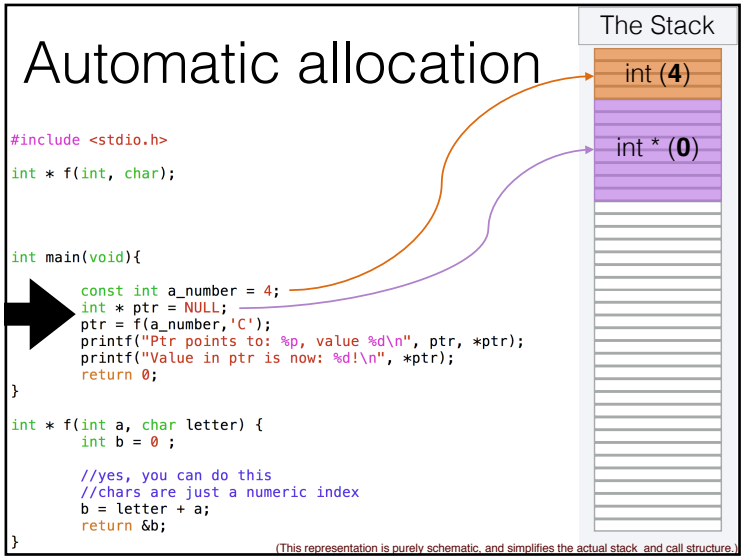
# C Lecture 5

More pointers, Command Line Arguments,
C Build Process (1)

# Beware Pointers!

- Last lecture, we saw how pointers could be used to allow you to access memory directly.
- This is useful for allowing functions to directly modify the contents of variables whose names are out of scope…
- …but it's also dangerous, as we have to be sure that the memory we're pointing at is still used for that variable.
- This requires a bit of understanding of how memory for variables is allocated, by default.

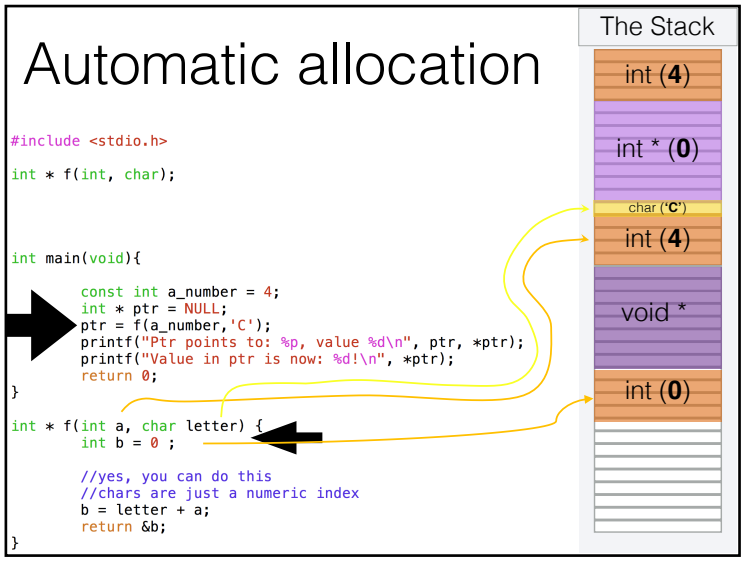
# Automatic allocation



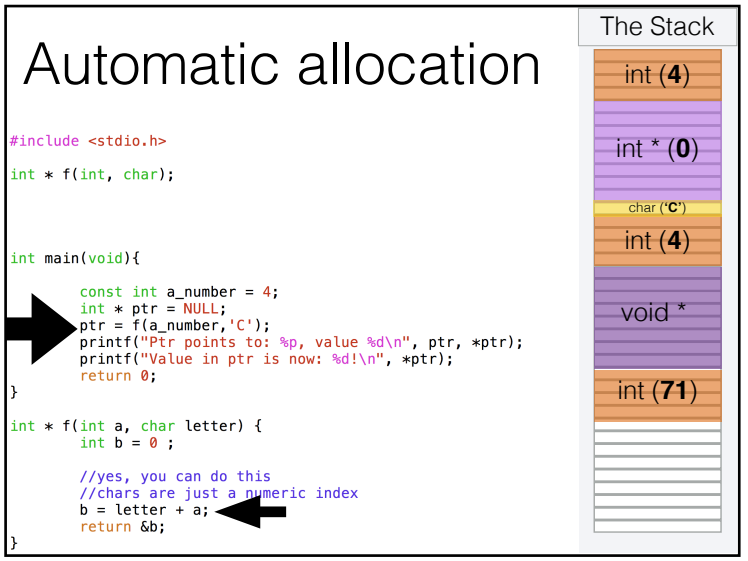

ASCII: American Standard Code for Information Interchange.

| Dec | Hx | Oct | Char | Dec | Hx | Oct | Html | Chr | Dec | Hx | Oct | Html | Chr | Dec | Hx | Oct | Html | Chr |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 000 | NUL (null) | 32 | 20 | 040 |   | Space | 64 | 40 | 100 | @ | @ | 96 | 60 | 140 | ` | ` |
| 1 | 1 | 001 | SOH (start of heading) | 33 | 21 | 041 | ! | ! | 65 | 41 | 101 | A | A | 97 | 61 | 141 | a | a |
| 2 | 2 | 002 | STX (start of text) | 34 | 22 | 042 | " | " | 66 | 42 | 102 | B | B | 98 | 62 | 142 | b | b |
| 3 | 3 | 003 | ETX (end of text) | 35 | 23 | 043 | # | # | 67 | 43 | 103 | C | C | 99 | 63 | 143 | c | c |
| 4 | 4 | 004 | EOT (end of transmission) | 36 | 24 | 044 | $ | $ | 68 | 44 | 104 | D | D | 100 | 64 | 144 | d | d |
| 5 | 5 | 005 | ENQ (enquiry) | 37 | 25 | 045 | % | % | 69 | 45 | 105 | E | E | 101 | 65 | 145 | e | e |
| 6 | 6 | 006 | ACK (acknowledge) | 38 | 26 | 046 | & | & | 70 | 46 | 106 | F | F | 102 | 66 | 146 | f | f |
| 7 | 7 | 007 | BEL (bell) | 39 | 27 | 047 | ' | ' | 71 | 47 | 107 | G | G | 103 | 67 | 147 | g | g |
| 8 | 8 | 010 | BS (backspace) | 40 | 28 | 050 | ( | ( | 72 | 48 | 110 | H | H | 104 | 68 | 150 | h | h |
| 9 | 9 | 011 | TAB (horizontal tab) | 41 | 29 | 051 | ) | ) | 73 | 49 | 111 | I | I | 105 | 69 | 151 | i | i |
| 10 | A | 012 | LF (NL line feed, new line) | 42 | 2A | 052 | * | * | 74 | 4A | 112 | J | J | 106 | 6A | 152 | j | j |
| 11 | B | 013 | VT (vertical tab) | 43 | 2B | 053 | + | + | 75 | 4B | 113 | K | K | 107 | 6B | 153 | k | k |
| 12 | C | 014 | FF (NP form feed, new page) | 44 | 2C | 054 | , | , | 76 | 4C | 114 | L | L | 108 | 6C | 154 | l | l |
| 13 | D | 015 | CR (carriage return) | 45 | 2D | 055 | - | - | 77 | 4D | 115 | M | M | 109 | 6D | 155 | m | m |
| 14 | E | 016 | SO (shift out) | 46 | 2E | 056 | . | . | 78 | 4E | 116 | N | N | 110 | 6E | 156 | n | n |
| 15 | F | 017 | SI (shift in) | 47 | 2F | 057 | / | / | 79 | 4F | 117 | O | O | 111 | 6F | 157 | o | o |
| 16 | 10 | 020 | DLE (data link escape) | 48 | 30 | 060 | 0 | 0 | 80 | 50 | 120 | P | P | 112 | 70 | 160 | p | p |
| 17 | 11 | 021 | DC1 (device control 1) | 49 | 31 | 061 | 1 | 1 | 81 | 51 | 121 | Q | Q | 113 | 71 | 161 | q | q |
| 18 | 12 | 022 | DC2 (device control 2) | 50 | 32 | 062 | 2 | 2 | 82 | 52 | 122 | R | R | 114 | 72 | 162 | r | r |
| 19 | 13 | 023 | DC3 (device control 3) | 51 | 33 | 063 | 3 | 3 | 83 | 53 | 123 | S | S | 115 | 73 | 163 | s | s |
| 20 | 14 | 024 | DC4 (device control 4) | 52 | 34 | 064 | 4 | 4 | 84 | 54 | 124 | T | T | 116 | 74 | 164 | t | t |
| 21 | 15 | 025 | NAK (negative acknowledge) | 53 | 35 | 065 | 5 | 5 | 85 | 55 | 125 | U | U | 117 | 75 | 165 | u | u |
| 22 | 16 | 026 | SYN (synchronous idle) | 54 | 36 | 066 | 6 | 6 | 86 | 56 | 126 | V | V | 118 | 76 | 166 | v | v |
| 23 | 17 | 027 | ETB (end of trans. block) | 55 | 37 | 067 | 7 | 7 | 87 | 57 | 127 | W | W | 119 | 77 | 167 | w | w |
| 24 | 18 | 030 | CAN (cancel) | 56 | 38 | 070 | 8 | 8 | 88 | 58 | 130 | X | X | 120 | 78 | 170 | x | x |
| 25 | 19 | 031 | EM (end of medium) | 57 | 39 | 071 | 9 | 9 | 89 | 59 | 131 | Y | Y | 121 | 79 | 171 | y | y |
| 26 | 1A | 032 | SUB (substitute) | 58 | 3A | 072 | : | : | 90 | 5A | 132 | Z | Z | 122 | 7A | 172 | z | z |
| 27 | 1B | 033 | ESC (escape) | 59 | 3B | 073 | ; | ; | 91 | 5B | 133 | [ | [ | 123 | 7B | 173 | { | { |
| 28 | 1C | 034 | FS (file separator) | 60 | 3C | 074 | < | < | 92 | 5C | 134 | \ | \ | 124 | 7C | 174 | | | \| |
| 29 | 1D | 035 | GS (group separator) | 61 | 3D | 075 | = | = | 93 | 5D | 135 | ] | ] | 125 | 7D | 175 | } | } |
| 30 | 1E | 036 | RS (record separator) | 62 | 3E | 076 | > | > | 94 | 5E | 136 | ^ | ^ | 126 | 7E | 176 | ~ | ~ |
| 31 | 1F | 037 | US (unit separator) | 63 | 3F | 077 | ? | ? | 95 | 5F | 137 | _ | _ | 127 | 7F | 177 | &#127; | DEL |

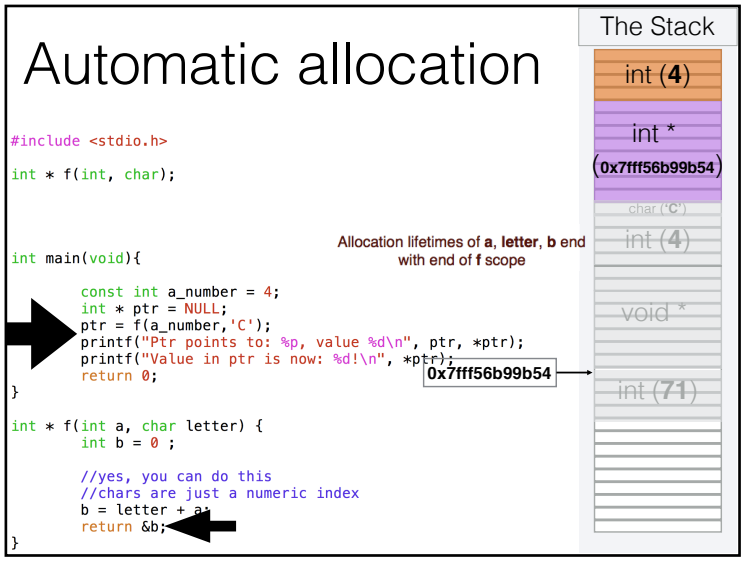
# Automatic allocation



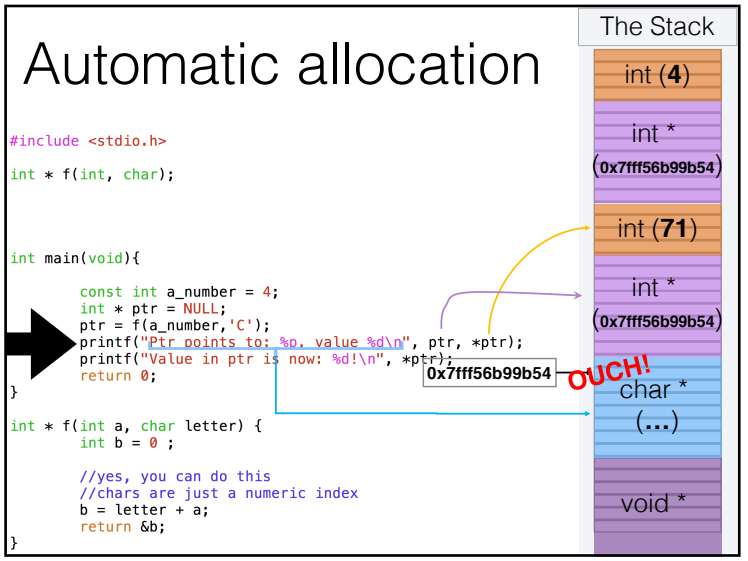

# Automatic allocation




# Automatic allocation




# Automatic allocation



# Pointers to structs

- We can create pointers to structured data types, just as we can to basic types.
- Care is needed when accessing components of the value.

Dereference ptr before taking component of value

-> does both the dereferencing and the component selection in one go.

```
struct particle *muon; //a pointer to a struct
(more code here)

(*muon).x = 3;

or

muon->x = 3;
```

# Pointers and arrays

- We can also create pointers to arrays… except that it turns out, we don't need to!
- The "base name" of a array variable can be treated as a pointer by C, and assigned to pointer variables.

C will convert "array style" and "pointer style" accesses.

```c
#include <stdio.h>

int main(void){
    int a[5] = {1,2,3,4,5};
    int *p1 = &a[0];  // this is long-hand way to get pointer to array
    int *p2 = a;      // this is C's short-hand (i.e "a" IS a pointer)
    printf("\n %p %p \n", p1, p2);  // p1 and p2 point to same address

    *p1 = 3; //a[0] is now 3; could have used p2 instead of p1
    if (p1[2] == a[2])
        puts("true");
    if(*(p1+4) == a[4])
        puts("true - this is called pointer arithmetic");

    return 0;
}
```

# Arrays and Functions

- When you pass an array to a function, C actually passes a pointer.
- This means that modifying the array in a function *does* modify the same array you passed to it.
- However, the function cannot know the "size" of the array you passed it (it just gets a pointer). You must also pass the size of the array to a function to let it know how big the array is.

```
int f(int n, int a[]);

int array[5];

f(5,array); //pass size appropriately
f(sizeof(array)/sizeof(array[0]),array); //or this
```

sizeof(a) in f is the size of a pointer, not the size of array!
sizeof(a[0]) *is* the size of an int, still!
We use n in any loops in the function that need size of a.

# Multidimensional Arrays

- `int f(int n, int m, int a[][m]);  //function prototype`
- `int array[5][3];`
- `f(5, 3, array); // function call`

# Multidimensional Arrays

- For multidimensional arrays, things are a bit more complex. (But less complex with C99!)
- The function will get passed a pointer to an array of one less dimension. So, the function will know about all the sizes except the left-most dimension.
- (You *must* still specify the sizes yourself so the function can work out what those dimensions should be!)

C99: You can use variables to specify an array dimension

```
int f(int n, int m, int a[][m]);

int array[5][3];

f(5, 3, array); //pass size appropriately
```

sizeof(a) in f is the size of a pointer, not the size of a!
sizeof(a[0]) *will* be the size of an array of 3 ints!
sizeof(a[0][0]) *will* be the size of an int!

# Early exit of a program.

- While "return" allows you to return a value from a function, exiting the function itself…
- …it doesn't let you end the entire program. (return in main, of course does this as a side effect!)
- If you ever need to stop a program suddenly (usually because something has gone wrong), you can use the `exit` function, which is defined in `stdlib.h`

```
exit(1);
```

exit the program *right this moment*, with a return code set to 1

# Command line arguments

* Now that we know more about how to write functions, and about pointers...
* The function `main` is special in that we can declare it in two different ways.
* The first signature, `int main(void)`, is what we've been using up to now.
* The second signature, `int main(int argc, char * argv[])`, adds two parameters to our function.
* If we declare `main` like this, then the compiler will use the parameters to tell us about arguments passed to the program when it is executed on a command line.

# Command line arguments

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

* The value of `argc` will be the number of arguments passed to the program (plus 1, as the first "argument" is always the name of the program).
* `argv` is an array of strings that contain, in order, the text of each argument.
* (`argv` is of type `char * []` and not `char[][]` because the different arguments will not all have the same length, and a multidimensional array must be square. Instead, we use the array/pointer magic that C provides to access the char * like separate arrays.)

Compare argc to $# in bash, argv to $@ in bash.

# Command line arguments example

```
#include <stdio.h>

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

        int tmp, result = 0;

        if (argc != 4) {
                puts("Please provide exactly 3 arguments!");
                return 1; //non-zero error code - see Gareth's slides
        }

        for(int i=1; i<4; i++) {
                sscanf(argv[i], "%d", &tmp);
                result += tmp;
        }

        printf("The sum of the numbers is: %d\n", result);

        return 0; //zero error code, as success!

}
```

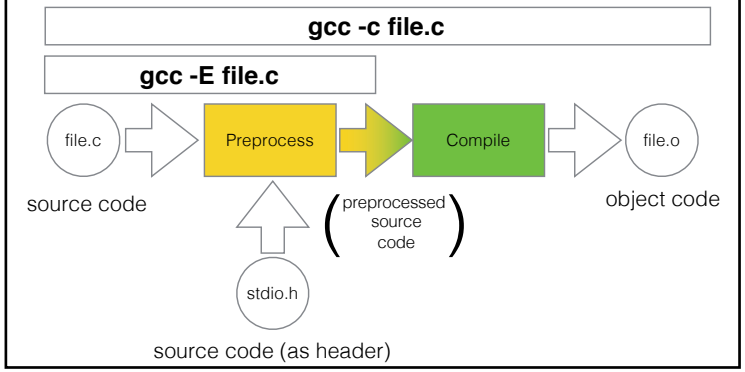
```
gcc -Wall cla.c
./a.out 1 2 4

The sum of the numbers is: 7
```

# Building Multi-file projects

- Up to now, all the code we've written has been in a single file.
- Managing a large project like this quickly becomes unmanageable.
- We need to know how to combine code in multiple source files into one program.
- In order to do this, we also need to understand more deeply how C code is taken from source to final executable.

# The C Build Process




# The C Build Process