EC440: C Programming Crash Course

September 12, 2022

Adapted from CS3224 Operating Systems Lecture 2 by Prof. Dolan-Gavitt

Learn C in 100 minutes

- Just kidding, it takes longer than that!
- Tons more material online. Here are some starting points:

Resource	Why use it?
Brian Kernighan and Dennis Ritchie *The C Programming Language* https://archive.org/details/cprogramminglang00denn (also known as K&R)	• Familiar format for textbook learners • Similar content to these slides, but more detailed • It is a well-known book with programmers, so it is at least worth skimming so you can join a conversation
The C Language Specification: http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf	• Note: The cppreference wiki often contains the same information in a distilled form. But understanding the writing style of a specification document can be generally useful. • Useful when aiming to follow the standard language • Language specifications are useful if you need to implement something from a *future* version of a language
Man pages! See section 3 for a lot of library functions. E.g. *man 3 printf*	• The manual is at your fingertips without leaving your console ○ Many text editors have shortcuts to open the man page for a function you have selected • Answers *how* to use a function or feature and *what* it does. Often includes example snippets.

https://en.cppreference.com/w/c/language	• Searchable. Use as a reference document
	  ○ Which functions exist in a category?
	  ○ How do I use this function? Can I use it in my version of C?
	• Downloadable/offline versions exist on the website.

Scope of this Session

- You don’t need to know everything in C for this course.
  (or for a C programming job, for that matter)
- These slides cover language features **you are likely to use**.
- It can be faster to write short experimental programs instead of searching for direct answers to your questions.
    - I hope this session shows you enough to try that out
- Come to office hours if you feel stuck!

What we'll cover

- A basic C program
    - Structure
    - Building/running
- Functions
- Variables
- Operators
- Control Flow Statements
- Arrays
- Strings
- Pointers
- Casting
- Structures
- Memory
- Build Process (more advanced)
    - Multiple files
    - Makefiles
- String parsing
- What to do when your code doesn't work!

A First Program

hello.c

```
#include <stdio.h>

int main() {
    printf("Hello, world\n");
    return 0;
}
```

```
$ gcc hello.c -o hello
$ ./hello
Hello, world
$
```

How do we get from here...

...to here?

A First Program

Declaration of function "main", returning type "int"

```
#include <stdio.h>

int main() {
    printf("Hello, world\n");
    return 0;
}
```

A First Program

Calling the printf function with a string parameter

```
#include <stdio.h>

int main() {
    printf("Hello, world\n");
    return 0;
}
```

A First Program

Each statement must end with a semicolon

```
#include <stdio.h>

int main() {
    printf("Hello, world\n");
    return 0;
}
```

A First Program

```
#include <stdio.h>

int main() {
    printf("Hello, world\n");
    return 0;
}
```

Return the value 0 (exit code of the program; executed successfully!)

A Closer Look...

Where does printf come from? Why can we call it?

```
#include <stdio.h>

int main() {
    printf("Hello, world\n");
    return 0;
}
```

A Closer Look...

Printf is a function. But how do we know?

Check the man pages!

```
$ man 3 printf
PRINTF(3)         Linux Programmer's Manual        PRINTF(3)

NAME
       printf,   fprintf,   dprintf,   sprintf,  snprintf,
       vprintf, vfprintf, vdprintf, vsprintf, vsnprintf  -
       formatted output conversion

SYNOPSIS
       #include <stdio.h>
...
GNU                     2017-09-15                PRINTF(3)
```

A Closer Look...

Printf is a function. But how does the program know?

```
#include <stdio.h>

int main() {
    printf("Hello, world\n");
    return 0;
}
```

The C ***Preprocessor*** expands macros, ***before the program is compiled***. The preprocessor replaces the #include macro with the contents of stdio.h

View the preprocessor output with gcc -E hello.c:

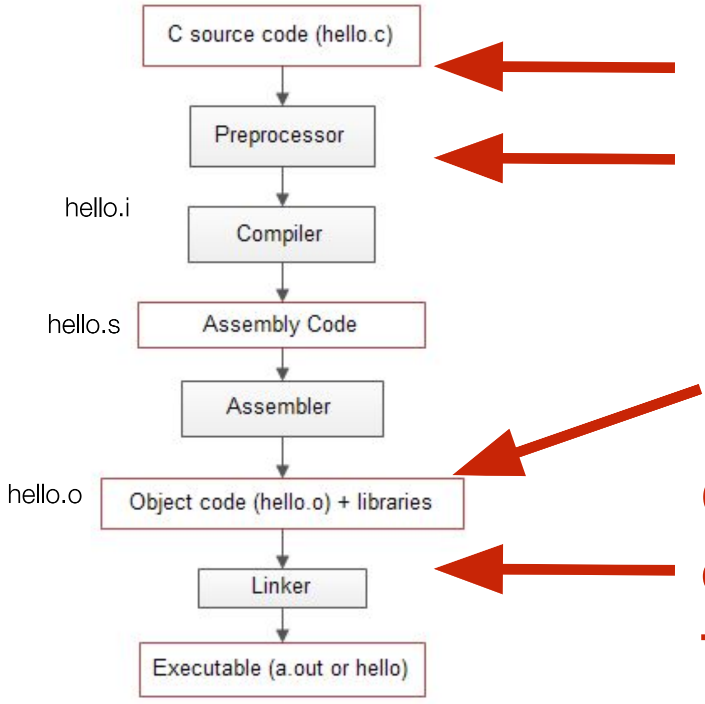
```
# 1 "hello.c"
#
typedef long unsigned int size_t;
    const char *__restrict __format, ...);
...
int printf (const char *__restrict __format, ...);
...
# 2 "hello.c" 2

# 2 "hello.c"
int main() {
    printf("Hello, world\n");
    return 0;
}
```

printf declaration

But what does printf do - where is the definition?

The search for printf

`gcc -save-temps hello.c -o hello`



uses printf

#include macro replaced with stdio.h which contains the printf definition

The object header indicates that the code relies on an undefined "symbol" printf.

GCC links against libc.so, by default. libc exposes a symbol for the printf function.

Functions

- All code in C lives in a function

- Declaring a function:

Return type **Function name**

```
double add(double x, double y) {
    return x + y;
}
```

Parameters

- Calling it:

```
double result = add(4.9, 12.1);
```

Declaring Variables

```c
#include <stdio.h>

int main() {
    int height = 70;
    int age;
    age = 31;
    printf("I'm %d years old and %d inches tall\n",
           age, height);
    return 0;
}
```

Variable type

Variable name

Initial value

Common Mistake: A variable is read from before it is assigned a value.
How to avoid: Use compiler warnings. Test your programs with valgrind or sanitizers. Try not to create variables until you know what to assign (requires *-std=c99* or later).
Caution: Assigning fake/placeholder values to a variable often hides the problem from the above tools, and doesn't fix the problem.

C Built-in Types

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	8 bytes	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615

These ranges are different on different platforms and implementations of C! Here I'm giving the ranges for 32-bit x86.
From the Spec:

> “A ‘‘plain’’ int object has the natural size suggested by the architecture of the execution environment”

C Operators

- Arithmetic:
  - + (plus), - (minus), / (division), * (multiplication), % (modulus)
- Logical
  - && (and), || (or), ! (not)
- Relational
  - < (less than), > (greater than), >= (greater or equal),
    <= (less or equal), == (equal), != (not equal)
- Assignment:
  - x = y: simple assignment
  - x += y: same as x = x + y
  - -=, *=, /=, <<=, >>=, %=, &=, |=

C Operators

- Bitwise:
    - ~x: one's complement (i.e., flip all bits)
    - x << n: shift x left by n bits
    - x >> n: shift y right by n bits
    - x & y: bitwise AND of x and y
    - x | y: bitwise OR of x and y
    - x ^ y: bitwise XOR of x and y
- (Pre/Post) Increment/decrement
    - x++: evaluate x, then increment it
    - ++x: increment x, then evaluate it
    - x--: evaluate x, then decrement it
    - --x: decrement, then evaluate it

Increment example

```
#include <stdio.h>

int main() {
    int x = 0;
    if (x++ > 0) {
        printf("1\n");
    }
    if (x > 0) {
        printf("2\n");
    }
    x = 0;
    if (++x > 0) {
        printf("3\n");
    }
    if (x > 0) {
        printf("4\n");
    }
    return 0;
}
```

Increment example

```c
#include <stdio.h>

int main() {
    int x = 0;
    if (x++ > 0) {
        printf("1\n");
    }
    if (x > 0) {
        printf("2\n");
    }
    x = 0;
    if (++x > 0) {
        printf("3\n");
    }
    if (x > 0) {
        printf("4\n");
    }
    return 0;
}
```

```
$ ./main
2
3
4
```

Control Flow

- Branching:

    - if (*condition*) {
    statements
}
else if (*condition*) {
    statements
}
else {
    statements
}

Braces are optional when there is only 1 statement inside

Common mistake: No braces are used. Some time in the future you add or remove a line somewhere, and now the if-condition does something unintended. Example: Apple's goto fail bug

How to avoid (or try to): Enable compiler warnings. Always use braces (debatable whether this helps). Keep your functions short, with focused intentions (so mistakes are easier to see)

Control Flow

- Branching:
  - switch (*expression*) {
    case 1:
      statements
      break;
    case 2:
      statements
      break;
    default:
      statements
      break;
    }

Common mistake: Missing *break*. Why doesn't C just leave the *switch* when the next *case* starts?

Common mistake: Missing a case.

How to avoid: Use (and fix) compiler warnings. *Switch* over types with well-understood domains. Only use *default* cases if you truly have a default case to handle.

Control Flow

- Looping

  while (*condition*) {
      statements
  }

  do {
      statements
  } while (*condition*);

Common Mistake: The program never ends.

How to avoid: Use compiler warnings. Test your program. Use code coverage tools to make sure you are actually testing what you think you are testing.

How to find where the code is stuck: Run your program inside *gdb*:

- When you reach a point where it feels stuck. Press CTRL+C to stop the program and receive a gdb prompt.
- Use the *list* command to show where the code has stopped.
- You might be in a library your code uses, instead of in your code itself. Use the *backtrace* command to show who called this function.
- Use the *frame <# from backtrace>* command to go to one of the functions in your code, and identify which loop is contained.
- Now you have to find and fix the actual bug. Other early commands are *print* (show a variable value), *info locals* (show all variable values in the current frame), and *continue* (make the program keep running).
- Hang up a gdb reference on your wall. e.g. https://users.ece.utexas.edu/~adnan/gdb-refcard.pdf

Control Flow

- Looping
- for (*init* ; *condition* ; *update*) {
      statements
  }

  Example:

```
int i;
for (i = 0; i < 100; i = i + 1) {
    printf("This is iteration %d\n", i);
}
```

Arrays

1. Arrays in C are *fixed-length* (their size must be specified at compile time)

2. To declare an array:
   int x[10];

3. Then individual elements are accessible as
   x[i] where i is an integer

Array Bounds

- Unlike higher-level languages, C does not check array bounds or keep track of them at runtime.
  - In some cases, the compiler can detect out-of-bounds accesses, but it's not part of the language
- Programming in C requires you to be very careful when programming; there are basically no safety features

Out Of Bounds Example

```
#include <stdio.h>

int main() {
    int x[5];

    x[10] = 9;

    printf("x[10] = %d\n", x[10]);
    return 0;
}
```

Segmentation fault

(if you are lucky)

```
$ gcc oob.c -o oob
oob.c:6:5: warning: array index 10 is past the end of the array
(which contains5 elements) [-Warray-bounds]
    x[10] = 9;
    ^ ~~
oob.c:4:5: note: array 'x' declared here
    int x[5];
    ^
oob.c:8:28: warning: array index 10 is past the end of the array
(which contains 5 elements) [-Warray-bounds]
    printf("x[10] = %d\n", x[10]);
                           ^ ~~
oob.c:4:5: note: array 'x' declared here
    int x[5];
    ^
2 warnings generated.

$ ./oob
x[10] = 9
Segmentation fault
```

Defining Strings

- A C string is an array of characters
- The final element in the array is NULL ('\0')
- So:

```
char hello[] = "hi";
```

is the same as

```
char hello[] = {'h', 'i', '\0'};
```

- When C was developed, memory was extremely limited. The Null terminator '\0' adds a 1-byte to store the length of the string and can represent arbitrarily long strings.

Pointers

- A pointer in C is a variable that stores the *address* of another variable
- To declare:
  `int* foo;   (or: int *foo;)`
- Two new operators:
  `&x`: get the address of variable x
  `*p`: access the value of the pointed-to variable
  (called *dereferencing*)

Pointers

```c
#include <stdio.h>

int main() {
    int* p = NULL;
    int y = 20;
    int x = 10;

    p = &x;
    *p = 5;
    p = &y;
    *p = 50;

    return 0;
}
```

Casting Types

- Some conversions need an *explicit cast* – a way of telling C that you really do want that conversion

- For example:

```
int x = 90000;
short y = (short) x;
printf("x = %d, y = %d\n", x, y);
```

prints

```
x = 90000, y = 24464
```

Casting

- Pointers of one type can be cast to another type
- As usual, C does ***not*** stop you from shooting yourself in the foot with this
- But it does let you do some handy things too!

```c
#include <stdio.h>

int main() {
    char x[] = "POLY";
    int *y;
    y = (int *) &x;
    printf("%s is %d\n", x, *y);
    return 0;
}
```

Output

POLY is 1498173264

?

Casting

- Pointers of one type can be cast to another type
- As usual, C does ***not*** stop you from shooting yourself in the foot with this

```
#include <stdio.h>

int main() {
    char x[] = "POLY";
    int *y;
    y = (int *) &x;
    printf("%s is %d\n", x, *y);
    return 0;
}
```

"POLY" = {'P', 'O', 'L', 'Y', '\0'}
= 0x50, 0x4f, 0x4c, 0x59, 0x00
= 0x594c4f50
= 1498173264

Structures

- Sometimes we want more complicated data types than just int, char, etc.

- For this we can define a ***struct*** – an aggregate type that contains several fields

Struct Example

```
#include <stdio.h>

struct Person {
    int age;
    int height_cm;
    int weight_kg;
};

int main() {
    int i;
    struct Person p[2];

    p[0].age = 60;
    p[0].height_cm = 150;
    p[0].weight_kg = 90;

    p[1].age = 21;
    p[1].height_cm = 180;
    p[1].weight_kg = 80;

    for (i = 0; i < 2; i++) {
        printf("Person %d is %d years old and weighs %d kg\n",
            i, p[i].age, p[i].weight_kg);
    }
    return 0;
}
```

One Last Operator

- If you have a pointer to a structure, you can dereference the pointer and access its member in one step:

```
struct Person *p = &q;
p->age = 64;
```

Arrays and Pointers

- Arrays and pointers have a special relationship in C
- An array can be treated as a pointer in most contexts, and vice versa
- If you hand an array of type T to something that expects a pointer to type T, it's treated as if it were a pointer to the first element of the array

Pointer Arithmetic

- You can do *arithmetic* on pointers

- Given a pointer P of type T, P + N will point to the memory at address P + (N * sizeof(T))

- For example:

```
int x[4] = {1, 2, 3, 4};
int *p = x;
p = p + 2;
printf("*p = %d\n", *p);    => Prints "*p = 3"
```

Pointer Arithmetic vs Array Indexing

- You may have noticed that x[2] and *(p + 2) refer to the same element
- This is not an accident!
- In fact, we can use array index notation with pointers as well:

```
int x[4] = {1, 2, 3, 4};
int *p = x;
printf("p[2] = %d\n", p[2]);    => Prints "p[2] = 3"
```

Memory Allocation

- C only supports *fixed-length* arrays
- So how do you deal with variable amounts of data?
- To allocate: p = malloc(*size in bytes*) – returns a pointer to a memory region that you can then assign to a variable of whatever type you like
- To free: free(p)

Memory Allocation

- Once again, C is not going to hold your hand:
  - You have to know how much space you need
  - You are responsible for freeing memory you allocate
  - You are responsible for making sure you don't try to put too much data into too small a buffer

Example: Linked List

```
struct list {
    int data;
    struct list *next;
};
 struct list * list_insert(struct list *head, int data) {
     struct list *new = malloc(sizeof(struct list));
     new->data = data;
     new->next = head;
     return new;
 }
```

Memory allocation

sizeof operator gets the size of a type

Example: Linked List

```
struct list {
   int data;
   struct list *next;
};
 struct list * list_insert(struct list *head, int data) {
     struct list *new = malloc(sizeof(struct list));
     new->data = data;
     new->next = head;
     return new;
 }
```

Example: Linked List

Freeing memory

```
int main() {
    int i;
    struct list *head = NULL;
    struct list *cursor = NULL;
    // Add some stuff to our list
    for (i = 0; i < 20; i++) {
        head = list_insert(head, i * 2);
    }
    // Now print it out
    cursor = head;
    while (cursor != NULL) {
        printf("List entry at %p has data %d\n",
                cursor, cursor->data);
        cursor = cursor->next;
    }
    // And free everything
    cursor = head;
    while (cursor != NULL) {
        struct list *to_delete = cursor;
        cursor = cursor->next;
        free(to_delete);
    }
}
```

Output

```
List entry at 0x7b6517d0 has data 38
List entry at 0x7b6517c0 has data 36
List entry at 0x7b6517b0 has data 34
List entry at 0x7b6517a0 has data 32
List entry at 0x7b651790 has data 30
List entry at 0x7b651780 has data 28
List entry at 0x7b651770 has data 26
List entry at 0x7b651760 has data 24
List entry at 0x7b651750 has data 22
List entry at 0x7b651740 has data 20
List entry at 0x7b651730 has data 18
List entry at 0x7b651720 has data 16
List entry at 0x7b651710 has data 14
List entry at 0x7b651700 has data 12
List entry at 0x7b6516f0 has data 10
List entry at 0x7b6516e0 has data 8
List entry at 0x7b6516d0 has data 6
List entry at 0x7b6516c0 has data 4
List entry at 0x7b6516b0 has data 2
List entry at 0x7b6516a0 has data 0
```

Example: Linked List

Why did we have to use the extra to_delete variable?

```
int main() {
    int i;
    struct list *head = NULL;
    struct list *cursor = NULL;
    // Add some stuff to our list
    for (i = 0; i < 20; i++) {
        head = list_insert(head, i * 2);
    }
    // Now print it out
    cursor = head;
    while (cursor != NULL) {
        printf("List entry at %p has data %d\n",
               cursor, cursor->data);
        cursor = cursor->next;
    }
    // And free everything
    cursor = head;
    while (cursor != NULL) {
        struct list *to_delete = cursor;
        cursor = cursor->next;
        free(to_delete);
    }
}
```

Example: Multiple Files

hello.h:

```
#ifndef HELLO_H
#define HELLO_H

void say_hello(void);

#endif
```

hello.c:

```
#include <stdio.h>
#include "hello.h"

void say_hello(void)
{
    printf("Hello World\n");
}
```

main.c:

```
#include "hello.h"

int main(int argc, char** argv)
{
    say_hello();
    return 0;
}
```

gcc main.c hello.c -o hello

Example: Multiple Files

1 Preprocessing (multiple passes)

stdio.h hello.h

hello.c:

```
...
int printf (const char *__restrict
__format, ...);
...
void say_hello(void);


void say_hello(void)
{
	printf("Hello World\n");
}
```

main.c:

```
void say_hello(void);


int main(int argc, char** argv)
{
	say_hello();
	return 0;
}
```

Look at the output of this stage by using option -E

gcc main.c hello.c -o hello

Example: Multiple Files

1 Preprocessing (multiple passes)

2 Compiling

3 Assembling

hello.c

↓

hello.o:
110010101001...

main.c

↓

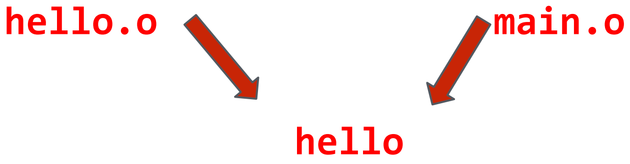
main.o:
1010100010101...

Look at the output of stage 2 by using option -S

gcc main.c hello.c -o hello

Example: Multiple Files

1 Preprocessing (multiple passes) 2 Compiling 3 Assembling 4 Linking



gcc main.c hello.c -o hello

Aside: Makefiles

- Remembering and reproducing the commands to compile a project is tedious
- Larger projects may have multiple files, many compilation options, external libraries, etc.
- To manage this complexity we can use a Makefile

make

- Instead of calling gcc, we can instead just do:

```
$ make hello
cc     hello.c   -o hello
```

- The make command has some built-in rules that understand how to compile basic C and C++ programs

- The make command will check whether the file "hello" exists and is newer than the file "hello.c"; if not, it will try to build "hello" using "hello.c"

A Makefile

- For more complex rules, or to build things that make doesn't natively understand how to create, we use a Makefile:

```
CFLAGS=-Wall -g

clean:
        rm -f hello
```

Extra compile flags. Will be treated as if passed as args to gcc
See [implicit variables]() in the *make* user manual

- Now we can have make clean up for us:

```
$ make clean
rm -f hello
```

A Makefile

- For more complex rules, or to build things that make doesn't natively understand how to create, we use a Makefile:

```
CFLAGS=-Wall -g

clean:
	rm -f hello
```

The name of our new build target

- Now we can have make clean up for us:

```
$ make clean
rm -f hello
```

A Makefile

- For more complex rules, or to build things that make doesn't natively understand how to create, we use a Makefile:

```
CFLAGS=-Wall -g

clean:
	rm -f hello
```

The command to run to build the target

- Now we can have make clean up for us:

```
$ make clean
rm -f hello
```

A Makefile

- For more complex rules, or to build things that make doesn't natively understand how to create, we use a Makefile:

```
CFLAGS=-Wall -g

clean:
	rm -f hello
```

The command to run to build the target

NOTE: this must be a tab character, not spaces!

- Now we can have make clean up for us:

```
$ make clean
rm -f hello
```

Key Takeaways

- C can be complex, but you can get far with parts of it
- The man pages answer many Linux and C questions
- Memory and pointers will pop up a lot
    - Find patterns that help you work with them
    - Use tools that help find and avoid bugs

String parsing in C

- Could be useful for homework 1… ;)
- **Goal**: take an input character array and split it at each space
- **Input**: char* in = “This is a sentence”;
- **Output**: char** out = [“This”, “is”, “a”, “sentence”];

String parsing in C

- Problem 1: How do we read in text from the command line?
    - Option 1: Using **`argv`**

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

    printf("Argument supplied:\n%s\n", argv[1]); // print input
```

- Option 2: Using **`fgets`**

```
char* line = fgets(input, 512, stdin); // get max 512 input chars from stdin
```

Separating the string: plan of attack

- Assumptions:
  - Maximum of 32 individual words
  - No leading or trailing spaces

1. Declare a **`char**`** array to hold pointers to individual words
2. Find the spaces using **`strpbrk`**
3. Assign each pointer to the beginning of each word and replace spaces with ‘/0’

What if my code doesn’t work?

- Compiler warnings/errors
- Print statements
- Unit testing
- GDB

Unit Testing

Example using the assert() macro

main.c

```
#include <stdio.h>
#include <square.h>

int square(int x)
{
    return x+x;
}

int main(void)
{
    // Do a lot of stuff… Probably call square()
    return 0;
}
```

square.h

```
int square(int x);
```

test.c

```
#include <assert.h>
#include <square.h>

int main(void)
{
    assert(square(2) == 4);
    assert(square(-2) == 4);
}
```

Passes (oops)

Fails (and aborts)

GDB: The Basics

- Compile code with the -g flag to enable built-in debugging support: `gcc -g program.c -o prog`
- Allows you to set breakpoints, step through code, and examine the values of variables at different points in time
- **The basics**: `run`,`break`,`step`,`next`,`watch`, and `print`

Pointers

```c
#include <stdio.h>

int main() {
    int * p = NULL;
    int y = 20;
    int x = 10;

    p = &x;
    *p = 5;
    p = &y;
    *p = 50;

    return 0;
}
```