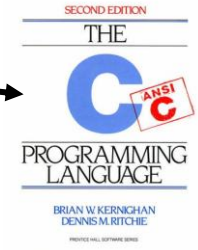
SCC.211: A guide to (relatively) pain-*free()* C programming

Dr. Andrew Scott & Damian Borowiec

1

Preface

Ideally, read this…



- We won't go into detail (not a module on C programming)
- Non-exhaustive!
- Read this guide together with the following:
  - https://cheatography.com/ashlyn-black/cheat-sheets/c-reference/pdf/
- Aims and Objectives:
  - Revisit pointers
  - Revisit structures
  - Go through some examples
  - Revisit compilation
  - Introduce "Make" – (makes life easier in the long-run)

This will help when you get stuck or the compiler is "shouting" at you…

2

© Damian Borowiec, 2020

Pointers (and arrays) in C (1)

- Pointers:
  - They are variables which store ***addresses*** of memory areas
  - The space for the address is allocated on the **stack**
    - Just like in regular variables
    - The space holds a value – the address of some memory area (e.g. on **stack** or **heap**)
    - De-referencing a pointer (*): accessing the value stored at that memory area
    - Assigning a pointer: storing a memory address in the pointer so that it can be later de-referenced
  - Some examples…

3

Pointers (and arrays) in C (2)

Line:1
- Declare a pointer of type <*int*> called **"x"**
- Allocate 32/64-bits on the stack to hold memory address
- Tell the compiler, the value pointed to by that pointer will be of type <*int*>.
- Initialize to NULL – e.g. 0 / nothing – pointer doesn't point to anything

Line:3
- Regular integer variable storing a value: 10

Line:5
- Regular integer variable storing a value: 10

Line:7
- & - "address of"
- Take the memory address/ location of **"a"** and stick it into **"x"**
- "Pointer assignment"

Line:9
- "De-referencing a pointer" **"x"**
- a.k.a. Access the contents of memory area pointed to by **"x"**
- a.k.a. Take the address held in **"x" (&a),** take the contents held at that address, stick the contents into **"b"**

```
1  int * x = NULL;
2
3  int a = 10;
4
5  int b = 5;
6
7  x = &a;
8
9  b = *x;
10
```

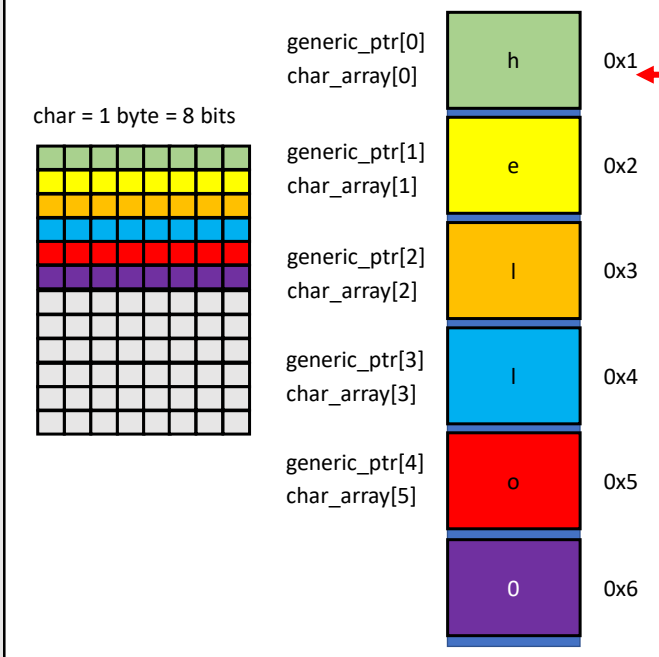
4

Pointers (and arrays) in C (3)

- Why pointers have types?
  - Add some meaning to generic memory areas.
  - I.e. makes sense to think of a memory area as an array of integers or characters
  - Use the semantics of the language to operate on this representation
- Line:1
  - Allocate 100 bytes of space and stick the beginning address into **"generic_ptr"**
- Line:2
  - Make **"char_array"** point to the same address
- Line:5-7
  - Initialize **"char_array"** with some data
- Line: 9-11
  - Use **"generic_ptr"** to access the same memory area to print out the contents
- Line: 13
  - Free the previously allocated memory areas

```
1 void * generic_ptr = malloc(100);
2 char * char_array = (char *) generic_ptr;
3 char string[] = "hello";
4
5 for(int i = 0; i < 5; i++){
6     char_array[i] = string[i];
7 }
8
9 for(int j = 0; j < 5; j++){
10    printf("%c", (char) generic_ptr[j]);
11 }
12
13 free(char_array);
```

5




```
1 void * generic_ptr = malloc(100);
2 char * char_array = (char *) generic_ptr;
3 char string[] = "hello";
4
5 for(int i = 0; i < 5; i++){
6     char_array[i] = string[i];
7 }
8
9 for(int j = 0; j < 5; j++){
10    printf("%c", (char) generic_ptr[j]);
11 }
12
13 free(char_array);
```

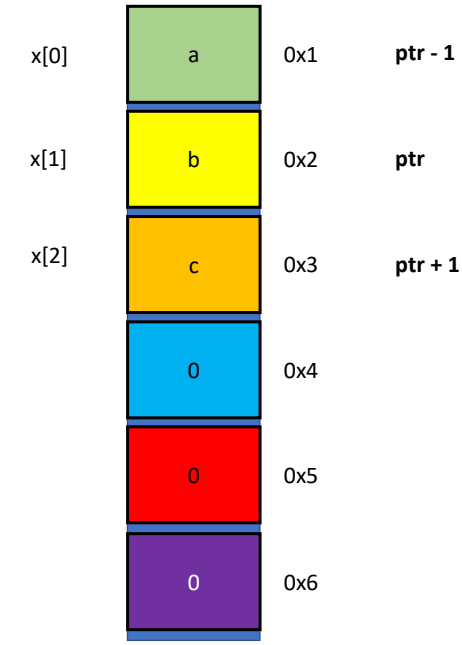
6

Pointers (and arrays) in C (4)

- Pointer arithmetic:
  - Memory addresses are just numbers
  - Can add, subtract, multiply
  - Allows us to efficiently navigate the memory
- Line:1
  - Array of characters
- Line:2
  - Pointer to a memory area containing characters
- Line:4-6
  - Character variable declarations
- Line:8
  - Stick address of second array element into **"x"**
- Line:9
  - De-reference the pointer = **var_b** holds **'b'**
- Line:10
  - ***(ptr-1)** – take the value at address held in **ptr** – 1 x <type size> i.e. if ptr held address 0xff01, **ptr-1** means 0xff00. Then de-reference that address
- Line:11
  - As above but with addition

```
1  char x[3] = {'a', 'b', 'c'};
2  char * ptr = NULL;
3
4  char var_a;
5  char var_b;
6  char var_c;
7
8  ptr = &x[1];
9  var_b = *ptr;
10 var_a = *(ptr-1);
11 var_c = *(ptr+1);
```

7

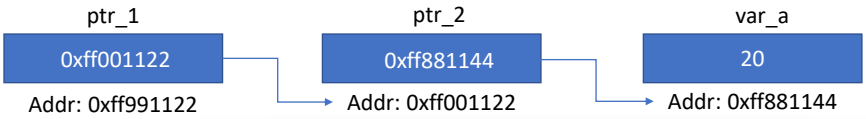



```
1  char x[3] = {'a', 'b', 'c'};
2  char * ptr = NULL;
3
4  char var_a;
5  char var_b;
6  char var_c;
7
8  ptr = &x[1];
9  var_b = *ptr;
10 var_a = *(ptr-1);
11 var_c = *(ptr+1);
```

8

Pointers (and arrays) in C (5)

- Double pointers:
  - A pointer to a pointer which points… to a value! Phew… :D




```c
int var_a = 20;
int * ptr_2 = &var_a;
int ** ptr_1 = &ptr_2;

printf("var_a: %d, *ptr_2: %d, **ptr_1: %d", var_a, *ptr_2, **ptr_1);
```

9

Pointers (and arrays) in C (6)

- Printing pointers:
  - %p format flag for printf() – 0xf2345000
  - %x format flag for printf() – f2345000

```
&G    = 0x100001078
&s    = 0x10000107c
&a    = 0x7fff5fbff2bc
&p    = 0x7fff5fbff2b0
p     = 0x100100080
main  = 0x100000e18
```

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

int G = 0;

int main(int argc, char ** argv){
    static int s;
    int a;
    int * p = (int *) malloc(sizeof(int));

    print("&G    = %p\n", (void *) &G);
    print("&s    = %p\n", (void *) &s);
    print("&a    = %p\n", (void *) &a);
    print("&p    = %p\n", (void *) &p);
    print("p     = %p\n", (void *) p);
    print("main  = %p\n", (void *) main);

    free(p);
    return 0;
}
```

10

Structures in C (1)

- Group of variables of different data types represented by 1 name

```c
struct StudentData {
  char * student_name;
  int student_id;
  int student_age;
};

struct StudentData student;
student.student_name = "John";
student.student_id = 1234;
student.student_age = 30;

printf("Student Name is: %s", student.student_name);
printf("Student Id is: %d", student.student_id);
printf("Student Age is: %d", student.student_age);

```

11

Structures in C (2) - Typedefs

- For structures which you intend to use a lot
- Declare a new type, which will represent the structure
- Later, re-use the type to quickly, declare new variables of type “mytype”

```c
typedef struct home_address {
  int street;
  char * city;
  char * country;
} addr_t;

addr_t var_1;
var_1.city = "Lancaster";

```

12

Structures in C (3) – Gotchas!

- Depending on the system and the compiler, structures may be laid out differently in memory…
  - Often, the compiler decides to "pad" structures such that the memory access performed by the CPU is on "word" boundary – much quicker
  - **Wastes some space in memory**
    - **hint hint… you may face this in your coursework…**

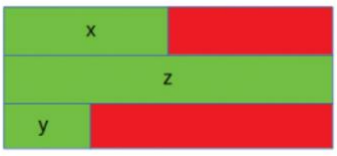
13

int – 4 bytes
double – 8 bytes
short int – 2 bytes

```c
struct A {
  int x;
  double z;
  short int y;
};

printf("Size of struct: %d", sizeof(struct A));
```

Allocated Size = 24 bytes
Required Size = 14 bytes

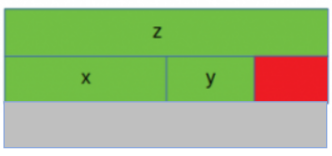


Laying things out differently can have different effects

```c
struct A {
  double z;
  int x;
  short int y;
};

printf("Size of struct: %d", sizeof(struct A));
```

Allocated Size = 16 bytes
Required Size = 14 bytes



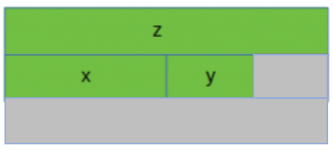
Built-in attribute:

__packed__

```c
struct __attribute__((__packed__)) A {
  double z;
  int x;
  short int y;
};

printf("Size of struct: %d", sizeof(struct A));
```

Allocated Size = 14 bytes
Required Size = 14 bytes



14

Structures in C (6) – Structure Pointers (1)

- Structures are essentially groups of primitives in C (ints, chars, doubles)
  - As each primitive takes up some space, groups of them take up as much space as its elements summed up
  - They are contiguous in memory – i.e. placed one after another
  - We can use pointers to refer to dynamically allocated structures, i.e. using malloc()

15

```c
struct person
{
    int age;
    float weight;
};

struct person * person_ptr;
struct person person1;

int age = 10;
float weight = 35.0;

person_ptr = &person1;

person_ptr->age = age;

person_ptr->weight = weight;

printf("Age: %d", person_ptr->age);
printf("Weight: %f", person_ptr->weight);

```

```c
typedef struct person
{
    int age;
    float weight;
} person_t;

person_t * person_ptr = (person_t *) malloc(sizeof(person_t));

int age = 10;
float weight = 35.0;

person_ptr->age = age;

person_ptr->weight = weight;

printf("Age: %d", person_ptr->age);
printf("Weight: %f", person_ptr->weight);

```

16

Structures in C (7) – Zero-length arrays

- Flexible arrays in C – variable length
- Structure containing regular variables and an array of some type of size 0…???
- Due to C GNU extension, this array can expand at runtime
  - Possible due to dynamic tail padding
- May provide some inspiration for your coursework…

17

```c
typedef struct line {
  int length;
  char contents[0];
} line_t;

line_t * thisline = (line_t *) malloc (sizeof(line_t) + this_length);
thisline->length = this_length;

thisline->contents[5] ????
```

18

Compilation of C to binary

- You may remember this from SCC.110, SCC.150
- Take the C source code, compiles it to assembly for your processor, then assembles into a binary blob that can execute on the OS & said CPU

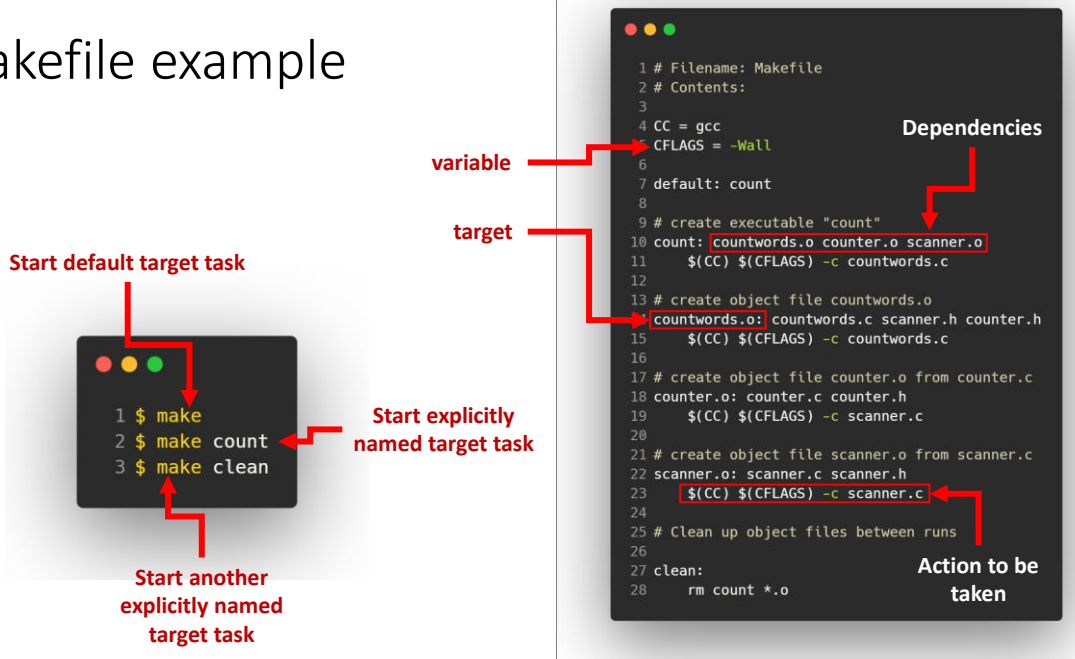
19

Making things less explicit

- Make
  - Automation tool that allows you to specify a build (compilation) procedure
  - You specify a series of actions you want to take:
    - i.e. compile this, compile that, link both, assemble, run, clean-up files
      - Sky is your limit (also the syntax :D)
  - Super useful in any programming task which is repetitive or complex
    - Imagine Linux developers manually typing out >1,000,000 command line statements to compile Linux… impractical to say the least
  - Linux: https://www.gnu.org/software/make/
  - WIN: http://gnuwin32.sourceforge.net/packages/make.htm
  - https://www.norwegiancreations.com/2018/06/makefiles-part-1-a-gentle-introduction/

20

Makefile example



21

Compiler errors and how to read them

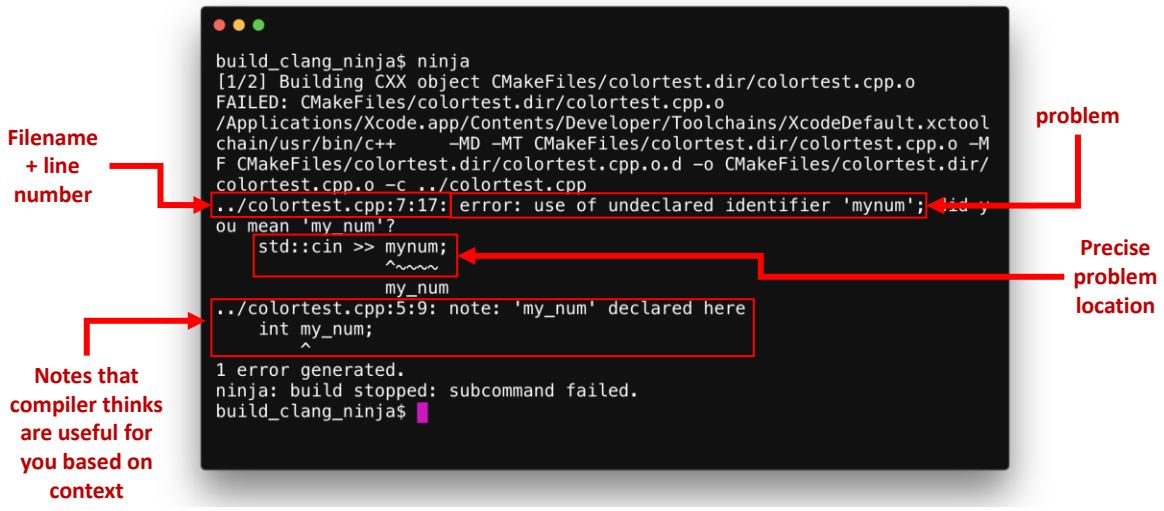
- A compiler is like your moody friend
  - Sometimes talks and talks for hours on end and you still don't get what the point is…
  - Sometimes doesn't say anything and you end up losing your marbles later
- The key? Read between the lines :D

```
build_clang_ninja$ ninja
[1/2] Building CXX object CMakeFiles/colortest.dir/colortest.cpp.o
FAILED: CMakeFiles/colortest.dir/colortest.cpp.o
/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctool
chain/usr/bin/c++     -MD -MT CMakeFiles/colortest.dir/colortest.cpp.o -M
F CMakeFiles/colortest.dir/colortest.cpp.o.d -o CMakeFiles/colortest.dir/
colortest.cpp.o -c ../colortest.cpp
../colortest.cpp:7:17: error: use of undeclared identifier 'mynum'; did y
ou mean 'my_num'?
    std::cin >> mynum;
                ^~~~~
                my_num
../colortest.cpp:5:9: note: 'my_num' declared here
    int my_num;
        ^
1 error generated.
ninja: build stopped: subcommand failed.
build_clang_ninja$
```

```
Segmentation fault (core dumped)
nato@NatoHelionUbuntu:~$
```

22

Dissecting a compiler message – **compile-time error**



23

Error messages are rarely explicit and easy to grasp

- Sometimes you get:

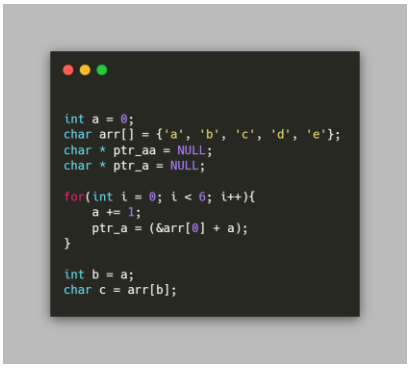
```
Segmentation fault (core dumped)
nato@NatoHelionUbuntu:~$
```

  - Compiler didn't catch the issue – **runtime error**
    - Most likely a logical problem
  - The terminal is telling you that invalid memory region was accessed
    - NULL value in a pointer?
    - Unallocated memory area?
    - Looping over an array of size 100, 500 times? – Accessing garbage data
- What to do?
  - Do not panic.
  - Step through your code and print things out…

24

printf() debugging

- Insert print statements in key program areas until you locate a line that causes a “runtime” error – one not caught by the compiler early



```c
int a = 0;
char arr[] = {'a', 'b', 'c', 'd', 'e'};
char * ptr_aa = NULL;
char * ptr_a = NULL;

for(int i = 0; i < 6; i++){
    a += 1;
    ptr_a = (&arr[0] + a);
}

int b = a;
char c = arr[b];
```

```c
int a = 0;
char arr[] = {'a', 'b', 'c', 'd', 'e'};
char * ptr_aa = NULL;
char * ptr_a = NULL;
printf("Hello world 1");
for(int i = 0; i < 6; i++){
    a += 1;
    printf("Hello world 2");
    ptr_a = (&arr[0] + a);
}
printf("Hello world 3");
int b = a;
char c = arr[b];
```

25

GDB – GNU Debugger
Not for faint-hearted…

- Very powerful debugger for binaries
  - Allows to step through instructions and directly peek into variables and memory addresses
  - Experiment if you feel confident
    - Becomes useful in larger projects

```
── Output/messages ──
0x0000000100000f2a      7           for (i = 0; i < n; i++) {
── Source ──
 2
 3 void fun(int n, char *data[])
 4 {
 5     int i;
 6
 7     for (i = 0; i < n; i++) {
 8         printf("%d: %s\n", i, data[i]);
 9     }
10 }
11
12 int main(int argc, char *argv[])
── Assembly ──
0x0000000100000f18 b0 00          fun+56 mov    $0x0,%al
0x0000000100000f1a e8 4b 00 00 00 fun+58 callq  0x100000f6a
0x0000000100000f1f 89 45 e8       fun+63 mov    %eax,-0x18(%rbp)
0x0000000100000f22 8b 45 ec       fun+66 mov    -0x14(%rbp),%eax
0x0000000100000f25 05 01 00 00 00 fun+69 add    $0x1,%eax
0x0000000100000f2a 89 45 ec       fun+74 mov    %eax,-0x14(%rbp)
0x0000000100000f2d e9 c4 ff ff ff fun+77 jmpq   0x100000ef6 <fun+22>
0x0000000100000f32 48 83 c4 20    fun+82 add    $0x20,%rsp
0x0000000100000f36 5d             fun+86 pop    %rbp
0x0000000100000f37 c3             fun+87 retq
── Threads ──
[1] id 4355 from 0x0000000100000f2a in fun+74 at scrot.c:7
── Stack ──
[0] from 0x0000000100000f2a in fun+74 at scrot.c:7
arg n = 3
arg data = 0x7fff5fbffb60
loc i = 1
[1] from 0x0000000100000f62 in main+34 at scrot.c:14
arg argc = 3
arg argv = 0x7fff5fbffb60
(no locals)
── Registers ──
  rax 0x0000000000000002      rbx 0x0000000000000000      rcx 0x0000010000000203
  rdx 0x0000002000000200      rsi 0x0000000000012068      rdi 0x00007fff79e86110
  rbp 0x00007fff5fbffb20      rsp 0x00007fff5fbffb00       r8 0x0000000000000040
   r9 0x00007fff79e86110      r10 0xffffffffffffffff      r11 0x0000000000000246
  r12 0x0000000000000000      r13 0x0000000000000000      r14 0x0000000000000000
  r15 0x0000000000000000      rip 0x0000000100000f2a   eflags [ TF IF ]
   cs 0x0000002b               ss <unavailable>            ds <unavailable>
   es <unavailable>            fs 0x00000000               gs 0x00000000
── Expressions ──
[1] data[i] = 0x7fff5fbffcf0 "hello"
── Memory ──
0x00007fff5fbffb0c 01 00 00 00 60 fb bf 5f ff 7f 00 00 00 00 00 00 ....`._........
0x00007fff5fbffb1c 03 00 00 00 40 fb bf 5f ff 7f 00 00 62 0f 00 00 ....@._....b...
0x00007fff5fbffb2c 01 00 00 00 60 fb bf 5f ff 7f 00 00 03 00 00 00 ....`._........
0x00007fff5fbffb3c 00 00 00 00 50 fb bf 5f ff 7f 00 00 ad 15 f7 9c ....P._........

0x00007fff5fbffcf0 68 65 6c 6c 6f 00 47 44 42 00 54 45 52 4d 5f 50 hello.GDB.TERM_P
── History ──
$$1 = 63
$$0 = {[0] = 0x7fff5fbffcf0 "hello", [1] = 0x7fff5fbffcf6 "GDB"}
──
>>>
```

26

Other sources:

- Stack Overflow – it's not a shame… *well... :)*
- Google
- Books – linked at the beginning of this guide

...but remember the plagiarism rules

– what you submit must be your own work

27