6.S081: Intro to C

Fall 2021, TA: Cel Skeggs (they/them)

Why use C?

Why we might not:

- C is old and complicated, with subtle behaviors and sharp edges.
- Lots of recent work building OSes in newer languages. Rust, Go, Java, etc.

However:

- Used *everywhere* in OS engineering. Many (not all) real systems are in C.
- Supported *everywhere* on (nearly) every platform.
- Forces you to gain a better understanding of the underlying machine.

SKIPPED IN PRESENTATION

What is different about C? (vs. Python) (1/2)

- C is more or less a “high level assembler”
  - Code constructs map directly to the machine instructions that implement them.
  - Python dictionaries, as a contrasting example, reflect a lot of hidden underlying code.
- C is compiled, not interpreted.
  - Can execute directly on a processor, without any underlying runtime. Very fast.
- C is *statically typed*.
  - In Python, types are associated with the *value* in a variable.
  - In C, types are associated with the *variable*, and *interpret* the raw bytes of the value.
  - Type errors are caught at compile time.
  - Code can execute faster if it doesn’t need to check what the types are.

SKIPPED IN PRESENTATION

What is different about C? (vs. Python) (2/2)

- C uses *manual memory management*, not garbage collection.
  - Explicit "`malloc`" and "`free`" calls. Direct access to memory.
  - This is faster, but much more error-prone.
- Integers and floats in C have specific but indeterminate bounds.
  - Different types have different meanings on different platforms.
  - Certain types have *common* meanings on most *modern* platforms, but not guaranteed.

C vs. Python: types, variables, and values

- In a language like Python, a *value* has a *type*, and a variable can contain any value of any type.

  ```
  x = 10.5
  y = "hello"
  x = y
  ```

- x holds a float at first, but then it holds a string. This works fine in Python.
- Each value is stored in a region of memory, and that region includes information on the *type* of the value.
- **This is not the case with C.**

C vs. Python: types, variables, and values

- In C, values do not store any type information. All type information is stored in variables.

```
int x = 10;
char *y = “hello, world!”;
```

- A value of a type only has that type because it is stored in a variable of that type. Each variable is backed by a *memory region* that is large enough for the value.
- This type information only exists when the program is being compiled. It does not exist anymore when the program is running.

Primitive Integer Types in C

Type	Size on 64-bit RISC-V	Signed min/max on 64-bit RISC-V	Unsigned min/max on 64-bit RISC-V	*Guaranteed minimum across platforms*
[signed/unsigned] char	1 byte integer	-128 to +127	0 to 255	*8 bits, -127 to +127*
[unsigned] short	2 byte integer	-32768 to +32767	0 to 65535	*16 bits, -32767 to +32767*
[unsigned] int	4 byte integer	-2^31 to 2^31 - 1	0 to 2^32 - 1	*16 bits, -32767 to +32767*
[unsigned] long	8 byte integer	-2^63 to 2^63 - 1	0 to 2^64 - 1	*32 bits, -(2^31-1) to 2^31-1*
[unsigned] long long	8 byte integer	-2^63 to 2^63 - 1	0 to 2^64 - 1	*64 bits, -(2^63-1) to 2^63-1*

You don’t need to concern yourself with the guaranteed minimums in this class… just keep it in mind when you move to writing C for another platform.

Primitive Floating-Point Types in C

Type	Size on RISC-V	Format on RISC-V	*Guarantees in general*
float	4 byte floating-point	32-bit IEEE 754-2008	*None*
double	8 byte floating-point	64-bit IEEE 754-2008	*At least as long as float*
long double	16 byte floating-point	128-bit IEEE 754-2008	*At least as long as double*

Defining primitive variables in C

```
int value_1;
int value_2 = 83;
float value_3 = 125.0;
char value_4, value_5 = 3, value_6 = 0xFF;
```

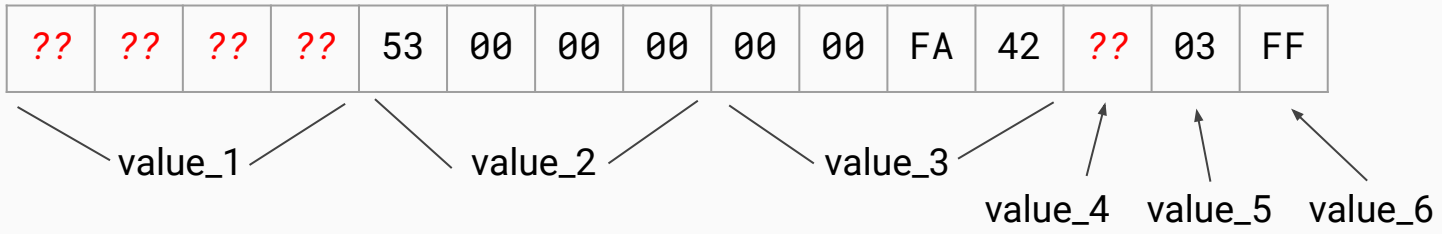
Three parts to a variable definition: *type*, *name*, and (optionally) *initializer*.

Can define multiple variables at once, but this can get confusing.

If you don't initialize a variable, it *might* default to zero, but you can't rely on that… you *must* initialize it before use! (Except static/global variables.)

Memory for primitive variables

```
int value_1;
int value_2 = 83;
float value_3 = 125.0;
char value_4, value_5 = 3, value_6 = 0xFF;
```



(One possible layout of these variables in memory… not guaranteed!)

Endianness

How do we write a number like 0x12345678 (= 305419896) in memory?

0x12, 0x34, 0x56, 0x78 *(big-endian)* - or - 0x78, 0x56, 0x34, 0x12 *(little-endian)*

It depends on the platform! This is a long-standing technical debate. Our particular RISC-V platform is **little-endian**, so 6.S081 is as well.

Mnemonic: *big*-endian means you start with the *big* end (the MSB, or “most significant byte”).

Bonus: read IEN 137 (1980) to understand the source of the terminology (Gulliver’s Travels).

Types of memory in C

- **Stack Memory**
  - Local variables allocated within functions. This memory is destroyed and may be reused after a function exits.
  - Not initialized by default. Will reflect whatever happened to be in that piece of memory.
- **Static Memory**
  - Variables declared outside any function, and variables declared with "`static`".
  - A single copy is stored, at a predefined and unchanging memory address.
  - Initialized to zero by default.
- **Heap Memory**
  - Explicitly allocated (`malloc`) and freed (`free`).
  - After being freed, memory may be reused (in whole or in part) for future allocations.
  - Not initialized by default. Will reflect whatever happened to be in that piece of memory.

Key topic in C: **memory safety**

There are many ways that running C programs can get corrupted. These can result in mysterious and baffling bugs. Some examples:

- **Use-after-free**: if a program frees a region of memory, but keeps using it.
- **Double-free**: if a program frees a region of memory twice instead of once.
- **Uninitialized memory**: if a program uses memory that was never initialized.
- **Buffer overflow**: if a program modifies memory beyond the end of a region.
- **Memory leak**: if a program allocates memory, but never frees it.
- **Type confusion**: if a program unintentionally uses the wrong data type to access a variable in memory.

Key topic in C: **pointers**

We represent regions of memory in C using *pointers*:

```
int value_1 = 6828;
int *pointer_to_value_1 = &value_1;
*pointer_to_value_1 = 6081;
printf("%d\n", value_1); // prints 6081, not 6828!
```

Pointers are *references* that describe the location of an underlying piece of memory. (The *start* of the memory… but *not* the size.)

Pointers are integers in disguise

```
int global_variable   = 0;
void test_function(void) {
  int local_variable  = 0;
  int *heap_reference = malloc(sizeof(int));
  printf("Global: 0x%x\n", &global_variable);
  printf(" Local: 0x%x\n",  &local_variable);
  printf("  Heap: 0x%x\n",   heap_reference);
}
```

```
$ test
Global: 0x8C8
 Local: 0x2FAC
  Heap: 0x12FF0
```

Pointers are integers that specify the *address* where a region of memory starts, and the *type* of the value expected to be found there.

All different sorts of pointers

```
int    *a;          // pointer to int
float  *b;          // pointer to float
int   **c;          // pointer to pointer to int
char  (*d)(int);    // pointer to a function (int -> char)
char (**e)(int);    // pointer to pointer to function (int -> char)
void   *f;          // pointer to untyped memory
void  **g;          // pointer to pointer to untyped memory
```

Pointers can be arbitrarily nested:

```
int ******value;    // PLEASE don't actually ever do this.
```

Pointer Example

```
int a = 10;
int *b = &a;
int c = a;
int d = a;
*b = 20;
d = 30;

// what does this print?
printf(“a = %d, *b = %d, c = %d, d = %d\n”, a, *b, c, d);
```

Pointer Example

```
int a = 10;
int *b = &a;
int c = a;
int d = a;
*b = 20;
d = 30;

// what does this print?
printf(“a = %d, *b = %d, c = %d, d = %d\n”, a, *b, c, d);
```

Answer: a = 20, *b = 20, c = 10, d = 30

Another data type: arrays

```
int fibonacci_numbers[] = {1, 1, 2, 3, 5, 8, 13, 21};
int even_numbers[6] = {0, 2, 4, 6, 8, 10};
int uninitialized_array[6];
int my_array[6] = {1};

my_array[3] = 100;
my_array[4] = 200;
printf(“my_array: [0] = %d, [3] = %d, [5] = %d\n”,
       my_array[0], my_array[3], my_array[5]);
```

Another data type: arrays

```
int fibonacci_numbers[] = {1, 1, 2, 3, 5, 8, 13, 21};
int even_numbers[6] = {0, 2, 4, 6, 8, 10};
int uninitialized_array[6];
int my_array[6] = {1};

my_array[3] = 100;
my_array[4] = 200;
printf("my_array: [0] = %d, [3] = %d, [5] = %d\n",
       my_array[0], my_array[3], my_array[5]);
```

Answers: [0] = 1, [3] = 100, [5] = 0. (Yes, the rest of the elements of a partially-initialized array are also initialized!)

Notes on arrays

- Arrays are **not** lists. They have a fixed length, and are not resizable!
- Array elements are laid out *sequentially* in memory:

```
int my_array[4];
printf("Locations: %x %x %x %x\n", &my_array[0],
       &my_array[1], &my_array[2], &my_array[3]);
// Prints: Locations: 2FB0 2FB4 2FB8 2FBC
```

- Note that arrays are 0-indexed, not 1-indexed. And unlike python, you cannot use negative indices to count backwards from the end.

Pointer arithmetic

- Dereferencing the *n*th element of an array (`array[n]`) is the same as dereferencing the memory at *n* plus the array pointer (`*(array + n)`)
- Taking a reference to the *n*th element of an array (`&array[n]`) is the same as *adding n* to the array pointer (`array + n`).
- **But wait… &my_array[3] was `0x2FBC`, which is 12 more than `0x2FB0`!**
- This is because pointer arithmetic *multiplies by* the size of the underlying data type!

```
(long) (my_array + 3) == ((long) my_array) + 3 * sizeof(int)
```

A side effect of pointer arithmetic

What does this print?

```
int values[5] = {10, 20, 30, 40, 50};
printf("%d\n", 4[values]);
```

(Consider… if x[y] == x + y, and we know that addition is commutative…)

Answer: 50! x[y] = *(x+y) = *(y+x) = y[x]

Size of arrays

```
int a[10];
printf(“sizeof(a)=%d, sizeof(a[0])=%d, num_elems=%d\n”,
       sizeof(a), sizeof(a[0]), sizeof(a)/sizeof(a[0]));

// Output: sizeof(a)=40, sizeof(a[0])=4, num_elems=10
```

Casting between primitive types

```
float a = 100; // assigning an int to a float works fine!
int b = a;     // but assigning a float to an int doesn't!
// To fix this problem... we need to cast!
float c = 701.7;
int d = (int) c; // receives the value 701! (truncated.)
```

Casting between pointers and integers

```
// Casting from a pointer to an integer
int x[4] = {1, 2, 3, 4};
int *x_ptr = &x[0];
long x_address = (long) x_ptr;

// Casting from an integer to a pointer
int *x2_ptr = (int *) (x_address + 4);
int x2_value = x2_ptr[1];

// What value does x2_value have?
```

Casting between pointers and integers

```
// Casting from a pointer to an integer
int x[4] = {1, 2, 3, 4};
int *x_ptr = &x[0];
long x_address = (long) x_ptr;

// Casting from an integer to a pointer
int *x2_ptr = (int *) (x_address + 4);
int x2_value = x2_ptr[1];

// What value does x2_value have?
```

Answer: x2_value is 3!

Casting: a source of memory unsafety

```
long x = 0xDEADBEEF;
int *x2 = (int *) x;
*x2 = 12345;
printf("Done!\n");
```

What happens?

```
$ test
usertrap(): unexpected scause 0x000000000000000f pid=3
            sepc=0x0000000000000016 stval=0x00000000deadbeef
$
```

Different platforms will give different errors. On Linux, you might get a Segmentation Fault. This is the error message from xv6.

What size are pointers?

- Depends on the specific platform.
- Our flavor of RISC-V uses 64-bit (8 byte) pointers, which makes them the same size as a long.
- (Frequently, but not always, pointers are the same size as a long.)
- This means, in 6.S081, we can cast a pointer to a *long*, but not to an *int*. Variables of type ‘int’ are too small for us to fit a complete pointer.

Functions in C

```
double add_numbers(double *numbers, int count) {
                    // ^ numbers and count are on the stack
  double result = 0; // <- result is also on the stack
  for (int i = 0; i < count; i++) { // <- so is 'i'
    result += numbers[i];
  }
  // after we return, all the stack variables go away!
  return result;
}
```

An unusual data type: **void**

- Represents the lack of a data type.
- Mostly useful in return types and parameters of functions:

```
void test_function(void);
```

- You can't define a variable of type `void`, because… what would that mean?
- But you *can* define a variable of type `void *`. You just can't dereference it.
- (You aren't *allowed* to use arithmetic on `void *` pointers officially, but we use GCC, which supports it as an extension to C.)

Oops

What's wrong with this code?

```
int *multiples_of(int number, int max) {
    int my_local_array[max];
    for (int i = 0; i < max; i++) {
        my_local_array[i] = number * (i + 1);
    }
    return &my_local_array[0];
}
```

Oops

What's wrong with this code?

```
int *multiples_of(int number, int max) {
    int my_local_array[max];
    for (int i = 0; i < max; i++) {
        my_local_array[i] = number * (i + 1);
    }
    return &my_local_array[0];
}
```

Answer: my_local_array will cease to exist when this function returns, so the pointer returned will be invalid, and is likely to be promptly overwritten by garbage data! And it would be even worse if it were changed!

Oops (fixed)

Solution:

```
int *multiples_of(int number, int max) {
    int *my_local_array = malloc(sizeof(int) * max);
    for (int i = 0; i < max; i++) {
        my_local_array[i] = number * (i + 1);
    }
    return &my_local_array[0];
}
// The caller will need to free() the returned pointer.
```

Definitions versus declarations

```
extern void function_1(void); // declares a function
extern int some_variable;     // declares a variable

void function_1(void);   // also declares a function
int some_variable;       // but this defines a variable!

void function_2(void) {  // this defines a function!
    // ...
}
```

Definitions versus declarations

- You must *declare* a variable or function in each file before it can be *used*, because C needs to know its type or type signature.
  - You can declare a variable or function as many times as you want, as long as it always has the same type or type signature.
- You must *define* each variable or function exactly once in your codebase.
  - A definition also counts as a declaration, but (of course) only *after* the point where the definition happens.
  - You can define a function or variable in one file, but use it in another.
- We generally put many of our *declarations* in separate “header files” so that each part of the program knows the important types for the other parts.
  - Statements like **`#include “kernel/types.h”`** are used to incorporate header files.

An example of needing a declaration

```
void function_1(void) {
    function_2(100);        // this DOES NOT WORK!
}

void function_2(int xyz) {
    printf("%d\n", xyz);
}
```

C proceeds top to bottom. If it hasn't seen a declaration yet, it won't know the correct types to use. How would it know that function_2 takes an integer, as opposed to (for example) a float?

Fixing the last example

```
void function_2(int xyz);

void function_1(void) {
    function_2(100);        // this works now!
}

void function_2(int xyz) {
    printf(“%d\n”, xyz);
}
```

Declaring **static** functions and variables

- If we name a function the same thing in two separate files, they will conflict! C will have trouble distinguishing them.
- To avoid conflicts, we can declare our variables and values as **static**:

```
static void function_2(int xyz);

static void function_2(int xyz) {
    printf("%d\n", xyz);
}
```

- This function is *only* accessible from the file it's defined in!

Declaring local variables as static

- While local variables within a function are *normally* allocated on the stack, we can specify that they be allocated in static memory instead:

```
int add_cumulative_numbers(int increase) {
    static int total_sum = 0;
    total_sum += increase;
    return total_sum;
}
```

- total_sum will be initialized to zero at program start, and it will keep its value across calls to add_cumulative_numbers! It won't be reinitialized.

Function pointers

```
static void my_function_1(int);
static void my_function_2(int);

void pointer_example(int variant) {
    void (*local)(int);
    if (variant == 1) {
        local = my_function_1;
    } else {
        local = my_function_2;
    }
    local(100); // call function via variable
}
```

Strings in C

- Strings are just arrays of characters.

```
char *my_string = “6.S081!”; // use " for string literals
printf(“string=\”%s\”, third_char=’%c’\n”,
       my_string, my_string[2]);
// Prints: string=”6.S081!”, third_char=’S’
```

Characters in C

- Characters in C are just integers that are (in our case) 1 byte long.
  - The mapping between numbers and letters is defined by ASCII, which you can read about on Wikipedia: https://en.wikipedia.org/wiki/ASCII

```
char c1 = ‘a’;    // use ' for character literals
char c2 = c1 + 1;
printf(“%c=%d, %c=%d\n”, c1, c1, c2, c2);
// Prints: a=97, b=98
```

- Sometimes ‘char’ is used to mean a character in a string… and sometimes it’s used just to mean a single byte. C doesn’t distinguish the two.

Lengths of strings

- The last character in any C string is ‘\0’ (= 0x00), and this is used so that C knows the length of the string.
- Make sure to leave room for this terminator when allocating a string!
- This means you can compute the length of a string as follows:

```
int strlen(const char *str) {
  int i;
  for (i = 0; str[i] != 0; i++) {}
  return i;
}
```

- (An implementation of strlen is provided for you in xv6.)

Type definitions

- Not satisfied with the built-in types? You can define your own!

```
// from kernel/types.h:
typedef unsigned char  uint8;
typedef unsigned short uint16;
typedef unsigned int   uint32;
typedef unsigned long  uint64;
```

- If we ever port xv6 to another platform where the sizes don't match up like this, we can just change the one file that defines these typedefs, and the rest of the code will update to match!

Header files (.h) and source files (.c)

In C, we separate out header files (that contain shared declarations) from source files (that contain definitions of different pieces of code).

kernel/spinlock.h: declarations describing the spinlock interface
kernel/spinlock.c: actual definitions of spinlock code

To include a .h file in a .c file:

```
#include “spinlock.h”
```

If the header file is in a different directory, you may need a longer path, such as “kernel/spinlock.h”. **Also, don’t include .c files! It will cause problems.**

The C preprocessor

Lines in C that start with '#' are *preprocessor* directives.

Some basic examples:

```
#include "spinlock.h"    -> Incorporate the contents of spinlock.h here
#define NPROC 64         -> Replace all instances of 'NPROC' with '64'
#define TWICE(x) ((x)*2)
    -> Replace all instance of 'TWICE(x)' with '((x) * 2)' … for any expression 'x'.
```

The preprocessor executes entirely before any actual compilation is run.
We don't use it much in xv6, but worth understanding eventually.

SKIPPED IN PRESENTATION

More preprocessor directives

```
#ifdef DEBUG  // only if DEBUG was defined by #define
    printf(“some debug message: %d”, my_value);
#else
    // do something else instead of printing
#endif
```

Besides, #ifdef, also available:

`#ifndef VAR` if VAR *wasn’t* defined by #define

`#if EXPR` if EXPR evaluates to true

Can also undefine macros defined with #define VAR using #undef VAR

SKIPPED IN PRESENTATION

Include guards

If a .h file is included twice, its contents will be pasted twice. Sometimes this is not preferable… in this case, an *include guard* is generally used:

```
// at the start of the something.h file
#ifndef SOMETHING_H
#define SOMETHING_H
// ... the normal contents go here ...
#endif /* SOMETHING_H */
```

(In practice, xv6 doesn't really use this feature.)

SKIPPED IN PRESENTATION

Syntax for comments

```
// This is a comment, which lasts for the rest of the line

/* This is also a comment,
    but it can stretch multiple lines,
    and won't stop until the terminator: */
```

Common code constructs: if

```
if (X) {
    /* run this code if X is nonzero, aka true */
} else if (Y) {
    /* if X isn't true, but Y is true, run this code */
} else {
    /* if neither X nor Y is true, run this instead */
}
```

Common code constructs: while

```
while (COND) {
    // if COND is true, run the code here.
    // when this code finishes, check COND again.
}
// we only stop when COND is false at the end of the loop
```

Common code constructs: do while

```
do {
    // run this code at least once
    // but we'll check COND at the end of the loop
    // and if it's true, we'll do this again
} while (COND);
// we only stop when COND is false at the end of the loop
```

Common code constructs: for

```
for (A; B; C) {
    // BODY
}
```

This is equivalent to:

```
A;
while (B) {
    // BODY
    C;
}
```

Example of a for loop

```
void print_even_numbers(int max) {
    for (int i = 0; i < max; i = i + 2) {
        printf(“%d\n”, i);
    }
}
```

Some convenient syntax you can use:

```
i += 2     -> This is equivalent to i = i + 2
i++        -> This is equivalent to i = i + 1
```

Common memory functions you might use

- **malloc(n)**: allocates a region of n bytes from heap memory, and returns a pointer to the start of it. If there's no memory left to allocate, returns NULL.
- **free(ptr)**: frees the region of memory starting at ptr that was previously allocated by malloc. If ptr is NULL, does nothing.
- **memset(ptr, v, n)**: sets every byte from ptr[0] to ptr[n-1] to v.
- **memmove(dst, src, n)**: copies src[0]...src[n-1] to dst[0]...dst[n-1]
- **memcpy(dst, src, n)**: alternate faster version of memmove, which may misbehave if dst and src overlap in any way. (Discouraged! Prefer memmove.)

Common string functions you might use

- **strlen(str)**: computes and returns the length of str, based on finding its null terminator. Will misbehave if the null terminator is missing!
- **strcmp(a, b)**: compares two strings a and b, and returns an integer < 0, == 0, or > 0, depending on whether a < b, a == b, or a > b.
- **strcpy(dst, src)**: equivalent to **memcpy(dst, src, strlen(src)+1)**;

Always remember the null terminators!

More data types: structures

- A structure lets you declare a package of values of different types as a single combined value.

```
struct xy_point {
    double x;
    double y;
};
struct xy_point my_point = { 12.5, -6.2 };
my_point.x = 50.6;              // modify field ‘x’
printf(“%f\n”, my_point.y);     // read field ‘y’
```

Fancy structures

You can also initialize structure fields by their names, instead of their order within the structure:

```
struct xy_point my_point = {
    .y = -6.2,
    .x = 12.5,
};
```

You can also give structures type names that don’t require the word ‘struct’:

```
typedef struct xy_point xy_point_t;
xy_point_t my_point;
```

Like structures, but worse: unions

- A structure is like a union, except that every field is stored at the same memory address:

```
union my_union {
    float x;
    int y;
}
```

- This means that you can only safely use a single field of a union at a time… and you'll have to have some other way to track which one is safe to use.
- You won't need these much. You can look up the details if you need them.

Logical operators

(Remember: true is anything except 0, false is 0.)

```
int boolean_1 = 1, boolean_2 = 0;

(boolean_1 && boolean_2) == 0     // logical AND
(boolean_1 || boolean_2) == 1     // logical OR
!boolean_2 == 1                   // logical NOT
```

Bitwise operators

```
unsigned short a = 0x1313, b = 0x3232;

(a & b) == 0x1212          // bitwise AND
(a | b) == 0x3333          // bitwise OR
(a ^ b) == 0x2121          // bitwise XOR
~a      == 0xECEC          // bitwise NOT
```

Next… a challenge about pointers!

SKIPPED IN PRESENTATION

Applied bitwise operators

```
unsigned int my_int;
// Set the Nth bit of an integer:
my_int |= 1 << N;
// Clear the Nth bit of an integer
my_int &= ~(1 << N);
// Check if any bits in MASK are set
if (my_int & MASK) { /* ... */ }
// Check if all bits in MASK are set
if ((my_int & MASK) == MASK) { /* ... */ }
// Check if integer is a power of two
if (my_int && !(my_int & (my_int - 1))) { /* ... */ }
```

Finally: a pointer challenge

Source: [see next slide]

```
int main() {
  int x[5];                  // x is at 0x7fffdfbf7f00
  printf("%p\n", x);
  printf("%p\n", x+1);
  printf("%p\n", &x);
  printf("%p\n", &x+1);
  return 0;
}
```

Finally: a pointer challenge: SOLUTION

Source: [The Ksplice Pointer Challenge - Oracle Linux Blog]

```
int main() {
  int x[5];                 // x is at 0x7fffdfbf7f00
  printf("%p\n", x);        // -> 0x7fffdfbf7f00
  printf("%p\n", x+1);      // -> 0x7fffdfbf7f04
  printf("%p\n", &x);       // -> 0x7fffdfbf7f00
  printf("%p\n", &x+1);     // -> 0x7fffdfbf7f14
  return 0;
}
```

Questions?