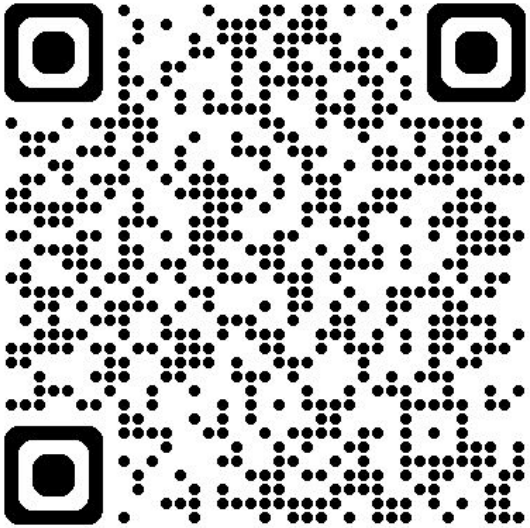
COMP1511/1911 Programming Fundamentals

Week 7 Lecture 1

Pointers, the Heap and Dynamic Arrays

Link to Week 7 Live Lecture Code

https://cgi.cse.unsw.edu.au/~cs1511/25T3/code/week_7/



This Week

Assignment 1 due today at 5pm

Remember to get support

- Help sessions
- Revision sessions
- Forum

Assignment 2

Release: Thursday 9am

Before Flex Week...

Week 5 Lecture 1:

- Arrays of strings, Command Line arguments
- 2d Array of structs with enums programming example

Week 5 Lecture 2:

- Pointers

Today's Lecture

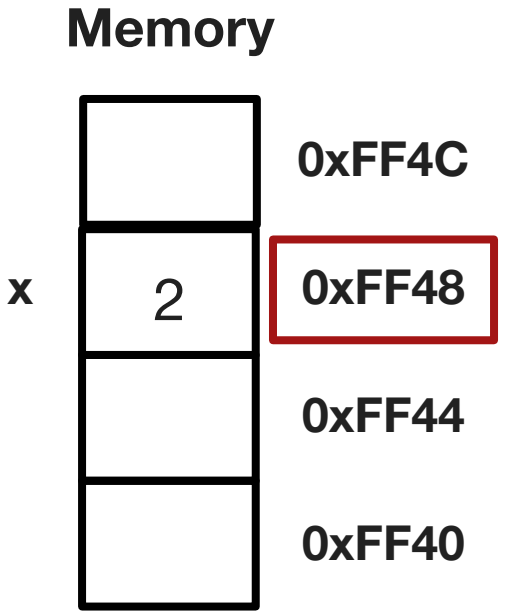
- Pointers recap
- Pointers
- Memory and the stack
- Dynamic Memory, malloc and the heap
- Multi-file Projects (if there is time)

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

The Address of Operator

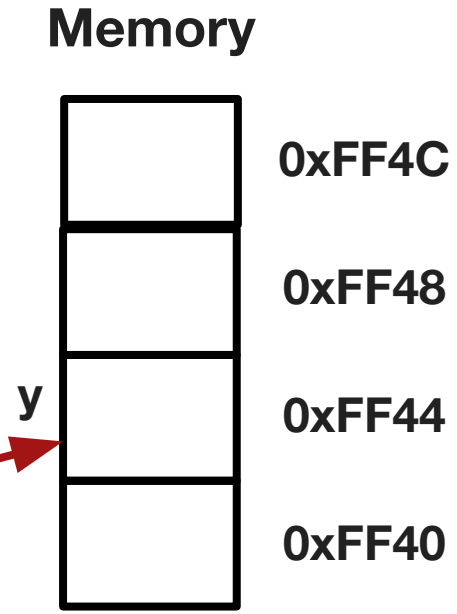
- We can get the address of a variable using the address of operator `&`




```c
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.




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

Declaring a Pointer

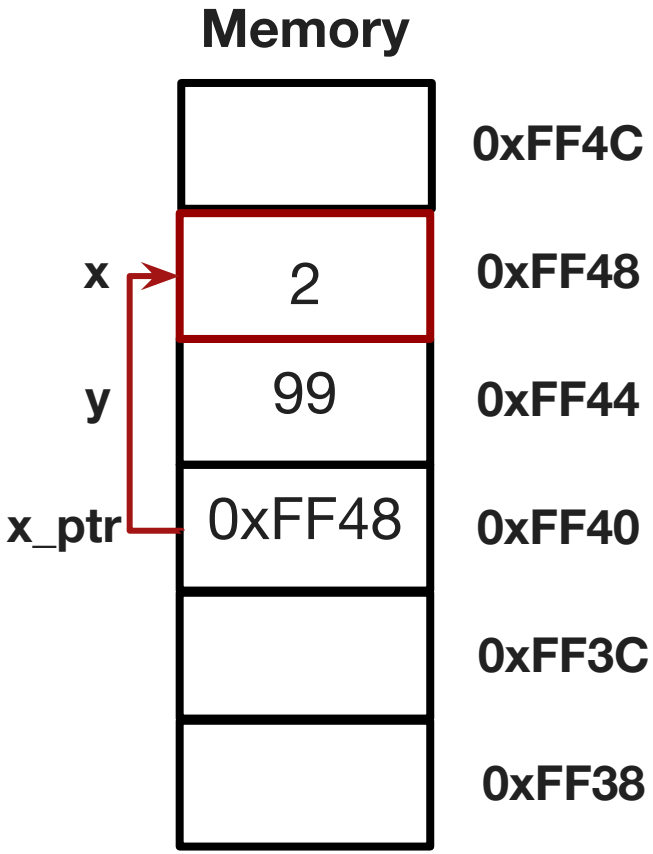
- Pointers are variables that can store memory addresses of variables
- To declare a pointer variable you specify what type the pointer points to and use an asterisk to indicate it is a pointer. E.g.
  - `type_pointing_to *pointer_variable_name;`

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

Dereference operator

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

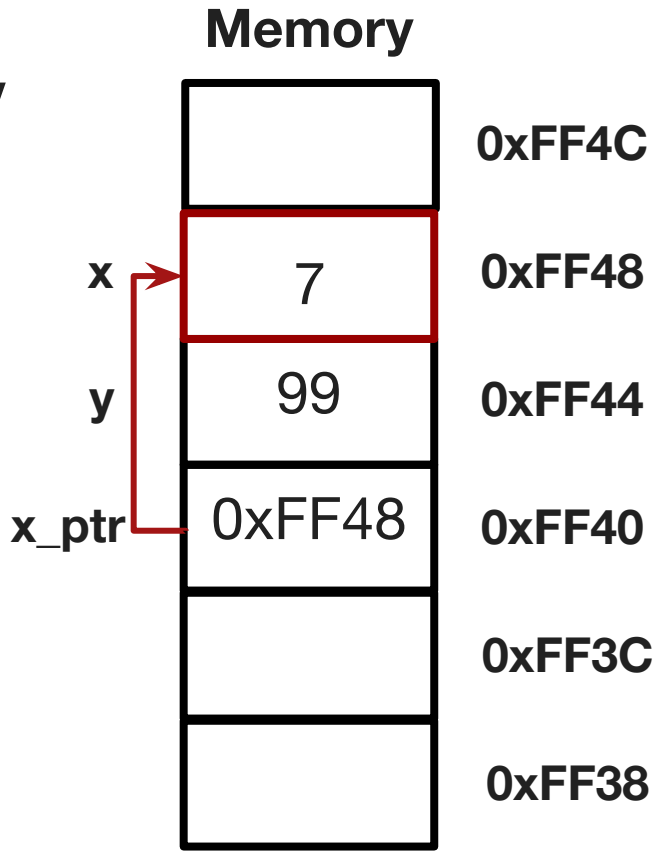
```
int x = 2;
int y = 99;
int *x_ptr = &x;
// *x_ptr will go to address
// 0xFF48 and get the value 2
printf("%d\n", *x_ptr); //prints 2
```



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;
```



recap_exercise.c What will this print out?

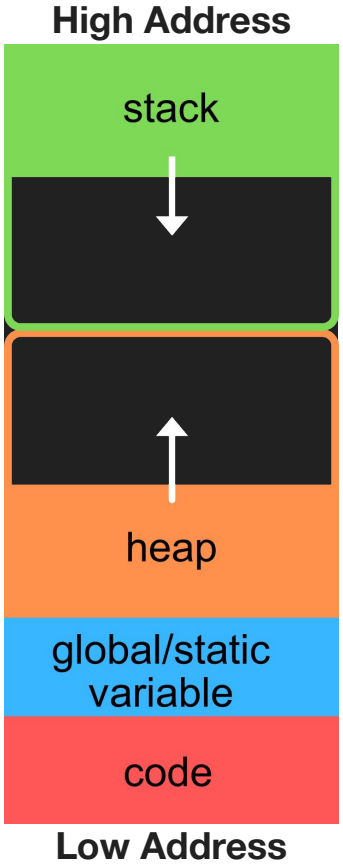
```c
int x = 2;
int y = 5;

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

int z = *ptr1 + *ptr2;
*ptr2 = z * 2;
printf("%d %d %d %d %d\n", x, y, z, *ptr1, *ptr2);

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

Pass By Value Recap: What will this print?

```c
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;
}
```

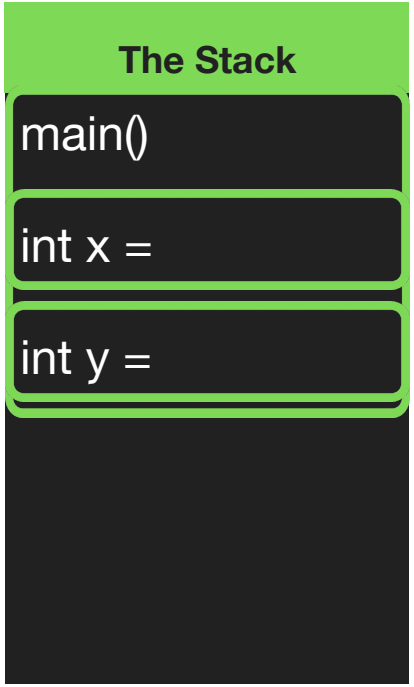
```c
void update(int x, int y) {
    x = x + 1;
    y = y - 1;
}

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

More about Memory: The Stack



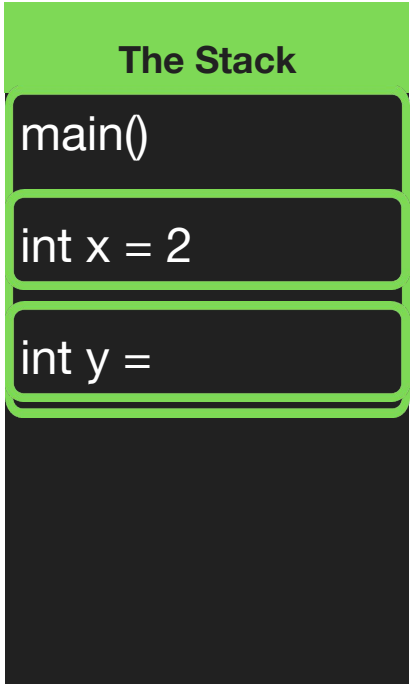
- Stack memory stores data about each function your program calls.
- When a function is called, data gets pushed onto the stack such as
  - local variables
  - where to return to when the function finishes
- Once your function finishes, its data including variables will **automatically** be removed from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

More about Memory: The Stack



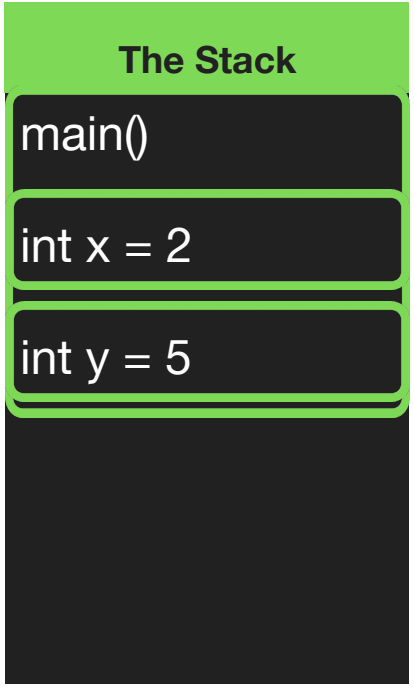

```c
void update(int x, int y) {
    x = x + 1;
    y = y - 1;
}

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



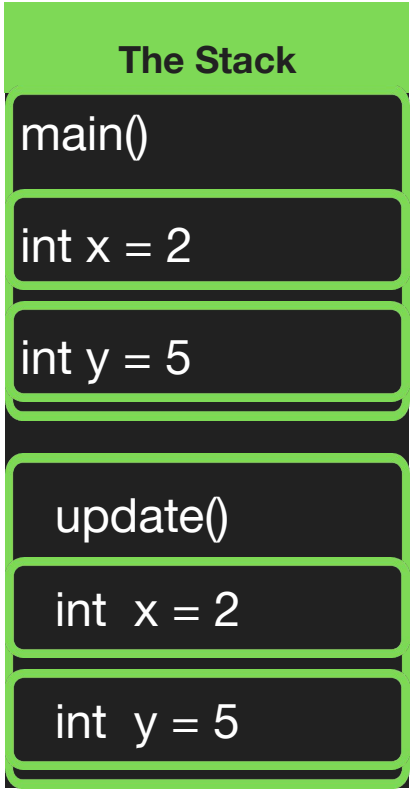

```c
void update(int x, int y) {
    x = x + 1;
    y = y - 1;
}

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



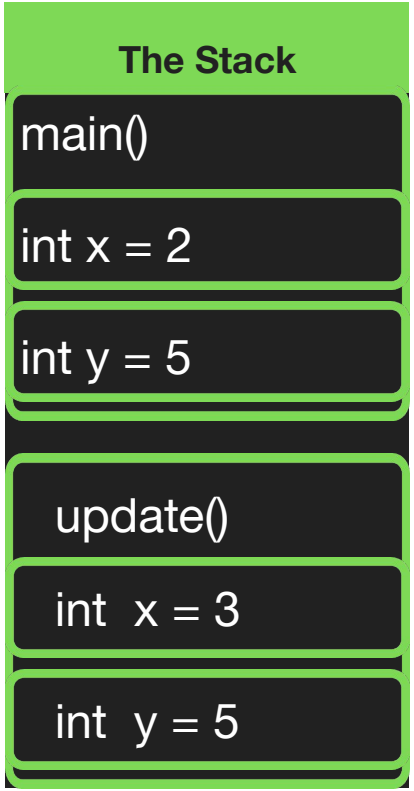

```c
void update(int x, int y) {
    x = x + 1;
    y = y - 1;
}

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



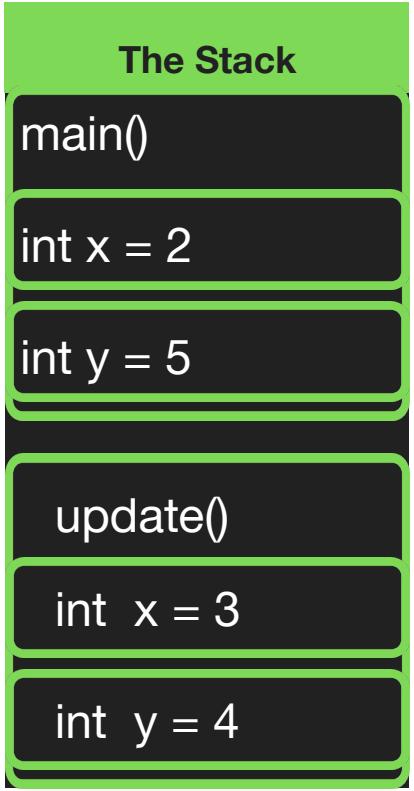

```c
void update(int x, int y) {
    x = x + 1;
    y = y - 1;
}
int main(void) {
    int x = 2;
    int y = 5;
    update(x,y);
    printf("%d %d\n", x, y);
    return 0;
}
```

More about Memory: The Stack



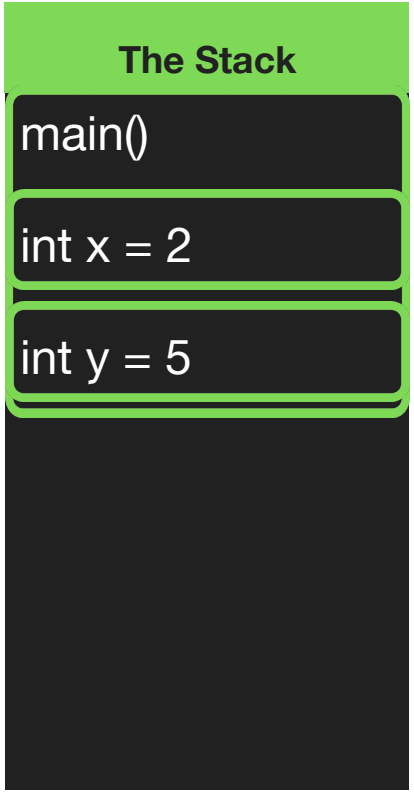

```c
void update(int x, int y) {
    x = x + 1;
    y = y - 1;
}
int main(void) {
    int x = 2;
    int y = 5;
    update(x,y);
    printf("%d %d\n", x, y);
    return 0;
}
```

More about Memory: The Stack




```c
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;
}
```

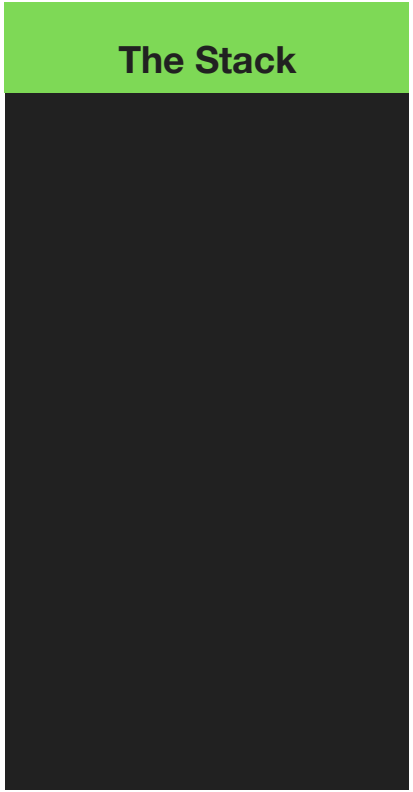
More about Memory: The Stack




```c
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




```c
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

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

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

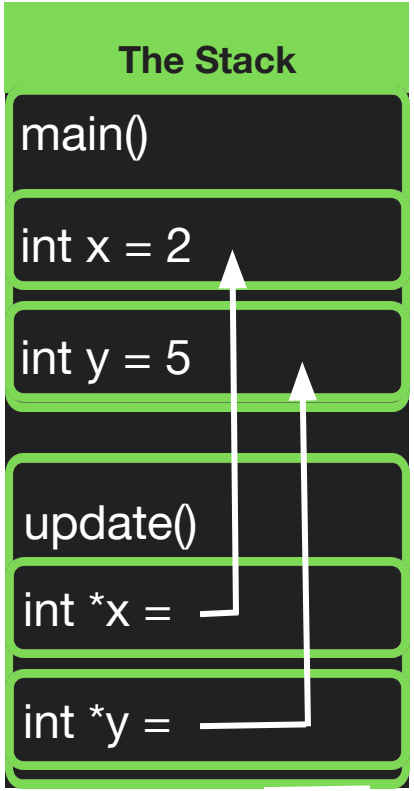
To do this:

- 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

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

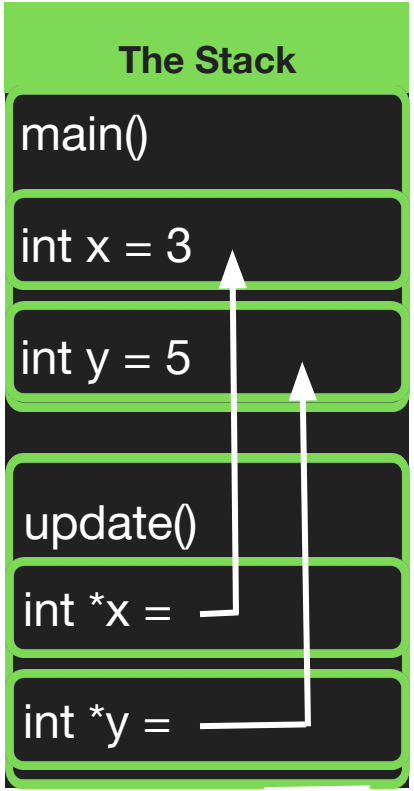
```c
void update(int *x, int *y){
    *x = *x + 1;
    *y = *y - 1;
}
```



Functions and Pointers

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

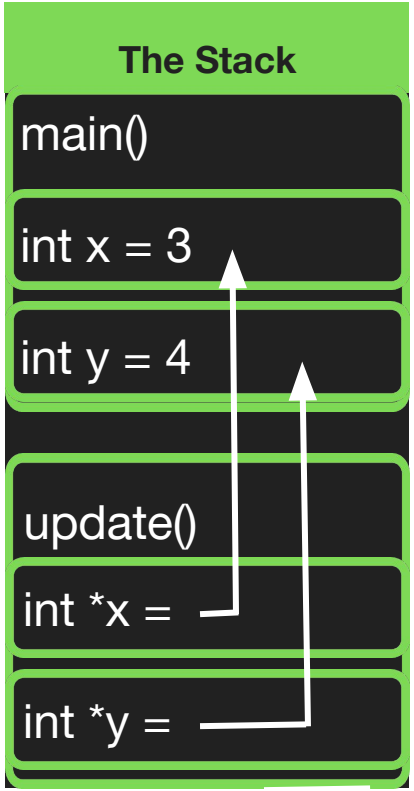
```c
void update(int *x, int *y){
    *x = *x + 1;
    *y = *y - 1;
}
```



Functions and Pointers

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

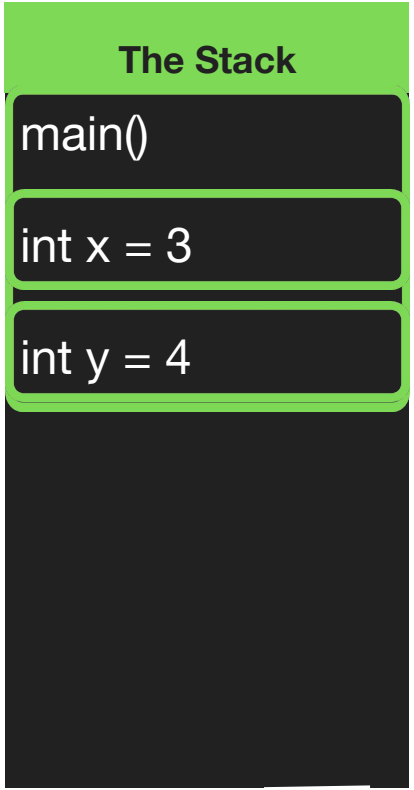
```c
void update(int *x, int *y){
    *x = *x + 1;
    *y = *y - 1;
}
```



Functions and Pointers

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

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



Now how can we modify swap?

```c
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;
}
```

```c
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;
}
```

Pointers to structs

```c
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

```c
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);
}
```

In this case we could return the updated copy!!

Pointers to structs

```c
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 - a pointer!
  - This is why we can modify the contents of our arrays arguments

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

Coding Demo Arrays and Pointers

array_arguments.c

arrays_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);
```

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?

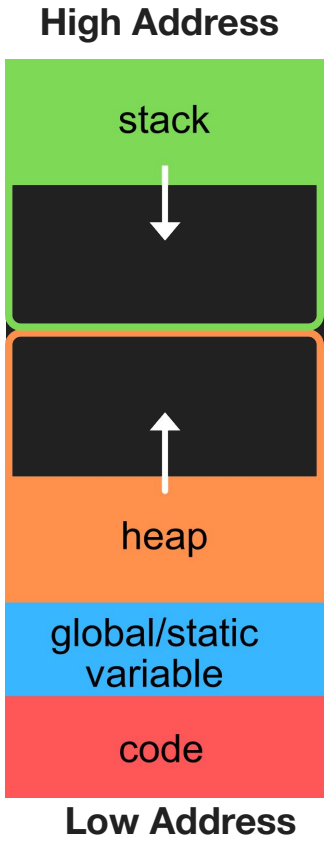
```
int *f(void){
    int numbers[] = {1, 2, 3};
    return numbers;
}
```

We can't return the address of a local variable

Local variables live on the stack

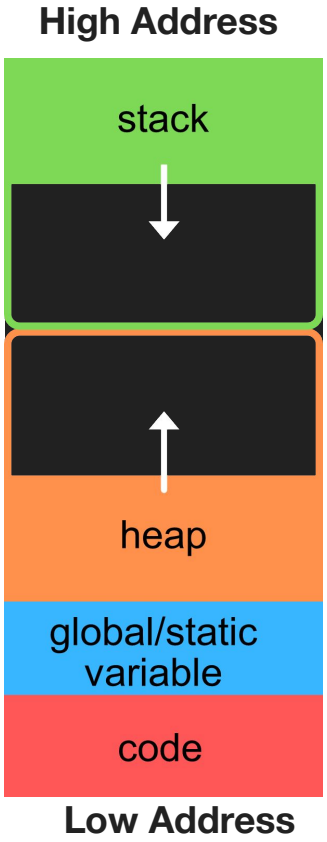
When the function returns it does not exist any more!

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

The Heap: malloc

- malloc is short for memory allocate
- malloc lets us ask for a number of bytes of memory on the heap
- malloc returns
  - a pointer to the chunk of memory or
  - NULL if there is not enough memory left to give us
  - You should always check for NULL in case.
- This allows us to dynamically create memory when we need it that will last beyond the end of functions and until we say we don't want it anymore.
- You need to #include <stdlib.h> to use malloc

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

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

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

sizeof Operator

- We should use the **`sizeof`** operator to help calculate the exact number of bytes we need to malloc
- sizeof returns an unsigned long value
  - long is a type like int that can stores larger numbers
  - unsigned means that it only stores value >= 0
- We can use it to find the sizeof a type or a variable.

```
// See sizeof_example.c for full program
int main(void) {
    printf("Size of an int: %lu bytes\n", sizeof(int));
    printf("Size of 10 ints: %lu bytes\n", 10 * sizeof(int));
```

Using malloc

- multiply the number of elements you need by the sizeof the type of the element to work out how many bytes you want malloc to give you
- malloc will return a pointer to the starting address of the chunk of memory it allocated

```
int *numbers = malloc(10 * sizeof(int));
```

Creating an array with malloc

```c
#include <stdlib.h>

int main(void) {
    int num_elements;
    scanf("%d", &num_elements);
    // Size of array is determined by user at run time
    int *data = malloc(num_elements *sizeof(int));
    // Check to see if malloc was successful
    if (data == NULL) {
        printf("Out of memory\n");
        return 1;
    }
```

Using array created by malloc

- You can use array indexes as usual on arrays created with malloc!
- You can pass them into functions whose inputs are arrays
- You can RETURN them from functions too!
  - Because they were malloced so are on the heap!!!!

Using array created by malloc

```c
// returns a malloced array that has been initialised
int *create_array(void) {
    int num_elements;
    scanf("%d", &num_elements);
    int *data = malloc(num_elements *sizeof(int));
    // error checking omitted for slide
    for (int i = 0; i < num_elements; i++) {
        data[i] = i;
    }
    return data;
}
```

The free function

- The free function lets the system know you don’t need chunk of memory any more
- Every malloc needs a corresponding free
  - If your program keeps calling malloc without corresponding free calls, the program will use more and more memory. This is called a memory leak.
  - This is a big issue for long running programs
  - Operating systems recover memory when the program ends
- Accessing memory after freeing or freeing it again causes nasty bugs

Putting it all together

```c
// create array
int *data = malloc(num_elements *sizeof(int));
// check malloc was successful

// Use the array somehow
// etc etc

// Free array when finished with array
free(data);
```

Note: You can check for memory leaks using dcc with the flag

`dcc --leak-check`

Coding with malloc

malloc_array.c

Demonstrate and check for memory leaks using dcc with the flag

```
dcc --leak-check
```

memory_hog.c

The realloc function

- What happens if this array wants to actually grow after you have filled it up?
- We can use realloc on a dynamically created array

```
// Ask malloc for enough memory for an array of 10 ints
int *numbers = malloc(10 * sizeof(int));

// Decide later we need enough for 20 ints
// It will make the array bigger, without destroying the
// contents
numbers = realloc(numbers, 20 * sizeof(int));
```

Realloc coding example

realloc_array.c

What are Multi-File Projects?

Multi-File Projects

- Big programs are often spread out over multiple files. There are a number of benefits to this:
  - Improves readability (reduces length of program)
  - You can separate code by subject (modularity)
  - Modules can be written and tested separately
- So far we have already been using the multi-file capability.
  - Every time we #include, we are actually borrowing code from other files
  - We have been only including C standard libraries

Multi-File Projects

- You can also #include your own! (FUN!)
- This allows us to join projects together
- It also allows multiple people to work together on projects out in the real world
- We will also often produce code that we can then use again in other projects
    - that is all that the C standard libraries are - functions that are useful in multiple instances)
- Assignment 2 will be a multi-file assignment.
    - Assignment 1 is not. Do NOT split it up into multiple files

Multi-File Projects

- In a multi file project we might have:
  - (multiple) header files - like the .h files that you have been using from standard libraries already
  - (multiple) implementation files - these are .c files, they implement what is in the corresponding header file.
- a .c file with a main function - this is the entry to our program, we try and have as little code here as possible

Header (.h) Files

- .h files typically contain:
  - function prototypes for the functions that will be implemented in the implementation (.c) file
  - comments that describe how the functions will be used
  - #defines and enums
  - they do not contain executable statements
- .h files give
  - the programmer all the information they need to use the code (a bit like documentation)
  - the compiler the information it needs to do type/syntax checking on the related .c files you **`#include`** it in

Implementation (.c) Files

- There will be exactly one .c file with a main function
- Other .c files typically contain:
  - Implementations of the functions that you have defined in the corresponding header files
- .c files **`#include`** relevant .h files
  - You use `""` instead of **<>** to include your own files E.g.
  - **`#include "array_utilities.h"`**

Example: Multi-File C Program

Suppose we have three files:

- header file **`array_utilities.h`**
- implementation file **`array_utilities.c`**
  - **`#include "array_utilities.h"`**
- file with main function **`program.c`**
  - **`#include "array_utilities.h"`**

Compiling Multi-File Programs

- You do **not** compile the **.h** files.
  - They should already be included in the relevant .c files
- You compile .c files together into 1 executable
  - Exactly one of the .c files should have a main function
- E.g.

```
$ dcc -o program program.c utilities.c
$ ./program
$ dcc -o other_program other_program.c utilities.c
$ ./other_program
```

What did we learn today?

- Pointers
  - Recap
  - Pointers and Arrays (array_arguments.c, array_addresses.c string_pointer_exercise.c)
- Dynamic Memory Allocation
  - the heap
  - malloc/free (sizeof_example.c, malloc_array.c, memory_hog.c)
  - realloc (realloc_array.c)
- Working with multi-file projects

Now that we know

- structs
- pointers
- malloc/free

Next lecture we are ready to learn …

Linked Lists!

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/u89nL992CA

Reach Out

Content Related Questions:

Forum

Admin related Questions email:

cs1511@unsw.edu.au