

C PROGRAMMING CRASH COURSE

Kai Mast

CS537

Fall 2022

THE C LANGUAGE IN A NUTSHELL

A statically-typed **systems programming language**.

Developed with (and for) the UNIX operating system.

- Exactly 50 years old this year!

Can be compiled into machine code and executes without a runtime or garbage collector.

Note that there are **different C standards**

- The example code is in “modern” C11
- Some code you encounter in the class might use a different version

A BASIC C PROGRAM

The main function is invoked at the start of a program

```
int main() {  
    char *str = "World";  
    printf("Hello %s!\n", str);  
    return 0;  
}
```

Functions can return a value

The return value of the main function is the exit code of the programs

BASIC DEV WORKFLOW

Edit

```
vim mycode.c
```

Compile

```
gcc mycode.c -o mybin
```

Run

```
./mybin
```

- More complex codebases also need to link binaries together
- Use Makefiles (or similar tools) for bigger projects

C TYPES AND TYPE CONVERSION

Variable Declaration

[type] [name], e.g.:

- `int foo;`
- `char bar;`
- `float xyz;`
- `unsigned long var;`

Explicit Type Conversion

Put new type in brackets

```
int a = 5;  
char b = (char)a;
```

Similar style as in Java and C#, which both derived their syntax from C/C++

FUNCTION DECLARATION AND DEFINITION

```
int my_func();
```

Declares the function **signature** so it can be called

```
int main() {  
    return my_func();  
}
```

main can use my_func before it is fully defined*

```
int my_func() {  
    return 0;  
}
```

Defines the actual behavior of the function

*in older C standards functions could be used without being declared but that caused all sorts of issues and is not allowed in modern C

IMPLICIT TYPE CONVERSION

C automatically converts variable types if the new type is larger*

```
char c = 5; // usually 1 byte
```

```
int i = 2; // usually 4 bytes
```

```
int result = i+c; // c gets converted to 'int'
```

**it is a little more nuanced than that but “larger” should be sufficient to understand the behavior in most cases*

POINTERS

Pointers are references to an address in memory

- Can point to values on the heap, the stack, **or** the program's data segment

Be careful: Pointers can be invalid!

- Can point to uninitialized values
- Can point to values outside of the bounds of the processes virtual memory

POINTER SYNTAX

Operators

`&var` - Get the address of a variable

`*var` - De-reference a pointer (get the value the pointer points to)

Variable Declaration

- Regular integer: `int i;`
- Pointer to integer: `int *i;`
- Declaring multiple variables: `int *i, j, *k;`
 - `i` and `k` are pointers
 - `j` is a regular integer

MEMORY ALLOCATION

```
int main() {  
    int a = 5;
```

variables on the stack stay
alive while in scope

```
    int *b = malloc(sizeof(int));
```

```
    *b = 5;
```

malloc() reserves
space on the heap

```
    printf("Value is %i", (a + *b));
```

```
    free(b);
```


```
    return 0;
```

```
}
```

allocated heap space stays
reserved until free() is called
(or the program terminates)

POINTERS EXAMPLE

Code

PC  `void func() {
 int a, *b, c, *d;
 a = 3;
 b = &a;
 *b = 4;
 c = *b;
 d = &c;
 *d = *b + 1;
 d = b;
}`

Stack

a	??
b	??
c	??
d	??

All variables start uninitialized with a random value

POINTERS EXAMPLE

Code

```
void func() {  
    int a, *b, c, *d;  
    a = 3;  
    b = &a;  
    *b = 4;  
    c = *b;  
    d = &c;  
    *d = *b + 1;  
    d = b;  
}
```



Stack

a	3
b	??
c	??
d	??

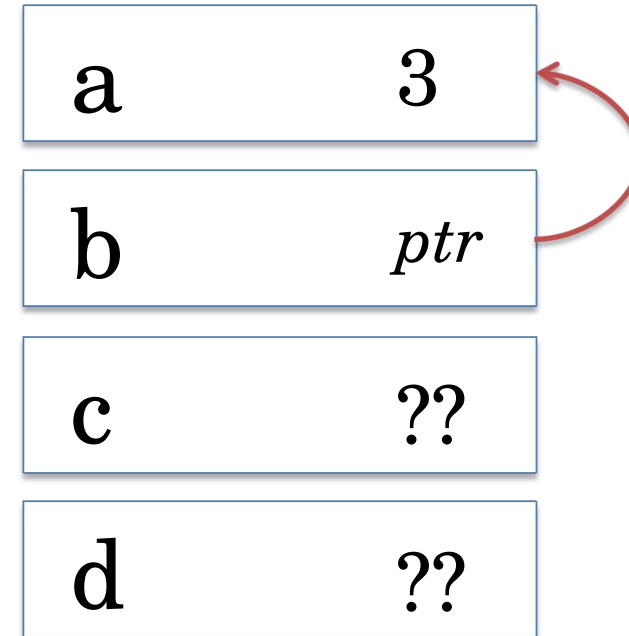
POINTERS EXAMPLE

Code

```
void func() {  
    int a, *b, c, *d;  
    a = 3;  
    b = &a;  
    *b = 4;  
    c = *b;  
    d = &c;  
    *d = *b + 1;  
    d = b;  
}
```



Stack



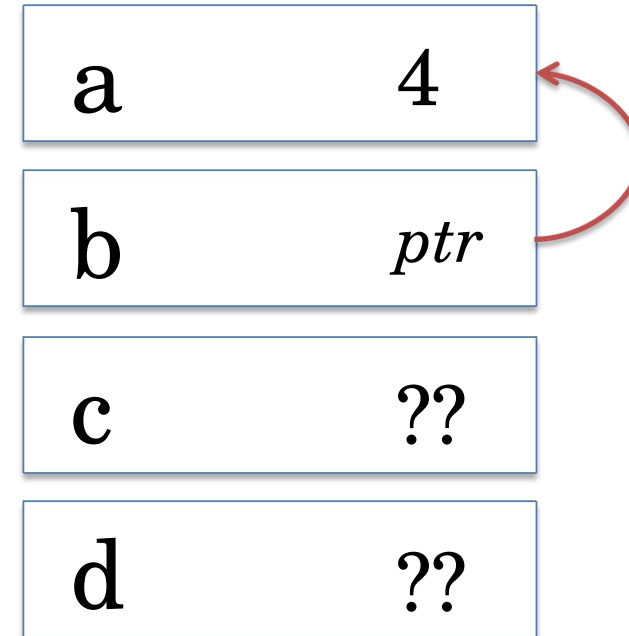
- The & operator gets variable's address
- b is now a pointer to a
(its value is the address of a)

POINTERS EXAMPLE

Code

```
void func() {  
    int a, *b, c, *d;  
    a = 3;  
    b = &a;  
    *b = 4;  
    c = *b;  
    d = &c;  
    *d = *b + 1;  
    d = b;  
}
```

Stack



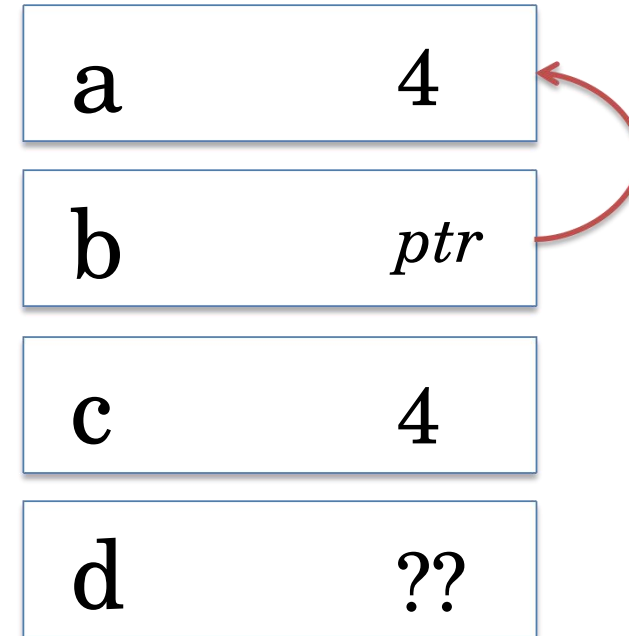
The * operator allows modifying the pointers underlying value

POINTERS EXAMPLE

Code

```
void func() {  
    int a, *b, c, *d;  
    a = 3;  
    b = &a;  
    *b = 4;  
    c = *b;  
    d = &c;  
    *d = *b + 1;  
    d = b;  
}
```

Stack



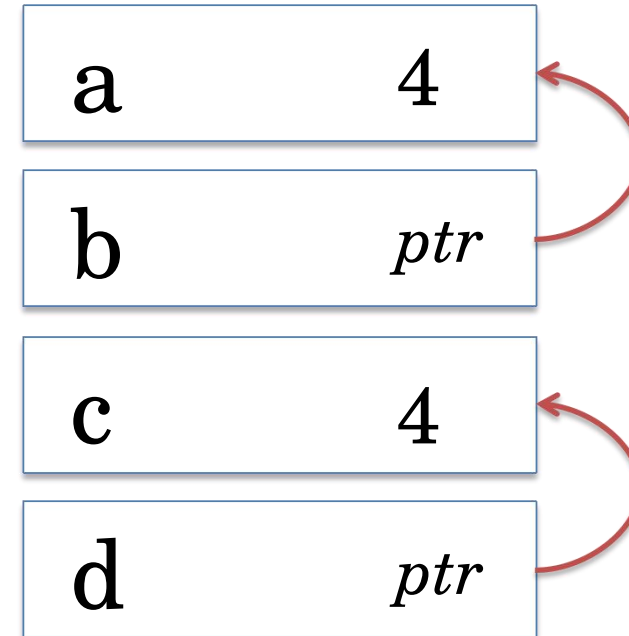
The * operator allows accessing the pointers underlying value

POINTERS EXAMPLE

Code

```
void func() {  
    int a, *b, c, *d;  
    a = 3;  
    b = &a;  
    *b = 4;  
    c = *b;  
    d = &c;  
    *d = *b + 1;  
    d = b;  
}
```

Stack

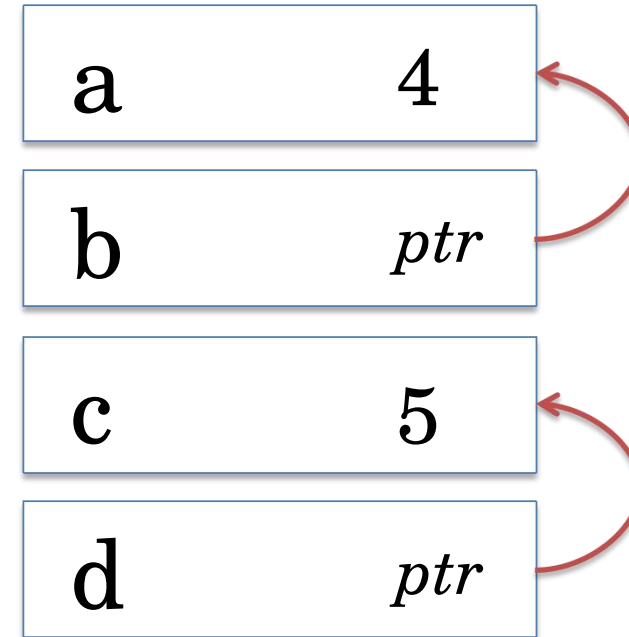


POINTERS EXAMPLE

Code

```
void func() {  
    int a, *b, c, *d;  
    a = 3;  
    b = &a;  
    *b = 4;  
    c = *b;  
    d = &c;  
    *d = *b + 1;  
    d = b;  
}
```

Stack



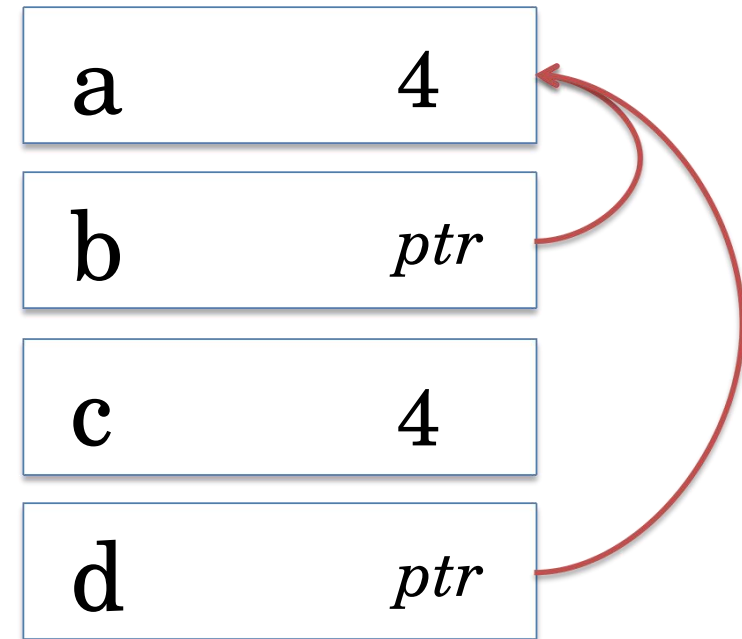
POINTERS EXAMPLE

Code

```
void func() {  
    int a, *b, c, *d;  
    a = 3;  
    b = &a;  
    *b = 4;  
    c = *b;  
    d = &c;  
    *d = *b + 1;  
    d = b;  
}
```

PC

Stack



- You can always change pointers to point to something else
- d now points to a instead of c

ARRAYS

Arrays on the stack:

```
bool array1[42];
```

Arrays on the heap:

```
bool *array2 = malloc(sizeof(bool)*42);
```

Both arrays are accessed the same way:

```
array1[5] = true;
```

```
array2[3] = false;
```

ARRAYS (CONT.)

```
int *i = {5, 3, 7};
```

is the same as

```
int i[] = {5, 3, 7};
```

and the same as

```
int i[3] = {5, 3, 7};
```

COMMON ERROR 1

Call free on an array stored on the stack

```
void func() {  
    int *i = {1, 2, 3};  
    [...]  
    free(i);  
}
```

i will automatically be deallocated when func terminates

every malloc() must have a matching free() and vice-versa

MEMORY MANIPULATION

`int memcmp(const void *p1, const void *p2, size_t n)`

- Compares the content of two memory regions of size n
- Make sure both pointers are valid and the memory regions are at least n bytes long

`void* memcpy(void *dest, const void *src, size_t n)`

- Copies the first n bytes of src to dest
- Make sure you allocate a memory region for dest before calling this function

POINTER MANIPULATION

```
char *mystring = "foobar";
```

```
char first_char = *mystring
```

```
// move pointer by sizeof(char)  
mystring = mystring+1;
```

```
char second_char = *mystring;
```

PROBLEM 1: POINTER MANIPULATION

```
int numbers[] = {1, 2, 3, 4};
```

```
int *ptr = &numbers[1];
```

```
 var = *(ptr+2) + 1;
```

What is the type and value of var?

What is the type and value of var?

var == 5

PROBLEM 2: MEMORY MANIPULATION

```
float numbers1[] = {1.0, 2.0, 3.0, 4.0};  
float numbers2[2];
```

```
memcpy(numbers2, numbers1, 2*sizeof(float));  
numbers1[1] += 5.0f;
```

What is the final
value of numbers2?

{1.0, 2.0}
the increment only affects
the value in numbers1

COMMON ERROR 2

Call free() on pointer with non-zero offset

```
int *ptr = malloc(16);
```

```
int *ptr2 = ptr+1;
```

```
free(ptr2); // wrong
```

```
free(&ptr[1]); // wrong
```

```
free(ptr); // correct
```

EXPLICIT POINTER CONVERSION

```
float f = 1.0f;
```

```
// Converts float pointer to char pointer  
char *float_ptr = (char*)&f;
```

```
// Gets the first byte of the float  
char first_byte = float_ptr[0];
```

```
// Gets the second byte of the float  
char second_byte = *(float_ptr + 1);
```

IMPLICIT POINTER CONVERSION

- `void*` can be implicitly converted to any pointer
 - `malloc` returns `void*`!
- Useful if the type of the underlying data is not known

// does not compile

```
int i = 0;  
bool *ptr = &i;
```

// compiles

```
int i = 0;  
void *ptr2 = (void*) &i;  
bool *ptr = ptr2;
```

COMMON ERROR 3

Casting integer to pointers

```
int i = 0;  
char *ptr = (char*)i;  
  
// might crash  
ptr[0] = 'a';  
// will definitely crash  
free(ptr);
```

There are some legitimate reasons for this, but most projects should not need it

POINTER ARGUMENTS

```
void set_value(int *var) {  
    *var = 1;  
}
```

```
int main() {  
    int my_var = 0;  
    set_value(&my_var);  
    [...]  
    return 0;  
}
```

C always passes arguments **by value**

- their value is copied to a new variable
- pointers allow passing a reference instead

PROBLEM 3: POINTER ARGUMENTS

```
void function(int *val) {  
    int i = 1;  
    val = &i;  
}
```

```
int main() {  
    int a=5, b=3;  
    int *ptrs[] = {&a, &b};  
    function(ptrs[0]);  
    return a;  
}
```

What is the return value of main?

5
(val is passed by value and a is never updated)

POINTING TO POINTERS

```
void create_array(int **ptr, int len) {  
    *ptr = malloc(sizeof(int)*len);  
}
```

```
int main() {  
    int *array;  
    create_array(&array, 42);  
    [...]  
    free(array);  
    return 0;  
}
```


FUNCTION POINTERS

```
void increment(int* val) {  
    *val += 1;  
}
```

```
void decrement(int* val) {  
    *val -= 1;  
}
```

Both Functions have
the same **signature**

FUNCTION POINTERS (CONT.)

```
void (*func)(int*) = increment;  
int val = 1;
```

```
// Increment `val`  
func(&val);
```

```
// Set to different function  
func = decrement;
```

```
// Decrement `val`  
func(&val);
```

func stores the address to increment

the value of **func** can be changed like other variables

C STRINGS

Stack-Allocated Strings

```
char* mystring = "cs537";
```

Dynamically-Allocated Strings

```
char* mystring = malloc(6);  
strcpy(mystring, "cs537");
```

STRINGS ARE NULL-TERMINATED

```
char *mystring = "cs537";
```

is the same as

```
char *mystring =  
    {'c', 's', '5', '3', '7', '\0'};
```

NULL-termination allows **detecting the end** when traversing a string

PROBLEM 4: STRINTG TERMINATION

```
#include "string.h"
```

```
int main() {  
    char *str = {'x', '\0', 'y', 'z'};  
    printf("%s", str);  
    return 1;  
}
```

What is the output of this program?

“x”

print will only insert the string up to the NULL-byte

STRING MANIPULATION

size_t strlen(**char** *str)

Get the length of a string (including NULL)

int strcmp(**const char** *str1, **const char** *str2)

Compares the content of two strings and returns 0 if they are equal

char* strcpy(**char** *dest, **const char** *src)

Compares the content of two strings and returns 0 if they are equal

PROBLEM 5: STRING COPY

```
#include "string.h"
```

```
int main() {  
    char *str1 = "hello world";  
    char str2[6];
```

```
    strcpy(str2, str1+6);  
    printf("%s\n", str2);
```

```
    return 0;
```

```
}
```

What is the console output of this program?

“world”
Only the second word is copied

IMPLICIT BOOLEAN CONVERSION

```
int a = 1;  
int b = 0;
```

integers != 0
evaluate as true

```
if (a) {  
    // this branch will be executed  
}
```

integers == 0
evaluate as false

```
if (!b) {  
    // this branch will also be executed  
}
```


IMPLICIT BOOLEAN CONVERSION (POINTERS)

```
int a = 1;  
int *ptr1 = &a;  
int *ptr2 = NULL;
```

pointers != NULL
evaluate as true

```
if (ptr1) {  
    // this branch will be executed  
}
```

pointers == NULL
evaluate as false

```
if (!ptr2) {  
    // this branch will also be executed  
}
```

PROBLEM 6: BOOLEAN CONVERSION

```
int main() {  
    char var = 0;  
    char *ptr = &var;  
  
    if (var || ptr) {  
        printf("A\n");  
    } else {  
        printf("B\n");  
    }  
    return 0;  
}
```

What is the console output of main()?

“A”
ptr is not NULL and evaluates to true

STRUCTURES

```
struct student {  
    int age;  
    float gpa;  
};
```

```
void func() {  
    struct student s;  
    s.age = 21;  
    s.gpa = 4.0f;  
}
```

C is not an object-oriented language

- Struct do not have methods
- No inheritance or interfaces

But, object-like functionality can be implemented combining structs and function pointers.

UNIONS

```
union my_union {  
    int val1;  
    char val2;  
    bool val3;  
};
```

```
void func() {  
    union myunion u;  
    u.val1 = 5;  
}
```

Union can be any *one* of its fields

Be careful:

Initializing one field and then accessing another on the same union is undefined behavior!

THE PREPROCESSOR

- Runs before the compiler
- Not a Turing-complete language
- Only supports text replacement and basic if statements

```
#define WINDOWS
```

```
#if defined WINDOWS  
    [win-specific code]  
#elif defined UNIX  
    [unix-specific code]  
#else  
    #error "Unknown OS"  
#endif
```

PREPROCESSOR CONSTANTS

We can use the preprocessor to define constants

```
#define NUMBER 5  
int my_value = NUMBER;
```

But in modern C it is better to use the const keyword

```
const int NUMBER = 5;  
int my_value = NUMBER;
```

THATS ALL, FOR NOW

There are many things we did not cover, including

- type definitions
- switch statements
- inline assembly
- static and extern keywords
- volatile variables
- enumerations
- various legacy features
(e.g., K&R function declarations)
- threading and synchronization primitives

THATS ALL, FOR NOW

All problems are on github

<https://github.com/kaimast/c-crashcourse>

For more in-depth coverage of C, take a look at these textbooks.

