COMP1511 - Programming Fundamentals

Week 4 - Lecture 8

What did we learn today?

Memory and Pointers

- Pointers are variables that contain memory addresses
- We can use them to get access to variables anywhere in our program
- Functions operate in their own memory "space"

Using Functions

- A practical example of how functions can separate code
- Makes our code very readable
- Also means that all of the code for a specific purpose is collected together

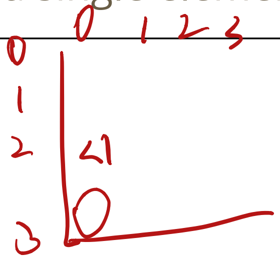
Accessing 2D Arrays

Two coordinates to access single elements

- We use two dimensions to create the 2D array
- We also use two coordinates to get access to a single element

```
int main (void) {
    // declare a 2D Array
    int grid[4][4] = {0};

    // test a value
    if (grid[2][0] < 1) {
        // print out a value
        printf(“The bottom left square is: %d”, grid[3][0]);
    }
```



Houses and addresses

Continuing the idea . . .

- A variable is a house
- That house is in a certain location in memory, its address
- The house contains the bits and bytes that decide what the value of the variable is

The address is an integer

- In a 64 bit system, we'll usually use a 64 bit integer to store an address
- We can address 2^{64} bytes of memory

指针

Introducing Pointers

A New Variable Type - Pointers

- Pointers are variables that hold memory addresses
- They are created to point at the location of variables

- If a variable was a house, the pointer would be the address of that house
- In C, the pointer is like an integer that stores a memory address
- Pointers are usually created with the intention of "aiming at" a variable (storing a particular variable's address)

Pointers in C

声明

Pointers can be declared, but slightly differently to other variables

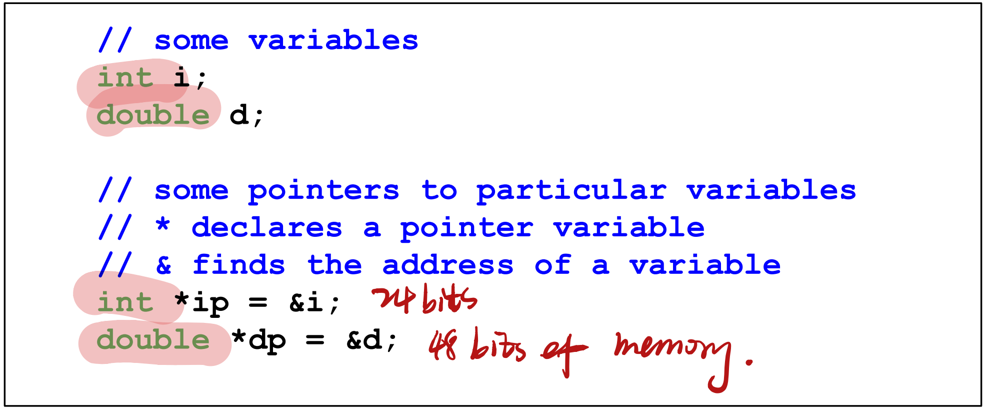
- A pointer is always aimed at a particular variable type
- We use a `*` to declare a variable as a pointer
- A pointer is most often "aimed" at a particular variable
- That means the pointer stores the address of that variable
- We use `&` to find the address of a variable

```
int i = 100;
// create a pointer called ip that points at
// an integer in the location of i
int *ip = &i;
```

a variable

Pointer Types

Different pointers to point at different variables



```
// some variables
int i;
double d;

// some pointers to particular variables
// * declares a pointer variable
// & finds the address of a variable
int *ip = &i;  24 bits
double *dp = &d;  48 bits of memory.
```

初始化

Initialising Pointers

Pointers should be initialised like other variables

- Generally pointers will be initialised by pointing at a variable
- "`NULL`" is a `#define` from most standard C libraries (including stdio.h)
- If we need to initialise a pointer that is not aimed at anything, we will use `NULL`

Using Pointers

If we want to look at the variable that a pointer "points at"

- We use the * on a pointer to access (dereference) the variable it points at
- Using the address analogy, this is like following the address to actually get to the house, then looking inside

```
int i = 100;
// create a pointer called ip that points at
// the location of i
int *ip = &i;
printf("The value of the variable at %p is %d", ip, *ip);
```

address

100 (value) of i

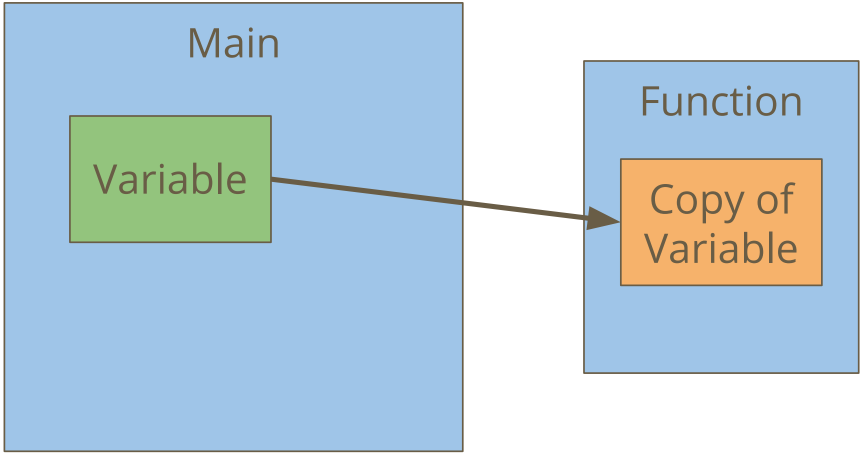
- `%p` in `printf` will print the address stored in a pointer

Pointers and Functions

Pointers allow us to pass around an address instead of a variable

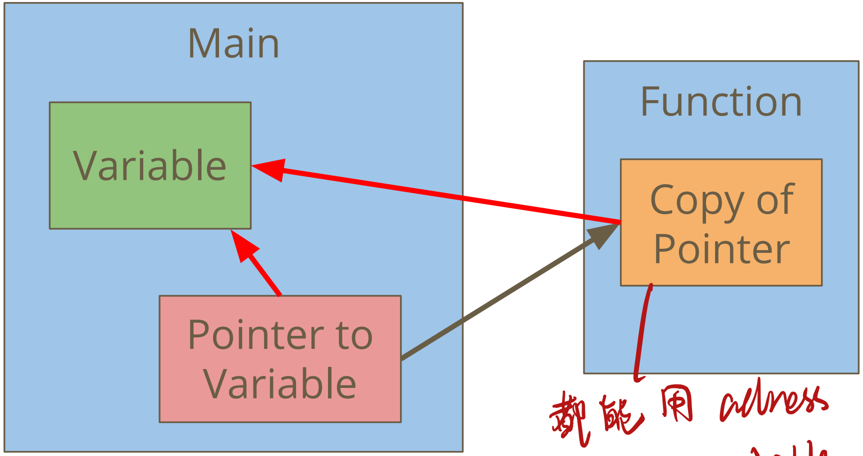
- We can create functions that take pointers as input
- All function inputs are always passed in "by value" which means they're copies, not the same variable
- But if I have a copy of the address of a variable, I can still find exactly the variable I'm looking for

Function variables pass in "by value"



In this case, the copy of the variable can't ever change the value of the variable, because it's just a copy

Pointers pass in "by value" also



都能用 address
找到 variable

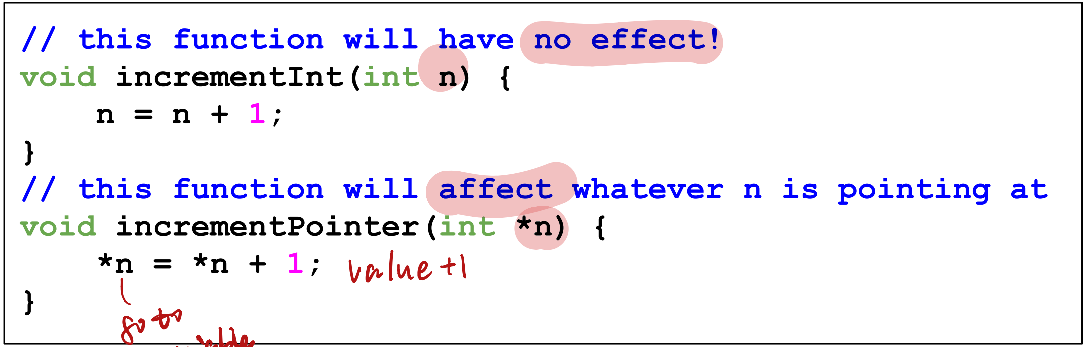
The function has a copy of the pointer.

However, even a copy of a pointer contains the address of the original variable, allowing the function to access it.

Pointers and Functions in code

The following code illustrates the two examples

- A variable passed to a function is a copy and has no effect on the original
- A pointer passed to a function gives us the address of the original



```
// this function will have no effect!
void incrementInt(int n) {
    n = n + 1;
}
// this function will affect whatever n is pointing at
void incrementPointer(int *n) {
    *n = *n + 1;
}
```

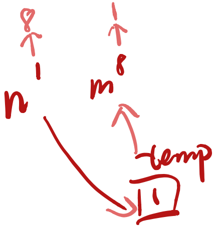
value +1

go to variable

Pointers and Functions

We can now do more with functions

- Pointers mean we can give multiple variables to a function
- This means one function can now change multiple variables at once



```
// This function is now possible!
void swap(int *n, int *m) {
     int tmp;
     tmp = *n; 1
   1 *n = *m; 8.
   8 *m = tmp;
}
```

交换

Pointers and Arrays

Arrays are blocks of memory 内存块

- An array variable is actually the memory address of the start of the array!
- This is why arrays as input to functions let you change the array

```
int numbers[10];
// both of these print statements
// will print the same address!
printf("%p\n", &numbers[0]);
printf("%p\n", numbers);
```

address

=

address for first element of array (start array)

whole array

Let's make a program using functions and pointers

This program is called The Jumbler打乱

- It will take some numbers as inputs
- It will jumble them a little, changing their order
- Then it will print them back out

- We'll make some use of functions to separate our code
- We'll show how pointers let us access memory in our program

What functions do we want?

Deciding how to split up our functionality

- A function that reads the inputs as integers
- A function that swaps two numbers
- A function that swaps several numbers
- A function that prints out our numbers

Reading Input

A function to read inputs into an array

- We're also going to want to know how many numbers are being entered!

contant # define 100 (足够大)

```
int read_inputs(int nums[MAX_NUMS]) {
     int i = 0;
     int inputCount = 0;
     printf("How many numbers? ");
     scanf("%d", &inputCount);
     while (i < MAX_NUMS && i < inputCount) { // have processed i inputs
          scanf("%d", &nums[i]);
          i++;
     }
     return inputCount
}
```

NT element

numbers in array

Printing our numbers

This is a trivial function

- The only issue is that we might have to work with an array that isn't full
- So we use numCount to stop us early if necessary

print array 上面 (inputCount)

```
void print_nums(int nums[MAX_NUMS], int numCount) {
    int i = 0;
    while (i < MAX_NUMS && i < numCount) {
        printf("%d ", nums[i]);
        i++;
    }
}
```

don't need return

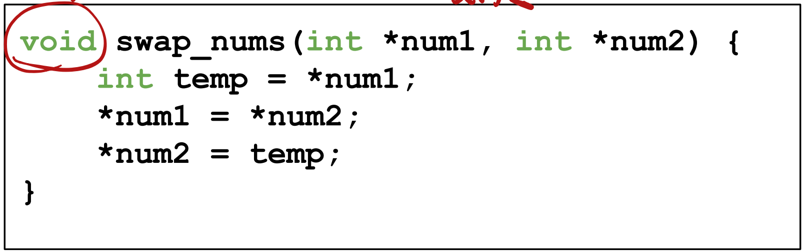
no information back to main function.

Using Pointers to swap variable values

A simple swap function

- This function doesn't even know whether the ints are in arrays or not
- It sees two memory locations containing ints
- and uses a temporary int variable to swap them

临时 交换



```
void swap_nums(int *num1, int *num2) {
        int temp = *num1;
        *num1 = *num2;
        *num2 = temp;
}
```

Jumble performs some swaps

This function just loops through and swaps a few numbers

候选

- This is a good candidate for a function that could be changed or written differently and just used by our main without thinking about it

Array

input count

```
void jumble(int nums[MAX_NUMS], int numCount) {
    int i = 0;
    while (i < MAX_NUMS && i < numCount) {
        int j = i * 2;
        if (j < MAX_NUMS && j < numCount) {
            swap_nums(&nums[i], &nums[j]);
        }
        i++;
    }
}
```

=2i

i j
1 <-> 2
2 <-> 4
3 — 6
4 — 8.
5 —

打乱

Using all the functions in the main

A nice main makes use of its functions

- It's very easy to read this main!
- It shows its steps using its function names
- There isn't much code to dig through

```
int main(int argc, char *argv[]) {
    int numbers[MAX_NUMS]; = {0}
    int numInputs = read_args(numbers); = function results
    jumble(numbers, numInputs);
    print_nums(numbers, numInputs);
    return 0;
}
```