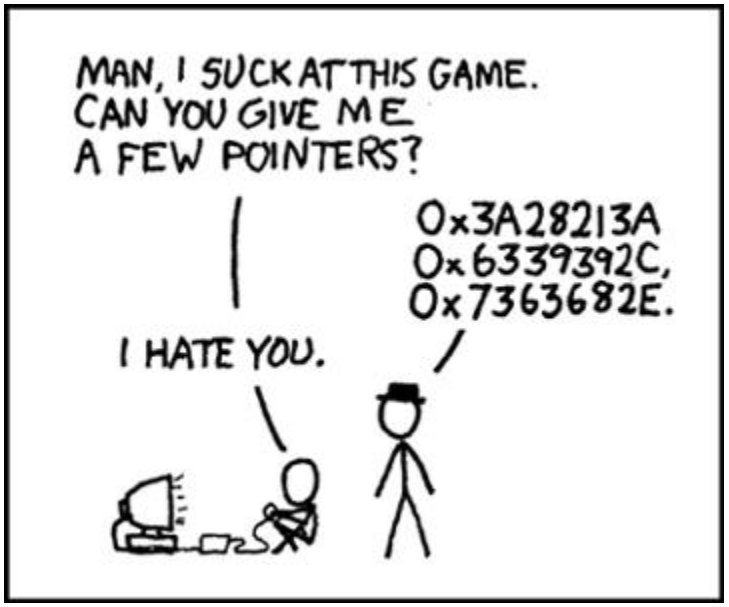
P2T 2019
C Lecture 5

Dr Gordon Stewart
Room 427, Kelvin Building
gordon.stewart@glasgow.ac.uk / x6439

Source material: Prof David Britton / Dr Sam Skipsey

Pointers



https://xkcd.com/138/

Functions

- Functions receive a **copy** of the values in their parameters
- Functions can only modify this copy, not the original values

Functions

- Functions receive a **copy** of the values in their parameters
- Functions can only modify this copy, not the original values

```
void timesTwo(int i) {
   i = i * 2;
}

int main(void) {
   int number = 6;
   timesTwo(number);
   printf("%d\n", number);

   return 0;
}
```

Functions

- Functions receive a **copy** of the values in their parameters
- Functions can only modify this copy, not the original values

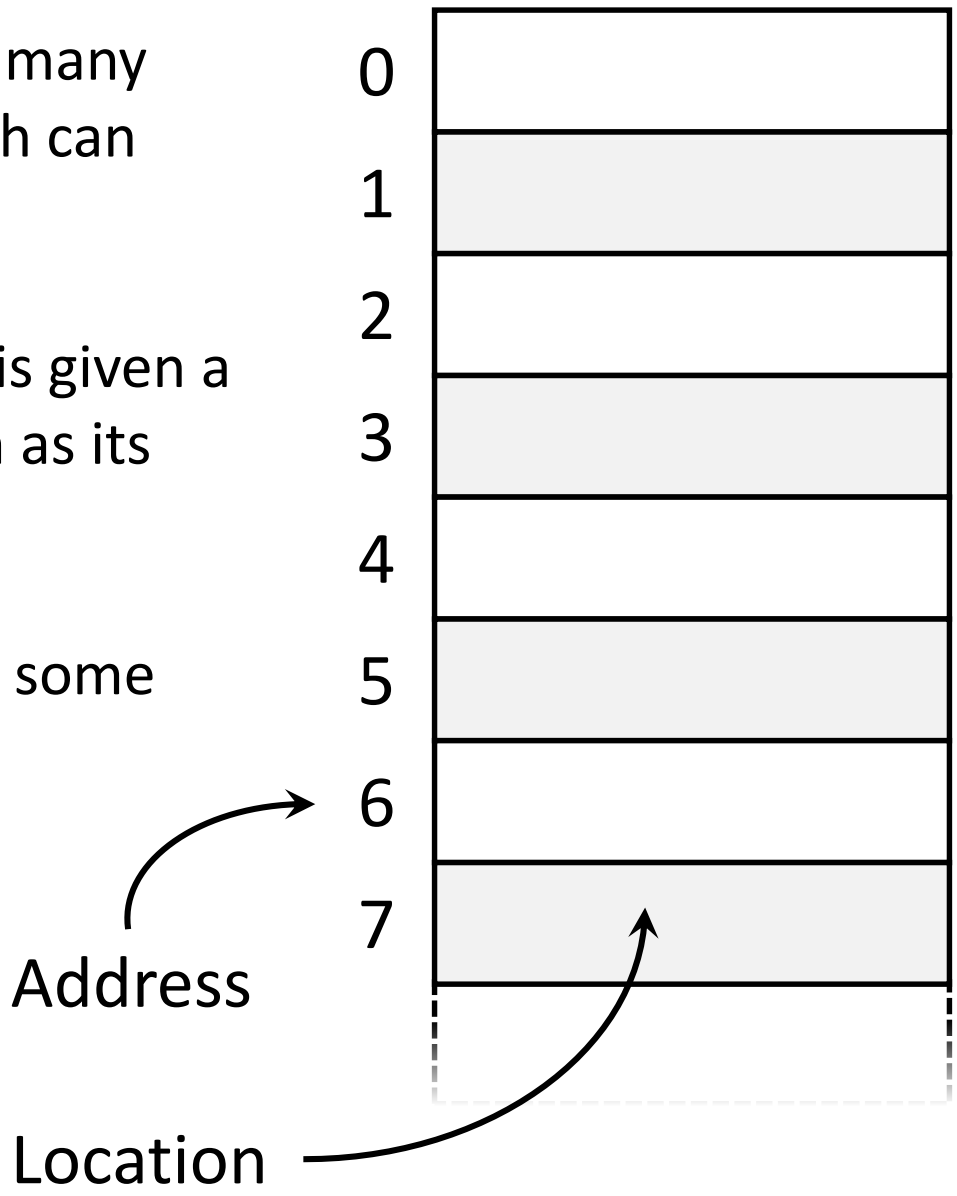
```
void timesTwo(int i) {
   i = i * 2;
}

int main(void) {
   int number = 6;
   timesTwo(number);
   printf("%d\n", number); // Prints 6, not 12!

   return 0;
}
```

Functions

- Functions receive a **copy** of the values in their parameters
- Functions can only modify this copy, not the original values
- To modify the original value of a variable rather than a copy, we need some way to tell the function about the variable itself
- We do this by telling the function “where in memory the variable is stored”
- Need a type which “points” to a variable’s storage location: a **pointer**

Memory

- Memory is made up of many **locations**, each of which can store a value
- Each memory location is given a numerical index known as its **address**
- Addresses go from 0 to some large number



Pointer types

- A **pointer** is a variable which stores the **address** of a location in memory

- A pointer has a type which tells the compiler how to interpret the value at that address

- It is assumed that the memory which is pointed to does actually contain a value of that type!

Declaring pointers

- Similar to arrays, we declare a pointer to a type by "decorating" a name with a symbol

- For arrays, we added **`[]`** to a name to make it an array of values of that type:

```
float temperatures[60];
```

- For pointers, we add **`*`** to the start of the name

- Here, **`x`** and **`y`** are variables of type **`int`**, and **`p`** is a variable of type **pointer to `int`**:

```
int x, y, *p;
```

Null

- We can give a pointer an initial value, but unlike other types, it is very hard to provide a literal value that will be useful

- We shouldn't point a pointer at an area of memory that doesn't contain a value of the correct type!

- The most useful literal pointer value is the special value **NULL**
  - Essentially, this means "do not point at anything"

```
int x, y, *p;

/* This pointer is explicitly pointing at
   nothing */
int *p1 = NULL;
```

Pointing at a variable

- We almost always want our pointers to point at the memory used by an existing variable

- The special operator **`&`** provides the address in memory where a variable is located

```
// Declare integer
int n_value = 12;

// Declare pointer
int *n_address = &n_value;
```

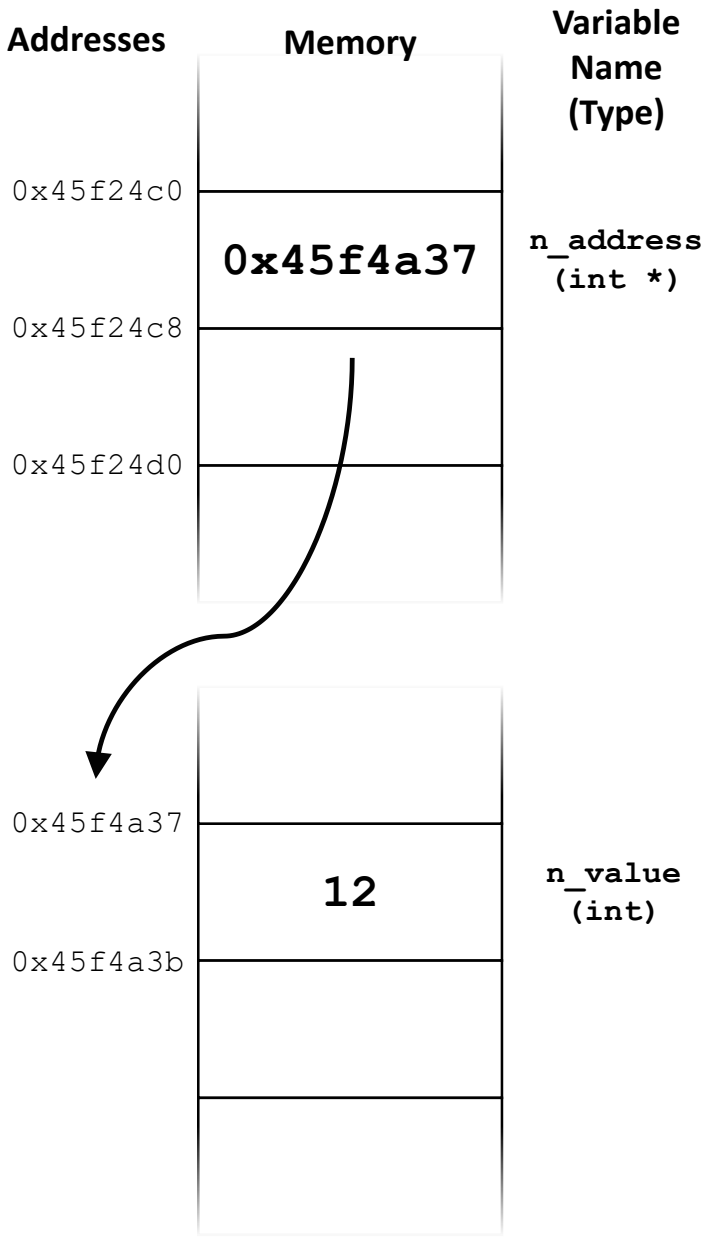
- Already seen this with **`sscanf()`**

Pointing at a variable

- We almost always want our pointers to point at the memory used by an existing variable
- The special operator **&** provides the address in memory where a variable is located

```
// Declare integer
int n_value = 12;

// Declare pointer
int *n_address = &n_value;
```

- Already seen this with **sscanf()**



Getting a value from a pointer

- The special operator **&** provides the address in memory where a variable is located

- The special operator ***** does the opposite: it provides the value located at the memory address indicated by the pointer (sometimes called **dereferencing** a pointer)

```
int n_value;
int *n_address = &n_value;

// Update value using pointer
*n_address = 15;

// Display value using pointer
printf("%d", *n_address);
```

Passing pointers to functions

Passing pointers to functions

- Function **addTwo** takes a **pointer to an int** as its argument
- It adds **2** to the value in the location pointed to by the pointer
- We call **addTwo** with the address of **num**, so **addTwo** modifies **num**'s contents
- The value printed is 5 (not 3)

```
// Increase value by two
void addTwo(int *ptrInt) {
    *ptrInt += 2;
}


int main(void) {
    int num = 3;

    // Call addTwo with address
    // of num
    addTwo(&num);

    // Prints 5
    printf("%d\n", num);

    return 0;
}
```

Dangers of pointers

- The value of a pointer is a memory address, which is just a number (so we can do arithmetic with pointers, if we want to)

- Have to use brackets carefully, otherwise we might change the value of the pointer itself (the address), rather than the thing it is pointing at

```
*p += 1;
(*p)++;
*p = (*p) + 1;
```

These all add one to the variable being pointed at by **p**

```
*p++;
```

This adds one to **p** (i.e. it changes the memory address it points to), as **++** has higher precedence than *****

Dangers of pointers

- Pointers can be very useful, but they need care
- A pointer simply contains the address of a memory location (with a type)
- C provides no guarantees regarding the *actual* status or use of this location
- It's up to us to ensure that pointers are actually pointing at what we think they're pointing at

Dangers of pointers

```
int main(void) {
    int *p = NULL;

    /* Some loop */
    for (int n = 0; n < 8; n++) {
        p = &n;
        printf("%d\n", *p);
    }

    // MORE CODE HERE

    /* n is now out of scope:
       the memory p points at
       might have changed! */

    *p += 1; // DANGEROUS!

    return 0;
}
```

Dangers of pointers

- The variable **`n`** has **block scope** within the **`for`** loop
- Within the loop, **`p`** is assigned the address of **`n`**
- Within the loop, **`n`** is always in scope, so the value at the memory address which **`p`** points to is always that of **`n`**

```
int main(void) {
    int *p = NULL;

    /* Some loop */
    for (int n = 0; n < 8; n++) {
        p = &n;
        printf("%d\n", *p);
    }

    // MORE CODE HERE

    /* n is now out of scope:
       the memory p points at
       might have changed! */

    *p += 1; // DANGEROUS!

    return 0;
}
```

Dangers of pointers

- At the end of the loop, **`n`** goes out of scope – the memory allocated to it can be reused if needed
- The value pointed to by **`p`** could now be anything
- Even worse, as we're adding one to it, we're altering the value in a memory location that could be used for anything!

```
int main(void) {
    int *p = NULL;

    /* Some loop */
    for (int n = 0; n < 8; n++) {
        p = &n;
        printf("%d\n", *p);
    }

    // MORE CODE HERE

    /* n is now out of scope:
       the memory p points at
       might have changed! */

    *p += 1; // DANGEROUS!

    return 0;
}
```

Pointers and arrays

- The bare name of an array can be treated as a pointer
- C performs some behind-the-scenes work to allow you to mix pointer and array semantics:

```
int myArray[5] = {0, 3, 12, 15, 36};

int *p = myArray; // p points to the start of myArray

p[3] = 123; // myArray[3] is now 123
```

- This is useful if you want to pass an array to a function

C Lab 4 Exercise 1

Arrays and functions

- When passing an array to a function, C actually passes a pointer

- The function does not know the size of the array, so this must be passed separately:

```
int doStuff(int a[], int size) {
    // OR:  int doStuff(int *a, int size)

    // SOME CODE HERE...
}

int main(void) {
    int arr[] = {3, 12, 15};

    // Call doStuff, and calculate size of array
    doStuff(arr, sizeof(arr) / sizeof(arr[0]));
}
```

Example: arrays and functions

```
#include <stdio.h>

/*
 * For full marks, I need to include some meaningful comments...
 *
 */

// Calculate the sum of the elements in array a
int sum(int a[], int size) {
    int total = 0;
    // Remember, array elements are numbered from zero
    for (int n = 0; n < size; n++) {
        total += a[n];
    }
    return total;
}

int main(void) {
    int arr[3] = {3, 12, 1};
    int total = sum(arr, sizeof(arr) / sizeof(arr[0]));
    // Here we could "cheat" and say sum(arr, 3)

    printf("The sum is %d\n", total);

    return 0;
}
```

Multidimensional arrays

```
#include <stdio.h>

/* Simple program to demonstrate 2D array parameter passing */


// Multiply each element in array by two
void timesTwoA(int size_y, int size_x, int arr[size_y][size_x]) {
    for (int y = 0; y < size_y; y++) {
        for (int x = 0; x < size_x; x++ ) {
            arr[y][x] *= 2;
        }
    }
}

int main(void) {
    int my2DArray[3][2] = {{12, 15}, {36, 3}, {123, 456}};

    timesTwoA(3, 2, my2DArray);

    return 0;
}
```

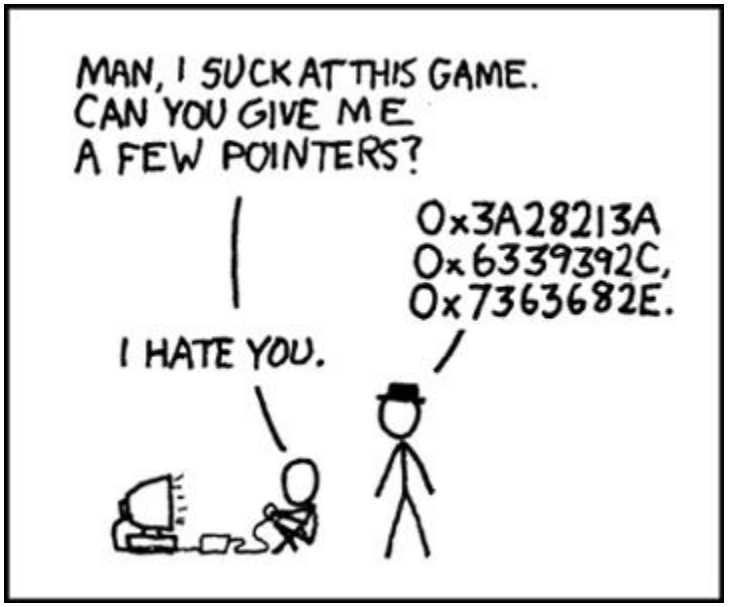
C Lab 4 Exercise 3

Pointers to structs

- We can also create pointers to structs
- Care is needed when accessing components of the value

```
struct particle {
    double x, y, z;
    double vx, vy, vz;
    char ptype;
};

int main(void) {
    struct particle *muon;

    // Dereference pointer, then select component...
    (*muon).x = 3.0;

    // ...OR dereference and select component in one go
    muon->x = 3.0;
}
```

Pointers



https://xkcd.com/138/

Command Line Arguments

Command Line Arguments

- Programs can be passed arguments on the command line:

```
tail –n 5 readme.txt
```

arguments

- The **`main`** function is special and can be declared in two different ways:

```
int main(void)

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

- The second form lets us get at the arguments passed to the program on the command line

Command Line Arguments

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

- **`argc`** is the **argument count**, i.e. the number of arguments including the name of the program itself
- **`argv`** is an array of **strings** which contains, in order, the text of each argument:

```
tail -n 5 readme.txt
```

argc	4
argv[0]	"tail"
argv[1]	"-n"
argv[2]	"5"
argv[3]	"readme.txt"

Command Line Arguments

```
#include <stdio.h>

int main(int argc, char * argv[]) {
    // Check number of arguments supplied
    if (argc != 4) {
            printf("Usage: args INT STRING FLOAT\n");
            return 1; // Exit with an error code
    }

    // Read arguments (remember argv[0] is the program name)
    int number1;  // First argument: int
    sscanf(argv[1], "%d", &number1);
    char * str = argv[2];  // Second argument: string
    float number2;  // Third argument: float
    sscanf(argv[3], "%f", &number2);

    printf("%d\n%s\n%f\n", number1, str, number2);

    return 0;
}
```

C Lab 4 Exercise 2