 5 Lecture 2

Pointers

Link to Week 5 Live Lecture Code

https://cgi.cse.unsw.edu.au/~cs1511/26T3/code/week_5/



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!



Revision Sessions

Revision sessions in Week 6

Details coming soon...

Last Lecture

- A larger array program (to help with assn1)
- Strings and array of strings kahoot recap
- String programming exercise.c

Today's Lecture

- Command Line Arguments
- Pointers!!
- Memory and the stack

What are Command Line Arguments?

Command Line Arguments

- So far, we have only given input to our program after we have started running that program (using `scanf()` or `fgets()`)
- Our main function prototype has always been `int main(void) ;`
- Command line arguments allow us to give inputs to our program at the time that we start running it! E.g.

```
$ gcc prog.c -o prog
$ ./prog argument1 argument2 argument3 argument4
$ ./prog 123 hello
```

Command Line Arguments

- To use command line arguments you need to change your main function prototype to

```
int main(int argc, char *argv[])
```

- **argc**
 - a counter for how many command line arguments you have (including the program name)
- **char *argv[]**
 - an array of the different command line arguments
 - each command line argument is a string (an array of char)

Command Line Arguments

- If we ran our program as follows:

```
$ ./prog 123 dog "hello world"
```

- argc would be equal to 4
- argv would be an array of strings we can visualise as follows:

“./prog”	“123”	“dog”	“hello world”
0	1	2	3

Command Line Arguments

```
int main(int argc, char *argv[]) {  
    printf("There are %d command line arguments\n", argc);  
  
    // argv[0] is always the program name  
    printf("This program name is %s\n", argv[0]);  
  
    // print out all arguments in the argv array  
    for (int i = 0; i < argc; i++) {  
        printf("Argument at index %d is %s\n", i, argv[i]);  
    }  
    return 0;  
}
```

Command Line Arguments

```
$ gcc -o command_line_args command_line_args.c
$ ./command_line_args 123 dog "Hello World" COMP1511
This program has 5 command line arguments
This program name is ./command_line_args
Argument at index 0 is ./command_line_args
Argument at index 1 is 123
Argument at index 2 is dog
Argument at index 3 is Hello World
Argument at index 4 is COMP1511
```

Exercise: greeting.c

```
./greeting Tina -good
```

```
Hello Tina. Have a lovely day!
```

```
./greeting Tina -evil
```

```
Hello Tina. We have been expecting you...
```

```
./greeting
```

```
Usage: ./greeting <name> [-good|-evil]
```

Converting Strings to Integers: atoi

- You may want to use your command line arguments to perform calculations, but they are strings!
- There is a function that converts strings to integers:
 - `atoi()` in the standard library: `<stdlib.h>`
 - E.g. `int x = atoi("952")`
 - Would give us a value of 952 stored in x

Converting Strings to Integers: atoi

```
int main(int argc, char *argv[]) {  
    int sum = 0;  
    for (int i = 1; i < argc; i++) {  
        sum = sum + atoi(argv[i]);  
    }  
    printf("%d is the sum of all command line args\n", sum);  
    return 0;  
}
```

Pointers and Memory

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
 - 0x is a prefix that means hexadecimal
- Real addresses on our system would be 8 bytes and look something like
 - 0x7ffcaa98655c

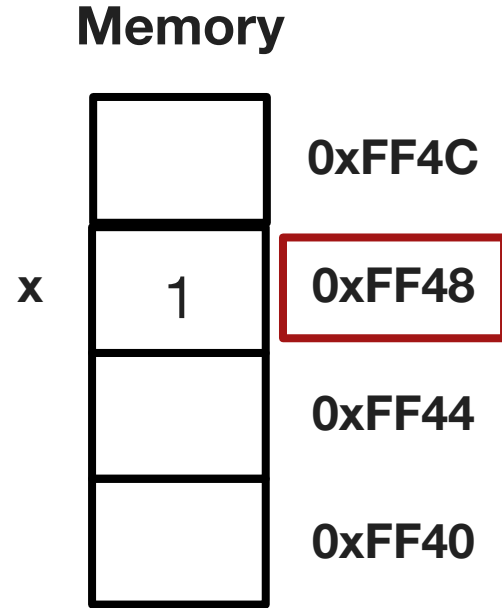
Memory



Memory and Variables

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

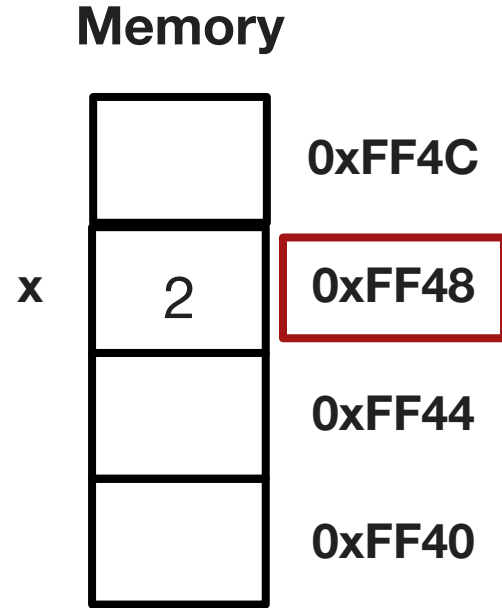
```
// In this scenario,  
// x is stored at address 0xFF48  
int x = 1;
```



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++;
```

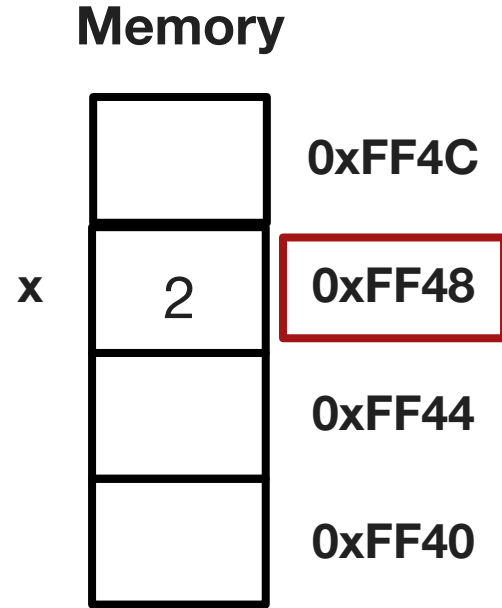


Even though the value in x has changed, the address is the same

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);
```

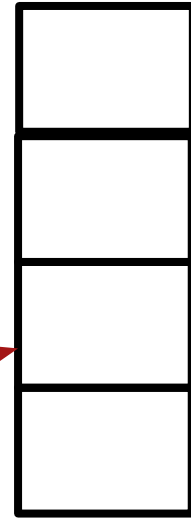


Addresses

- 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;
```

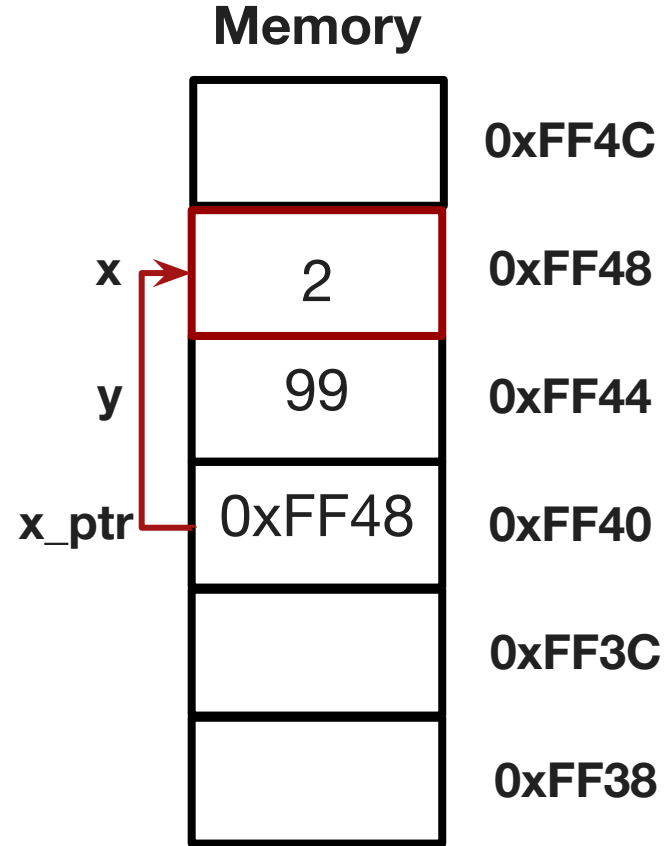
```
double y = 1.5;  
// real_ptr declared  
double *real_ptr;  
// real_ptr is initialised  
// and contains the  
// address of double  
// variable y  
real_ptr = &y;
```

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;
```

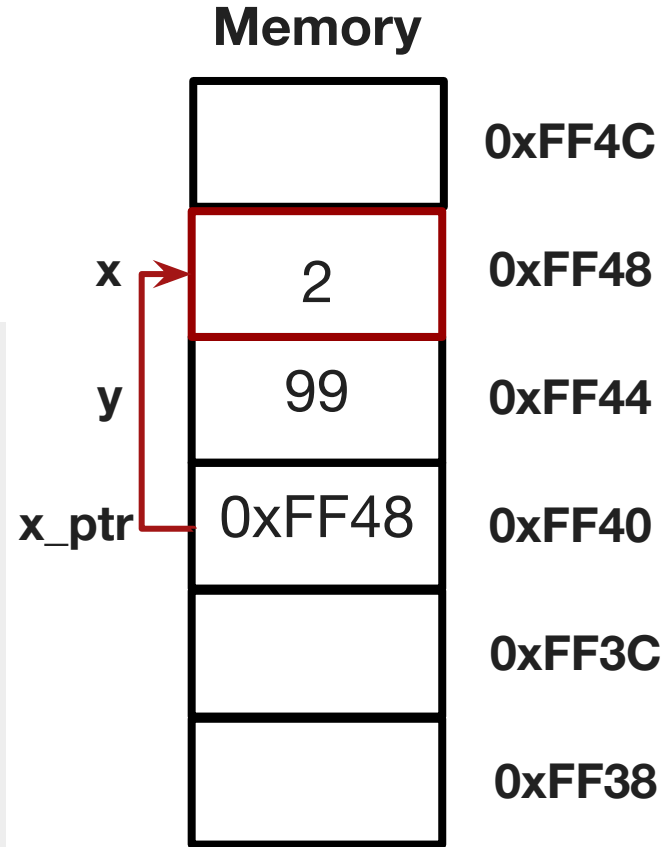
We say `x_ptr` references `x` or
`x_ptr` points to `x`



Dereference operator

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

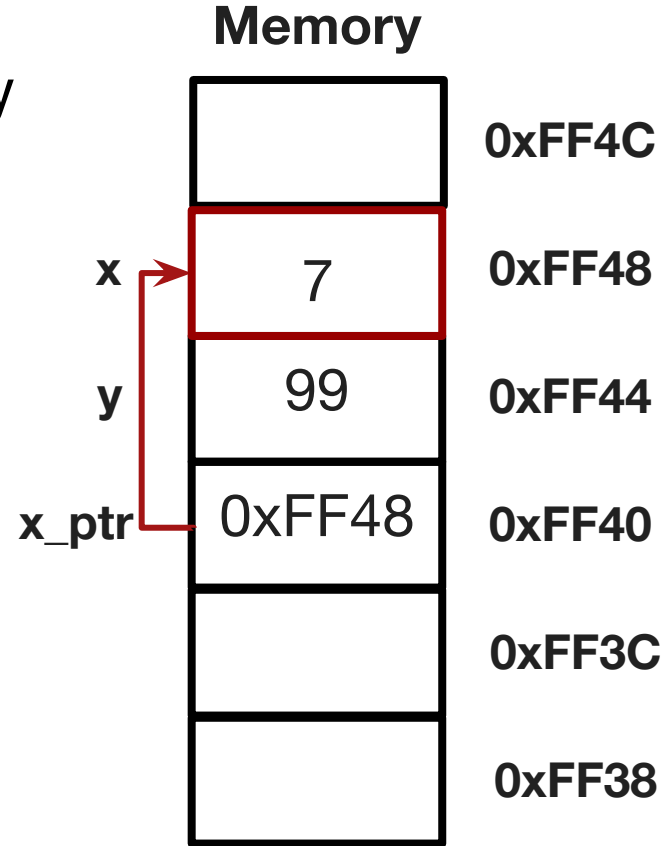
```
int x = 2;  
int y = 99;  
int *x_ptr = &x;  
// *x_ptr will go to address  
// 0xFF48 and get the value 2  
printf("%d\n", *x_ptr); //prints 2
```



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

changing_pointers.c

pointer_exercise.c

Exercise: 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);
```

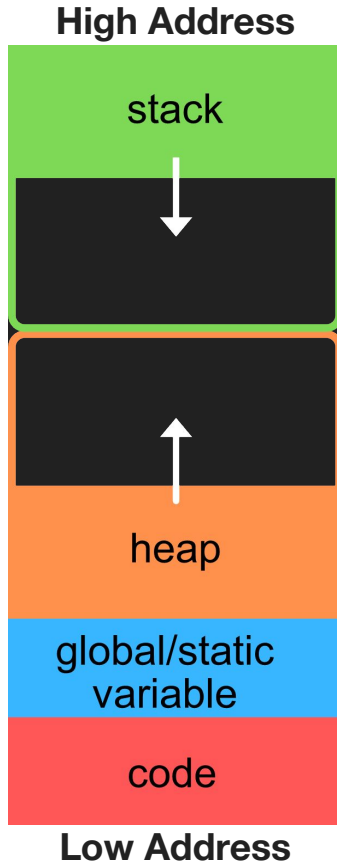
What is the point of all of this?

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;  
}
```

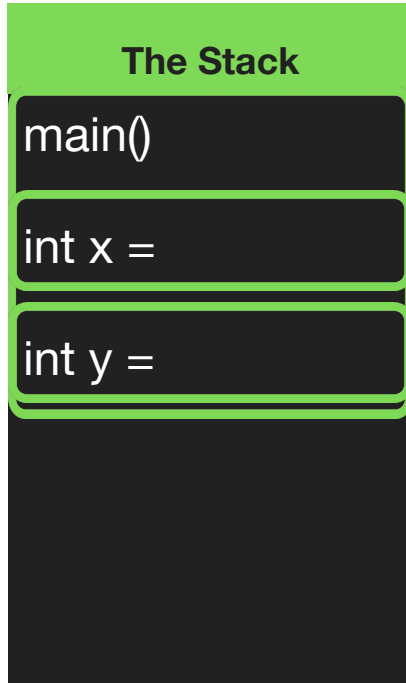
```
void update(int x, int y) {  
    x = x + 1;  
    y = y - 1;  
}  
  
void swap(int x, int y) {  
    int tmp = x;  
    x = y;  
    y = tmp;  
}
```

More about Memory: The Stack



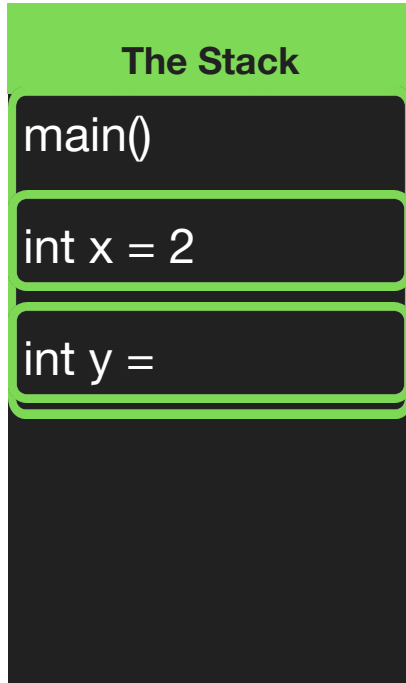
- Stack memory stores data about each function your program calls.
- When a function is called, data gets pushed onto the stack such as
 - local variables
 - where to return to when the function finishes
- Once your function finishes, its data including variables will **automatically** be removed from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

The Stack

main()

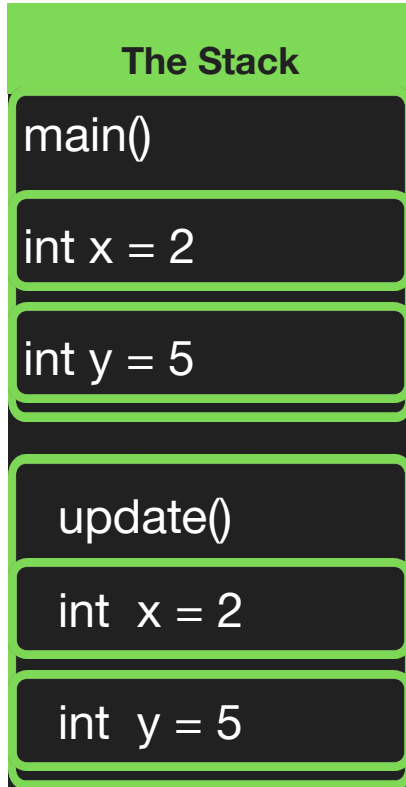
int x = 2

int y = 5

```
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;
```

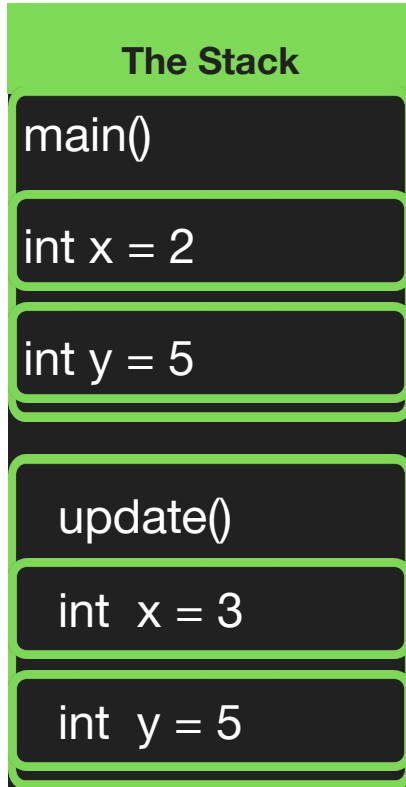
```
    update(x, y);
```

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

```
    return 0;
```

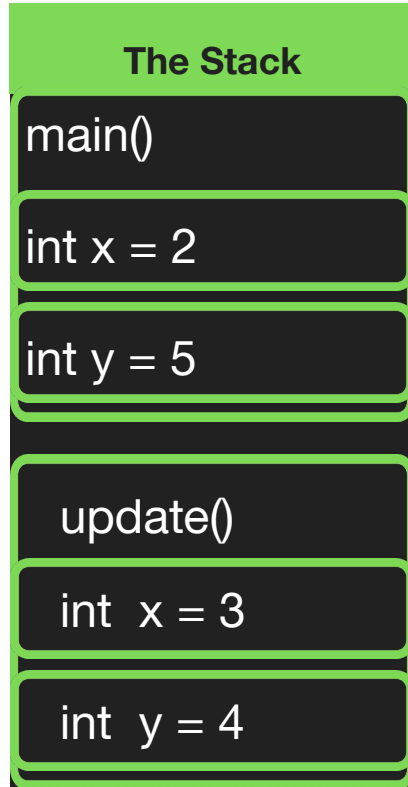
```
}
```

More about Memory: The Stack



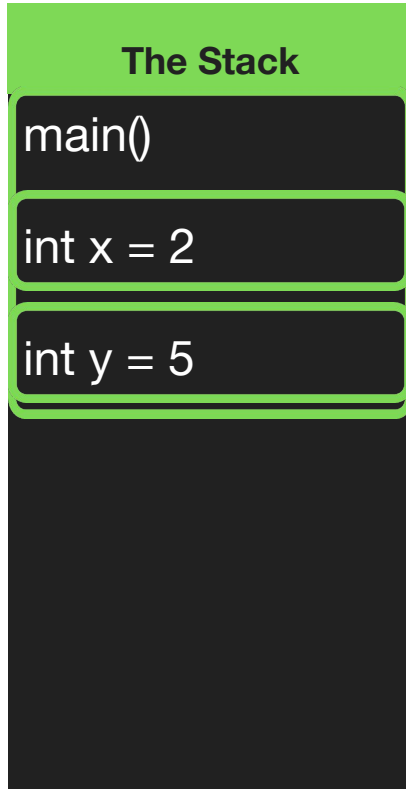
```
void update(int x, int y) {  
    x = x + 1;  
    y = y - 1;  
}  
  
int main(void) {  
    int x = 2;  
    int y = 5;  
    update(x, y);  
    printf("%d %d\n", x, y);  
    return 0;  
}
```

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 our 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;  
}
```

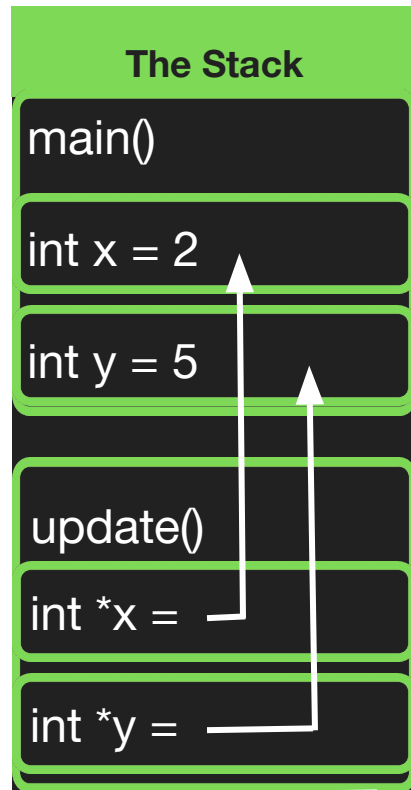
To do this:

- 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);  
}
```

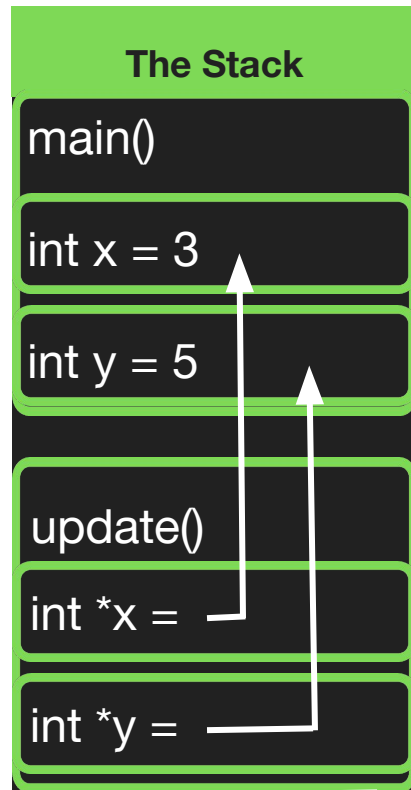
```
void update(int *x, int *y) {  
    *x = *x + 1;  
    *y = *y - 1;  
}
```



Functions and Pointers

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

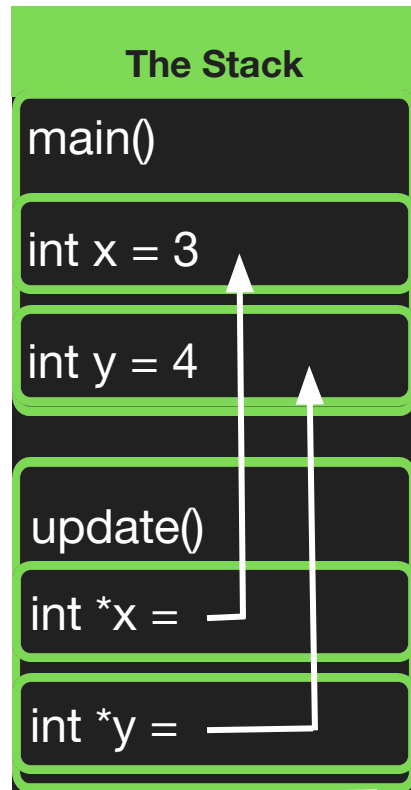
```
void update(int *x, int *y) {  
    *x = *x + 1;  
    *y = *y - 1;  
}
```



Functions and Pointers

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

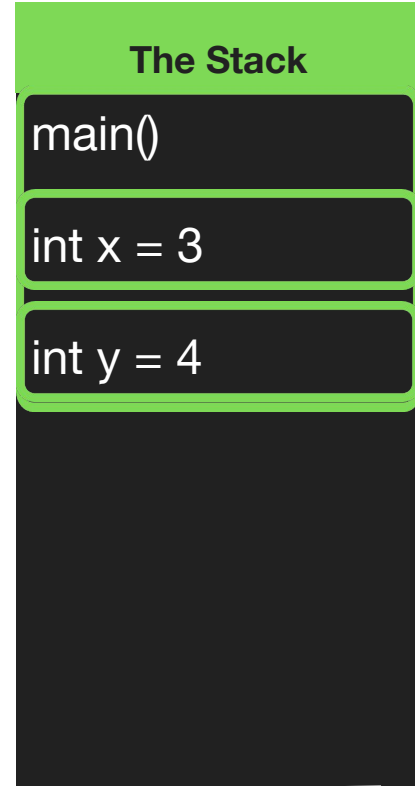
```
void update(int *x, int *y) {  
    *x = *x + 1;  
    *y = *y - 1;  
}
```



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;  
}
```

How can we work with pointers to structs?

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;  
}
```

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;
    }
}
```

Code demo: Arrays and Pointers

array_arguments.c

array_addresses.c

Exercise: What will this print?

string_pointer_exercise.c

```
char s[] = "Pointers!!!";  
char *sp = &s[1];  
printf("%c\n", *sp);  
printf("%c\n", sp[0]);  
printf("%c\n", sp[1]);  
printf("%s\n", sp);
```

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/F2nqhrNNJd>

What did we learn today?

- Command Line Arguments
- Addresses
- Pointers
- Memory and the Stack
- Pointers and Functions
- Pointers to structs with ->
- Pointers and Arrays

Have an amazing Flex week.

In week 7 we will learn about the Heap, malloc, dynamic arrays

Linked Lists!



Dr Henry Hickman

Week 7 and 8 I will be away
and the wonderful Dr Henry
Hickman will be lecturing!



Reach Out

Content Related Questions:

[Course Forum](#)

Admin related Questions email:

cs1511@unsw.edu.au



Student Support - I Need Help With...

My Feelings and Mental Health

Feeling depressed, overwhelmed
or not your usual self?



Mental Health Connect
student.unsw.edu.au/mhc



TalkCampus
student.unsw.edu.au/Talkcampus



UNSW (24/7) Mental Health Support: [+61\(2\) 9385 5418](tel:+61293855418)

Text support After Hours: [0485 826 595](tel:0485826595)



Offshore
Call 24-hour Medibank Hotline: [+61 2 8905 0307](tel:+61289050307)

Uni and Life in Australia

Stress, financial, visas,
Accommodation & More



Student Support
Indigenous Student Support

student.unsw.edu.au/advisors
nura-gili-centre-indigenous-programs

Reporting Sexual Assault/Harassment



Equity Diversity & Inclusion (EDI)

edi.unsw.edu.au/sexual-misconduct

Educational Adjustments

To manage my studies and
disability / health condition



Equitable Learning Services (ELS)

student.unsw.edu.au/els

Academic and Study Skills



Academic Skills

student.unsw.edu.au/skills

Special Consideration

Because life impacts our
studies and exams



Special Consideration

student.unsw.edu.au/special-consideration

Physical Health

Doctors visits, Vaccinations



Health Service

student.unsw.edu.au/hsu