Practical C

Learn by ~~Doing~~ Debugging

Tyler
deepcode.solutions
April 15, 2024

Outline

Expectations

The only way to get better at writing code is to write code.

An expert C programmer knows the theory and how to apply it.

Mastery often takes years, if not decades.

Most of an expert’s learning is self-study.

We will bootstrap your self-study by covering the fundamentals of C.

History of C

Dennis Ritchie created C around 1972; the language is now 52 years old.

Developed on the PDP-11.

Bootstrap compiler written in assembly.

Used to develop operating systems, then systems programs.

Standards: K&R C (1978), ANSI C (1989), C99, C11, C17, C23.

History of C

Dennis Ritchie (standing), Ken Thompson.

PDP-11/20:

- 32 KB - 4 MB RAM
- No clock, everything tuned to bus speed
- Floppy disk, platter hard disk, tape

Teletype 33 terminal: paper output.

See a similar computer in action here.

source

History of C

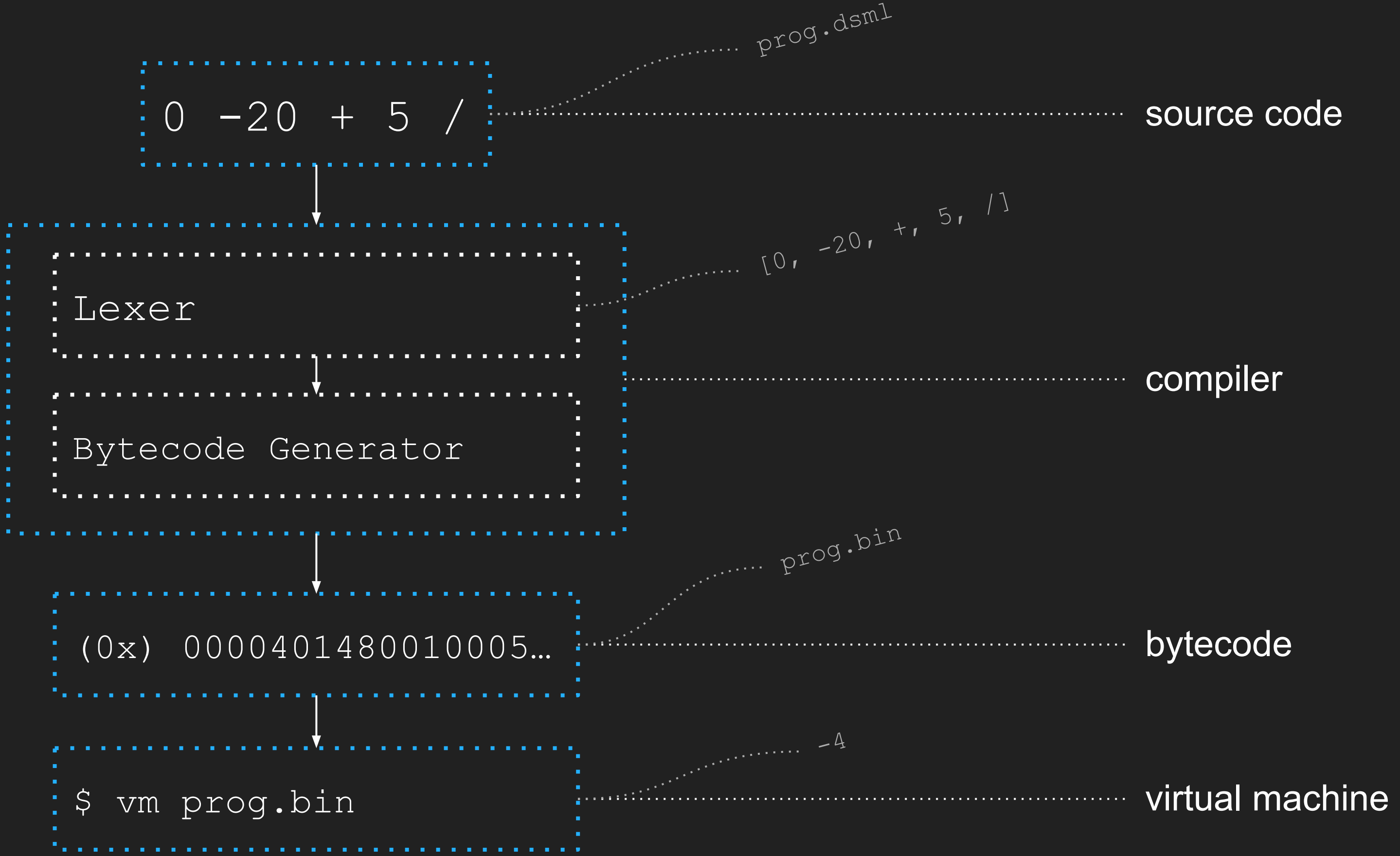
Probably used something like a VT05 when making C.

source

Project Introduction

We’ll make a calculator, but not just any calculator.

Project Introduction



Project Introduction

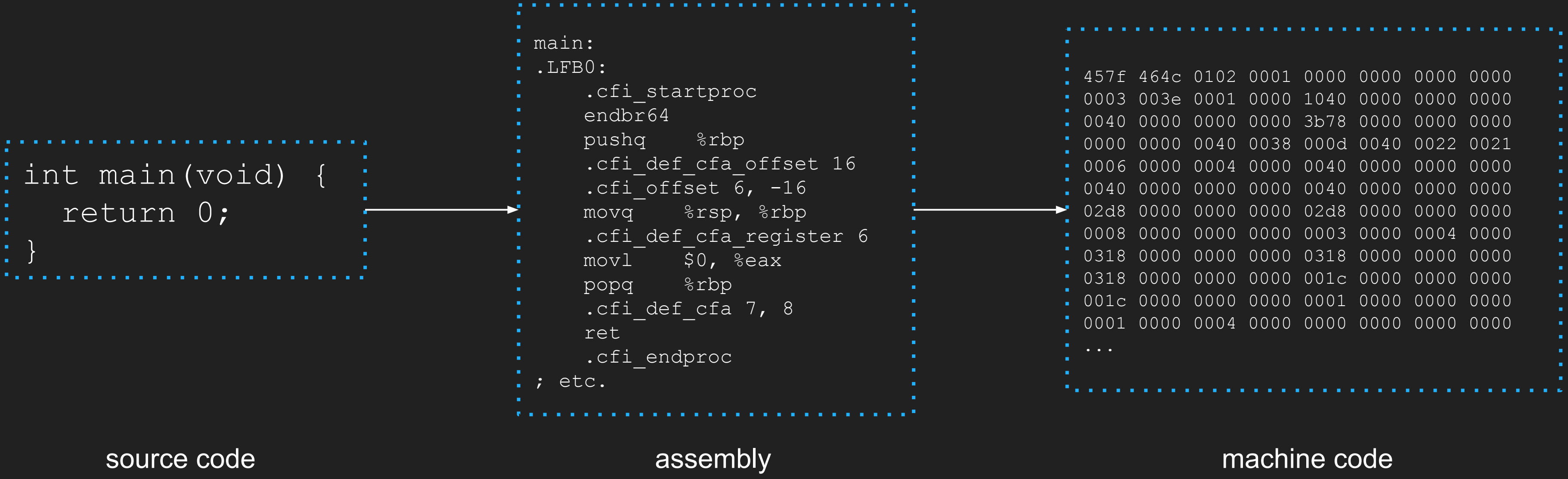
So we’ll actually make a virtual machine and toolchain.

C Language Primer

C Language Primer – Initial Tooling

C is a compiled language, meaning it requires a tool called a compiler to transform source code into machine code.

Popular compiler choices are gcc and clang.



```c
int main(void) {
   return 0;
}
```

```
main:
.LFB0:
    .cfi_startproc
    endbr64
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    movl    $0, %eax
    popq    %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
; etc.
```

```
457f 464c 0102 0001 0000 0000 0000 0000
0003 003e 0001 0000 1040 0000 0000 0000
0040 0000 0000 0000 3b78 0000 0000 0000
0000 0000 0040 0038 000d 0040 0022 0021
0006 0000 0004 0000 0040 0000 0000 0000
0040 0000 0000 0000 0040 0000 0000 0000
02d8 0000 0000 0000 02d8 0000 0000 0000
0008 0000 0000 0000 0003 0000 0004 0000
0318 0000 0000 0000 0318 0000 0000 0000
0318 0000 0000 0000 001c 0000 0000 0000
001c 0000 0000 0000 0001 0000 0000 0000
0001 0000 0004 0000 0000 0000 0000 0000
...
```

C Language Primer – Initial Tooling

Q Why would a compiler convert source code to assembly instead of directly to machine code?

C Language Primer – `main`

The C standard declares a special function called `main` which is the designated entrypoint to your code (in most cases).

There are multiple possible signatures. We will focus on one for now.

```
int main(void);
```

C Language Primer – `main`

Let’s write your first C program.

```c
int main(void) {
  return 0;
}
```

```
$ <editor> main.c
$ clang main.c
$ ./a.out
$ echo $?
```

C Language Primer – The Preprocessor

A typical hello-world program might be:

```
#include <stdio.h>

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

Compile and run this program.

C Language Primer – The Preprocessor

Q Remove the `#include <stdio.h>` line and recompile. What happens?

C Language Primer – The Preprocessor

`#include` is replaced with the contents of `stdio.h` as if you had typed the content into `main.c` yourself.

The mechanism powering `#include` is called the preprocessor.

The preprocessor has more features such as support for macros, but we won't cover those in this class.

C compilation is a multi-phase process; the preprocessor is run before typical compilation (translation) runs.

C Language Primer – Data Types

C has a variety of data types. We will only cover the subset required in our project.

We will start with numbers, booleans, and arrays.

C Language Primer – Data Types – Numbers

We will focus on the integer types.

C provides multiple integer types that differ in their numeric ranges.

For example, `unsigned char` is guaranteed to be one byte on your system.

C Language Primer – Data Types – Numbers

Q If a byte is 8 bits on your system, what is `unsigned char`'s maximum value?

C Language Primer – Data Types – Numbers

The “unsigned” in `unsigned char` means the type represents positive integers and 0, so the minimum value is 0.

Another type is `unsigned int`. If that type is 4 bytes on your system, its maximum value is much larger than `unsigned char`, but it occupies 4 times as much storage space.

C Language Primer – Data Types – Numbers

Signed versions of integer types also exist.

Signed versions occupy the same number of bytes as their unsigned counterparts, but their range is different.

The most significant bit is used to store the sign.

Signed versions use the `signed` qualifier instead of `unsigned`.

`unsigned char`

$$\underbrace{\texttt{1111 1111}}_{\text{magnitude bits}}$$

`signed char`

$$\underset{\text{sign bit}\nearrow}{\texttt{1}}\underbrace{\texttt{111 1111}}_{\text{magnitude bits}}$$

C Language Primer – Data Types – Numbers

Q If a `signed char` is 8 bits on your system, what is its maximum value?

C Language Primer – Data Types – Numbers

Q If a `signed char` is 8 bits on your system, what is its minimum value?

C Language Primer – Data Types – Numbers

The reason the magnitudes of the maximum and minimum values differ is due to the way negative numbers are encoded. We won’t go into detail here, but for more information, look into two’s complement.

Understanding two’s complement is essential when working with binary representations of numbers in C.

C Language Primer – Data Types – Numbers

Q Given the following binary and decimal values for a signed char,

`0b 0000 0010` – 2
`0b 0000 0001` – 1
`0b 0000 0000` – 0

what is the decimal value of `0b 1000 0000`?

C Language Primer – Data Types – Numbers

The integer types that will be most helpful in our project are `char`, `short`, and `int`.

See here for documentation on all of the integer types available in C.

C Language Primer – Data Types – Numbers

As suggested earlier, the number of bits in a byte and the number of bytes in an integer type are not guaranteed to be the same across platforms.

If your code depends on a specific number of bits or bytes, you may run into portability issues.

C99 introduced fixed-width integer types. These guarantee the number of bits in an integer type. These types are available in the `stdint.h` header file.

Examples: `int8_t`, `uint8_t`, `int16_t`, `uint16_t`, `int32_t`, etc.

C Language Primer – Data Types – Numbers

Signed numeric types can overflow or underflow *without warning*. Overflow and underflow is undefined behavior. The same behavior on unsigned types is called "wraparound" and is well-defined.

```c
#include <stdint.h>

int main(void) {
  int8_t i = 127;
  i = i + 1;
  return 0;
}
```

enable "all" warnings

```
$ gcc -Wall main.c
$
```

```rust
fn main() {
  let mut i: i8 = 127;
  i = i + 1;
}
```

```
$ rustc main.rs
error: this arithmetic operation will overflow
 --> main.rs:3:9
  |
3 |   i = i + 1;
  |       ^^^^^ attempt to compute
  |             `i8::MAX + 1_i8`, which would
  |             overflow
```

C Language Primer – Booleans

Historically C had no first-class boolean type. Developers would typically use 0 as `false` and some non-zero value (usually 1) as `true`.

C99 introduced a boolean support library which exposes `true`, `false`, and `bool` for use in your program.

```
#include <stdbool.h>

int main(void) {
  bool a = true;
  bool b = false;
  bool c = 1 < 2; // true

  return 0;
}
```

C Language Primer – Arrays

Arrays are C’s built-in type for collections.

Arrays can only hold elements of a single type.

Arrays are contiguous, meaning that all elements of a single array exist side-by-side in memory.

C Language Primer – Arrays

```c
#include <stdbool.h>

int main(void) {
  // Storage for 5 integer elements.
  int arr1[5];

  // Storage for 3 uint8_t elements, initialized.
  uint8_t arr2[3] = {0, 70, 21};

  // Storage for 3 boolean elements, but the size is set by the initializer.
  bool arr3[] = {true, false, true};

  return 0;
}
```

C Language Primer – Arrays

Accessing individual array elements is done using the “array subscript operator,” `[]`. This operator takes a single non-negative value and uses that as an index into the array.

```
int main(void) {
  int a[] = {0, 1, 2, 3, 4};

  int first_element = a[0]; // value: 0

  int third_element = a[2]; // value: 2

  // Change the value of the 4th element.
  a[3] = 5;

  return 0;
}
```

C Language Primer – Arrays

Q What happens if you use a negative index or an index larger than the bounds of the array?

```c
int main(void) {
  int a[] = {0, 1, 2, 3, 4};

  int first_element = a[0]; // value: 0

  int third_element = a[2]; // value: 2

  // Change the value of the 4th element.
  a[3] = 5;

  return 0;
}
```

C Language Primer – Printing

C provides various ways to print information to the console. We will use `printf`.

```
#include <stdbool.h>

int main(void) {
  printf("foo\n");
  return 0;
}
```

This prints “foo” to standard output (`stdout`).

It also prints a newline (`\n`).

`\n` is an escape sequence, a way to encode special characters in the string literal.

C Language Primer – Printing

The f in `printf` stands for “formatted.” The string we provide to printf is the format string.

Format strings feature format specifiers which control how data is printed.

```
#include <stdio.h>

int main(void) {
  int i = 2;
  char ch = 65;
  printf("i is %d, c is %c\n", i, ch);
  return 0;
}
```

C Language Primer – Printing

Q What does the following program print?

```c
#include <stdio.h>

int main(void) {
  int i = 2;
  char ch = 65;
  printf("i is %d, c is %c\n", i, ch);
  return 0;
}
```

C Language Primer – Printing

The order of arguments following the format string must match the order of format specifiers.

If you provide more arguments to print than format specifiers, the arguments that have no corresponding specifier are ignored.

If you provide more specifiers than arguments following the format string, the program behavior is undefined.

If the type of an argument to print does not match the expectations of the format specifier controlling the printing, the behavior may be undefined.

C Language Primer – Printing

Q Is printing an `int32_t` using the `%d` format specifier portable? Why or why not?

C Language Primer – Functions

We've already seen examples of functions: `main` and `printf`.

Here's a new, custom function.

```
int double_number(int a) {
  int n = a * 2;
  return n;
}

int main(void) {
  int number = 5;
  int doubled = double_number(number);
  return 0;
}
```

C Language Primer – Functions

Functions have signatures and implementations.

`void` as a parameter means the function has no parameters.

```
int double_number(int a) {
  int n = a * 2;
  return n;
}

int main(void) {
  int number = 5;
  int doubled = double_number(number);
  return 0;
}
```

C Language Primer – Functions

Q What happens if we move `double_number` below `main`?

```
int double_number(int a) {
  int n = a * 2;
  return n;
}

int main(void) {
  int number = 5;
  int doubled = double_number(number);
  return 0;
}
```

C Language Primer – Functions

We can place just the signature above `main` to let the compiler know that we will implement `double_number` elsewhere.

This is called a forward declaration.

```c
int double_number(int);

int main(void) {
  int number = 5;
  int doubled = double_number(number);
  return 0;
}

int double_number(int a) {
  int n = a * 2;
  return n;
}
```

C Language Primer – Functions

Q What will the following program print?

```
#include <stdio.h>

void add_one(int a) {
  a = a + 1;
}

int main(void) {
  int a = 0;
  add_one(a);
  printf("%d\n", a);
  return 0;
}
```

C Language Primer – Functions – Calling Convention

C is a “pass by value” (or “call by value”) language.

The called function gets copies of the caller’s arguments.

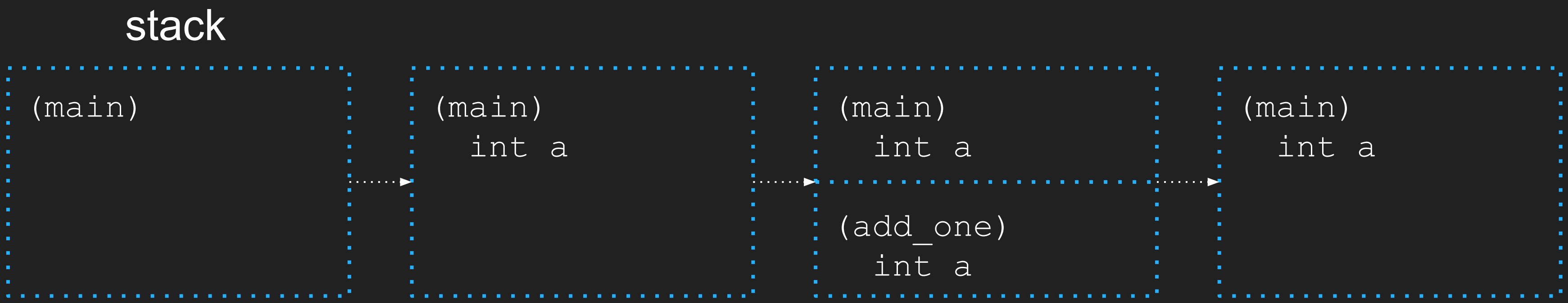
The caller’s arguments are preserved in the caller’s stack frame, and the called function’s copies are stored in the called function’s stack frame.

The stack is a data structure in memory that stores scopes’ local variables.

C Language Primer – Functions – The Stack

Let's study the evolution of the stack for a subset of code we recently saw.

```c
int main(void) {
  int a = 0;
  add_one(a);
  // printf(...); etc.
```



Set up space for `main`, `main` is called.

Reserve space for `main`'s variable `a`.

Set up space for `add_one` and its parameter `a`. Call `add_one`.

`add_one` returns, and its stack is popped. We're back in `main`.

C Language Primer – Pointers

We’ve learned about arrays and how to access individual elements via indexing.

Pointers are simply indices into arbitrary memory.

Every variable is stored somewhere in memory at runtime and so has a memory address.

You can get the address of a variable using the “address-of operator,” `&`.

The address-of operator returns a pointer.

A pointer in a type declaration is indicated by `*`.

C Language Primer – Pointers

This program may print `0x16d43f348`.

```c
#include <stdio.h>

int main(void) {
  int a = 0;
  int* b = &a;
  printf("%p\n", b);
  return 0;
}
```

C Language Primer – Pointers

Q If you run the below program multiple times, what does it print?

```
#include <stdio.h>

int main(void) {
  int a = 0;
  int* b = &a;
  printf("%p\n", b);
  return 0;
}
```

C Language Primer – Pointers

We can use pointers to modify the values at the memory addresses pointed to.

```c
#include <stdio.h>

int main(void) {
   int a = 0;
   int* b = &a;
   *b = 1;
   printf("%d\n", a);
   return 0;
}
```

The above prints 1.

We changed the value of `a` indirectly via `b`, a pointer to `a`.

C Language Primer – Pointers

The dereference operator, also `*`, converts a pointer to its underlying value.

```c
#include <stdio.h>

int main(void) {
  int a = 1;
  int* b = &a;
  int c = *b + 1;
  printf("%d\n", c);
  return 0;
}
```

The above prints 2.

C Language Primer – Pointers

With pointers, we can write functions that can change values in other scopes.

```
#include <stdio.h>

void add_one(int* a) {
  *a = *a + 1;
}

int main(void) {
  int a = 0;
  add_one(&a);
  printf("%d\n", a);
  return 0;
}
```

The above prints 1.

C Language Primer – Pointers

You can point to “nothing” by assigning `NULL` (or `nullptr` in C23) to a pointer.

Such a pointer is called a “null pointer.”

`NULL` is defined to be 0 or a pointer to address 0.

It is an error to dereference a null pointer.

The result of dereferencing a null pointer is platform-specific, but you will likely encounter a “segmentation fault.”

More generally, it is an error to dereference an address that is not mapped into your program’s memory space at runtime.

C Language Primer – Pointers

Q What does the following program do? Are there any bugs?

```c
#include <stdio.h>

int* make_one(void) {
  int one = 1;
  return &one;
}

int main(void) {
  int* a = make_one();
  printf("%d\n", *a);
  return 0;
}
```

C Language Primer – Pointers – Pointer Arithmetic

When we printed a pointer, we saw a value such as `0x16d43f348`.

This is just a (hexadecimal) number.

Changing the number changes what the pointer points at.

C supports mathematical operations on pointers, called pointer arithmetic:

1. Increment and decrement
2. Adding or subtracting integers from pointers
3. Subtracting one pointer from another to determine the number of elements in a range of memory

C Language Primer – Pointers – Pointer Arithmetic

Memory addresses count individual bytes, but pointer arithmetic adjusts pointers by the number of bytes of their underlying type.

It’s easier to visualize with an example.

C Language Primer – Pointers – Pointer Arithmetic

```c
#include <stdio.h>

int main(void) {
  int a = 0;
  int* b = &a;
  int* c = b + 1;
  printf("b, c: %p, %p\n", b, c);
  printf("sizeof(int): %d\n", sizeof(int));
  printf("c - b: %d\n", c - b);
  return 0;
}
```

```
$ ./a.out
b, c       : 0x16f187348, 0x16f18734c
sizeof(int): 4
c - b      : 1
```

C Language Primer – Pointers – Pointer Arithmetic

```c
#include <stdio.h>

int main(void) {
  int a = 0;
  int* b = &a;
  int* c = b + 1;
  printf("b, c: %p, %p\n", b, c);
  printf("sizeof(int): %d\n", sizeof(int));
  printf("c - b: %d\n", c - b);
  return 0;
}
```

```
$ ./a.out
b, c        : 0x16f187348, 0x16f18734c
sizeof(int): 4
c - b       : 1
```

Q Why do the last digits of the addresses `b` and `c` point to differ by 4 and not 1?

Q Why is `c - b` 1?

Q Aside: is there any problem with dereferencing `c`?

C Language Primer – Pointers – Pointer Arithmetic

The array subscript operator `[]` also works on pointers.

For example, when used on a pointer `p` as `p[n]`, the operator automatically dereferences the element referred to by `p + n`.

```
int main(void) {
  int a[] = {0, 1, 2, 3, 4, 5};
  int* b = a;
  int c = b[3];
  return 0;
}
```

C Language Primer – Pointers – Pointers to Pointers

You can have pointers to pointers.

For example, `int**` is a pointer to a pointer to an `int`.

In theory you can have infinite levels of indirection, but in practice toolchains put restrictions on the number of levels.

Many levels of indirection in your code is a complexity warning sign.

We will see practical examples of pointers to pointers later.

C Language Primer – Strings

Strings are sequences of characters.

We will focus on byte strings containing ASCII characters.

Strings must be null-terminated in C, meaning the last character must be `\0`.

Such strings are called null-terminated byte strings.

String variables are typed as `char*`.

C Language Primer – Strings

```c
#include <stdio.h>

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

String literals do not need an explicit null terminator. The compiler adds them for us.

Notice that we pass `str`, not `*str`, to `printf`. The `%s` format specifier expects a pointer to `char`.

C Language Primer – Strings

If we dynamically create strings, we need to take care to add the null terminator ourselves.

We will see how to dynamically create strings later.

String literals are read-only in C. We can only modify dynamically-created strings.

C Language Primer – Strings

As suggested by the type of the string pointer, `char*`, individual elements of a string are `char`s.

We’ve already seen how to assign numeric values to `char`s. We can also assign character literals.

```
int main(void) {
  char a = 'a';
  char null = '\0';
  char new_line = '\n';
  char capital_a = 65;
  return 0;
}
```

C Language Primer – Strings

Q How can we make `'B'` using `capital_a`?

```c
int main(void) {
  char a = 'a';
  char null = '\0';
  char new_line = '\n';
  char capital_a = 65;
  return 0;
}
```

C Language Primer – The Second Form of `main`

Earlier we showed the parameterless form of main.

There is another form that accepts command-line arguments.

```
int main(int argc, char** argv);
```

`argc` (argument count) is the number of command-line arguments received.

`argv` (argument vector) is a collection of command-line arguments received represented as null-terminated byte strings. Most popular runtimes today pass the program name as the first command-line argument automatically.

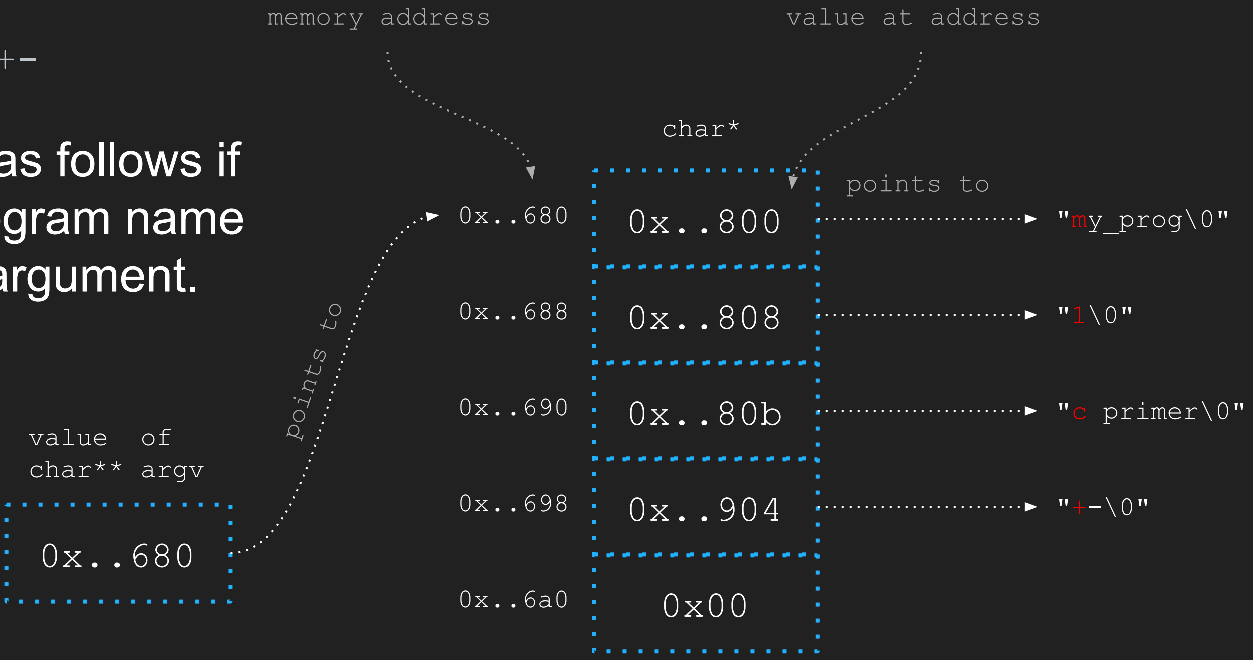
The collection itself is null-terminated.

C Language Primer – The Second Form of `main`

Given this program invocation:

```
$ ./my_prog 1 "c primer" +-
```

the contents of `argv` look as follows if the runtime passes the program name as the first command-line argument.



C Language Primer – Structures

C provides a way for users to define their own aggregate types called structures.

```
struct Position {
  int x;
  int y;
};

struct Player {
  char* name;
  int health_points;
  struct Position position;
};

int main(void) {
  struct Player player;
  return 0;
}
```

C Language Primer – Structures

When we create space on the stack for `player` via `struct Player player`, all members have uninitialized values. We need to manually initialize them.

We can access members on structs using the “member access operator,” sometimes called the “dot operator,” `.`.

```
// <Position and Player definitions here.>

int main(void) {
  struct Player player;
  player.name = "default name";
  player.health_points = 100;
  player.position.x = 0;
  player.position.y = 0;
  return 0;
}
```

C Language Primer – Structures

Such initialization is usually wrapped in a function so it can be reused.

```c
// <Position and Player definitions here.>

struct Player player_init(char* name, int hp, int pos_x, int pos_y) {
  struct Player player;
  player.name = name;
  player.health_points = hp;
  player.position.x = pos_x;
  player.position.y = pos_y;
  return player;
}

int main(void) {
  struct Player p = player_init("default_name", 100, 20, 30);
  return 0;
}
```

C Language Primer – Structures

If you have a handle on a struct via a pointer, you can access the struct's members by using either the dereference operator or the "member access operator" (sometimes called the "arrow operator"), `->`.

```c
#include <stdio.h>

// <Position and Player definitions here.>
// <player_init definition here.>

int main(void) {
  struct Player p = player_init("default_name", 0, 20, 30);
  struct Player* p_ptr = &p;
  printf("%s\n", (*p_ptr).name);
  printf("%d\n", p_ptr->health_points);
  return 0;
}
```

C Language Primer – Memory Allocation and Management

So far we talked about the stack and read-only memory where string literals are stored. There is another important section of memory called the “heap.”

Stack frames are destroyed when their associated functions return. Memory on the heap is independent of the stack and is allocated and destroyed explicitly by the programmer.

Using the heap, we can allocate memory for any content we’d like and share that memory with arbitrary scopes until we choose to deallocate the memory.

The C standard library provides various functions for dynamic memory management. We will focus on `malloc`, `calloc`, and `free`.

C Language Primer – Memory Allocation and Management

```
#include <stdlib.h>

int main(void) {
  int* a = malloc(4 * sizeof(int));
  free(a);
  a = NULL;
  return 0;
}
```

This program uses `malloc` (“memory allocation”) to reserve space on the heap for 4 `int`s. `malloc` allocates *bytes*, so we need to do some math.

We call `free` to deallocate the memory. We set `a` to `NULL` to remind ourselves that `a` does not currently point to any address allocated for our use.

C Language Primer – Memory Allocation and Management

`malloc` does not initialize memory. We can use `calloc` instead, which initializes all allocated bytes to 0.

Here is an example of dynamically creating a string.

```
#include <stdlib.h>

int main(void) {
  char* str = calloc(10, sizeof(char));
  str[0] = 'c';
  str[1] = 'a';
  str[2] = 't';
  str[3] = '\0';
  free(str);
  return 0;
}
```

C Language Primer – Memory Allocation and Management

Q Is the highlighted line below necessary in this case to form a null-terminated byte string?

```c
#include <stdlib.h>

int main(void) {
  char* str = calloc(10, sizeof(char));
  str[0] = 'c';
  str[1] = 'a';
  str[2] = 't';
  str[3] = '\0';
  free(str);
  return 0;
}
```

C Language Primer – Memory Allocation and Management

Q Can you dynamically create strings on the stack? If so, how? If not, why not?

C Language Primer – Memory Allocation and Management

Q How would you allocate memory on the heap to store an instance of `Position`?

```c
struct Position {
  int x;
  int y;
};
```

C Language Primer – Memory Allocation and Management

Q How would you allocate memory on the heap to store an instance of `Player`?

```
struct Position {
  int x;
  int y;
};

struct Player {
  char* name;
  int health_points;
  struct Position position;
};
```

C Language Primer – Working With Multiple Source Files

As you develop larger projects, you will want to split your code across multiple source files.

Code is typically organized in header files and corresponding implementation files.

Header files have the `.h` file extension. Implementation files have the `.c` file extension. This is not mandatory but is customary.

C Language Primer – Working With Multiple Source Files

Consider this example library that provides one function. Here is its header file, `my_lib.h`.

```c
#ifndef MY_LIB_H_
#define MY_LIB_H_

/**
 * @param a The number to add one to.
 * @return The sum of @c a and 1.
 */
int add_one(int a);

#endif // MY_LIB_H_
```

C Language Primer – Working With Multiple Source Files

Here is its corresponding implementation file, `my_lib.c`.

```c
#include "my_lib.h"

int add_one(int a) {
  return a + 1;
}
```

C Language Primer – Working With Multiple Source Files

To make an executable, you'll need a `main` function. If this is implemented in `main.c`, then the compiler invocation may look like:

```
$ clang main.c my_lib.c -I./
```

The `-I./` portion says to add `./` to the path to search for include files.

As programs grow to contain many source files across many directories, the compiler invocation becomes unwieldy to create manually. Build systems help by generating the invocation for you.

We will discuss build systems more later.

C Techniques Primer

1. `const`
2. Pragmas
3. Data-Oriented Programming
4. Design by Contract
5. Testing

C Techniques Primer – `const`

The `const` keyword in the C language makes variables read-only. Use `const` wherever possible to help ensure you're only modifying what you want to modify. Note that it is not foolproof, and note that meaning is slightly tricky when used with pointers.

See this documentation for more.

C Techniques Primer – Pragmas

We learned about include guards in header files. There is a non-standard but widely supported alternative form for header guards.

You can replace all of the following

```
#ifndef MY_MODULE_FILE_H_
#define MY_MODULE_FILE_H_

// ...

#endif MY_MODULE_FILE_H_
```

with simply `#pragma once` at the top of the file.

C Techniques Primer – Data-Oriented Programming

C is an “imperative” language, not an object-oriented one.

While C does have structures, such types do not support methods like classes do in other languages.

C also does not support namespacing.

One common way of associating a group of functions with a custom data type is by prefixing function names with the name of the custom type the functions operate on and taking an instance of the custom type as the first parameter.

C Techniques Primer – Data-Oriented Programming

```c
#pragma once

#include <stdbool.h>
#include <stddef.h>

struct Buffer {
  char* _buffer; // A place to store the data
  size_t _element_size; // The byte size of an element in @c _buffer
  size_t _capacity; // The maximum number of elements @c _buffer can hold
  size_t _size; // The current number of elements in @c _buffer
};

struct Buffer buffer_new(size_t num_elements, size_t sizeof_element);
bool buffer_push(struct Buffer* b, char* element);
char* buffer_at(struct Buffer* b, size_t index);
void buffer_free(struct Buffer* b);
```

C Techniques Primer – Design by Contract

Design by Contract is a software construction technique that formalizes software interfaces with so-called “contracts.”

Contracts feature pre-conditions, post-conditions, and invariants.

Pre-conditions are things that must happen before an interface is used.

Invariants are properties that must be true about the data coming into an interface.

Post-conditions are promises that the interface makes to the user about the resultant state of the runtime.

C Techniques Primer – Design by Contract

```c
/** A struct containing something stateful. */
struct StateThing {
  // ...
};

/**
 * Initialize the @c StateThing library.
 *
 * Preconditions:
 * - @c The runtime must not have called @c statething_library_init already.
 *
 * Postconditions:
 * - It is invalid to call @c statething_library_init again in the same
 *   runtime.
 */
void statething_library_init(void);
```

C Techniques Primer – Design by Contract

```c
/** A struct containing something stateful. */
struct StateThing {
  // ...
};

// void statething_library_init(void);

/**
 * Perform foo with a @c StateThing instance.
 *
 * Preconditions:
 * - @c statething_library_init has been called in the runtime.
 *
 * @param st The @c StateThing instance to operate on. Cannot be @c NULL .
 * @return Some data resulting from doing foo. The caller owns the memory.
 */
char* statething_foo(struct StateThing* st);
```

C Techniques Primer – Design by Contract

If an interface is used but its contract is not fulfilled, the contract is said to be broken or violated.

In this case, the program enters a (user-defined) invalid state.

If the API exposes error information, perhaps an error is raised. Or perhaps the implementation chooses to terminate the runtime. The behavior should be documented somewhere.

C Techniques Primer – Design by Contract

Q If your function’s contract is violated, is it okay to terminate the program?

C Techniques Primer – Testing

Tests are written as additional small programs that exercise your code.

It is therefore helpful to organize your code so that as much as possible can be used as a library. This way test programs can import and exercise your code.

Most modern build systems have first-class support for registering binaries as tests, for example CMake's `add_test` and Bazel's `cc_test`.

Tests will often make use of `assert` (from `assert.h`) to simply terminate the program if some expectation is not met in the test.

C Techniques Primer – Testing

You may want to reuse variable names in a single test application but find reassignment difficult, such as when assigning a new array to a variable.

You can create nested “blocks” to help here. A block creates a local, nested scope.

```c
int main(void) {
  {
    int arr[] = {0, 70, 21};
    // Do something with arr.
  }

  {
    int arr[] = {0, 70, 21, 50, 99};
    // Do something else with a different instance of arr.
  }

  return 0;
}
```

C Tooling Primer

1. Compiler Flags
2. Build Systems
3. Test Frameworks
4. Documentation Generators
5. Debuggers
6. Static Analyzers
7. Sanitizers

C Tooling Primer – Compiler Flags

Our compilers have a variety of checks and diagnostics that help us catch bugs.

`-Wall`: enable “all” warnings.

`-Wextra`: enable even more warnings not included in `-Wall`.

`-Werror`: treat warnings as errors; use with discretion.

`-Wconversion`: warn for any conversion that may change a value.

`-g3`: emit debugging symbols, in particular more information than `-g`.

`-Wno-<option-name>` to disable a particular warning.

C Tooling Primer – Build Systems

Compiler invocations become complicated as projects grow to many source files.

Build systems help us organize our code and manage compiler invocations.

There are many popular build systems. Here we’ll focus on Bazel.

Bazel is complicated, and the learning curve for the complex features is steep, but simple project organization and compilation is straightforward.

C Tooling Primer – Build Systems – Bazel

Assume the following source tree:

```
bin/
  |_ main.c
lib/
  |_ foo/
    |_ foo.c
    |_ foo.h
    |_ tests/
        |_ test_foo.c
```

The project will have two executables (one is a test) and one library.

C Tooling Primer – Build Systems – Bazel

We first mark the root of the workspace by adding an empty `WORKSPACE` file in the project root.

```
bin/
   |_ main.c
lib/
   |_ foo/
      |_ foo.c
      |_ foo.h
      |_ tests/
            |_ test_foo.c
WORKSPACE
```

C Tooling Primer – Build Systems – Bazel

To create a build target for our library “foo,” we place a `BUILD.bazel` file in `lib/foo/`.

Its contents are as follows.

```
cc_library(
  name = "libfoo",
  srcs = ["foo.c"],
  hdrs = ["foo.h"],
  visibility = ["//visibility:public"],
)
```

```
bin/
   |_ main.c
lib/
   |_ foo/
      |_ BUILD.bazel
      |_ foo.c
      |_ foo.h
      |_ tests/
         |_ test_foo.c
WORKSPACE
```

C Tooling Primer – Build Systems – Bazel

To create a build target for our test, in the same `BUILD.bazel` file, we add:

```
cc_library(
   name = "libfoo",
   srcs = ["foo.c"],
   hdrs = ["foo.h"],
   visibility = ["//visibility:public"],
)

cc_test(
   name = "test_libfoo",
   srcs = ["tests/test_foo.c"],
   deps = [":libfoo"],
   timeout = "short",
)
```

```
bin/
   |_ main.c
lib/
   |_ foo/
      |_ BUILD.bazel
      |_ foo.c
      |_ foo.h
      |_ tests/
          |_ test_foo.c
WORKSPACE
```

C Tooling Primer – Build Systems – Bazel

Lastly, to create a build target for our main executable, we add a new `BUILD.bazel` file in `bin/`, and its contents are as follows:

```
cc_binary(
  name = "main",
  srcs = ["main.c"],
  deps = ["//lib/foo:libfoo"],
)
```

```
bin/
  |_ BUILD.bazel
  |_ main.c
lib/
  |_ foo/
     |_ BUILD.bazel
     |_ foo.c
     |_ foo.h
     |_ tests/
         |_ test_foo.c
WORKSPACE
```

C Tooling Primer – Build Systems – Bazel

We can build our entire project by issuing `bazel build //...`.

We can run our main binary issuing `bazel run //bin:main`.

We can run all of our tests by issuing `bazel test //...`.

We can run a specific test by issuing `bazel test //lib/foo:test_libfoo`.

If a test failure occurs, Bazel may hide the failure output. You can show the output by passing `--test_output=errors` to `bazel test`.

If you need to build binaries with debugging symbols, pass `--strip=never -c dbg` to `bazel build`.

C Tooling Primer – Build Systems – Bazel

If you need to build against a specific language version, create a `.bazelrc` file in the same directory as `WORKSPACE` and add `--copt=-std=c23` to the `build` profile:

```
build --copt=-std=c23
```

You can put other compiler command-line arguments in that file as well. For each argument, you need an additional `--copt=`.

Bazel suggests that if your option applies only to C and not to C++, you should use `--conlyopt=`.

The more scalable way to set the language version is to use a formal toolchain configuration. We will not cover that here.

C Tooling Primer – Build Systems – Bazel

Note the structure of the source code in the example project.

The header files and implementation files are next to each other.

This is not usually the case on your system for installed libraries.

```
bin/
   |_ BUILD.bazel
   |_ main.c
lib/
   |_ foo/
      |_ BUILD.bazel
      |_ foo.c
      |_ foo.h
      |_ tests/
            |_ test_foo.c
WORKSPACE
```

C Tooling Primer – Test Frameworks

Test frameworks support creating and executing tests and reducing boilerplate.

Some options for C are

1. Unity
2. µnit
3. check
4. criterion

Another popular option is GoogleTest. GoogleTest is a C++ test framework, but you can use it to test your C functions (in a C++ context) as well. You will need to make your C functions callable from C++ or vice versa; see here for more.

C Tooling Primer – Documentation Generators

The most popular documentation generator for C and C++ projects is Doxygen.

Search online for how to set it up for your project.

Note that other options exist, such as Sphinx, but typically they still interface with Doxygen during the build process.

C Tooling Primer – Debuggers

Two of the more popular debuggers are `gdb` (GNU debugger) and `lldb` (Low-Level Debugger, in the LLVM family).

There are plenty of online tutorials on how to use these debuggers. Focus on:

1. Breakpoints
2. Next (go to next instruction, skip over function calls)
3. Step (go to next instruction, enter function calls)
4. Printing values
5. Printing the stack
6. Moving up and down the stack
7. Printing source code around the line the debugger is currently stopped on

C Tooling Primer – Static Analyzers

Static analyzers are programs that evaluate your source code to detect coding style violations and potentially bugs in your implementation.

Among the most popular is `clang-tidy`.

C Tooling Primer – Sanitizers

Sanitizers are compiler features that add special information into your compiled programs to help find issues at runtime. They are very useful, but they also impose some time and memory overhead.

Two popular sanitizers relevant to us are `ubsan` ("u-b-san," undefined behavior sanitizer), and `asan` ("a-san," address sanitizer).

These are available for at least GCC and Clang, but you may need to reconfigure your environment or even build your toolchain from source to enable advanced features. There are plenty of tutorials on the web.

Enable these sanitizers via: `-fsanitize=address -fsanitize=undefined`.

C Tooling Primer – Sanitizers

Another popular sanitizer is valgrind. The memcheck feature is particularly well-known.

Valgrind will detect memory issues such as leaks by examining your program at runtime.

Look up the valgrind documentation for usage instructions.

Project Specification

1. Stack Machine
2. Language Specification
3. Code Generator
4. Tasks
5. Hints

Project Specification – Stack Machine – Specification

The stack machine supports the following instructions. Instructions are 16 bits wide. The first 2 bits counting from the most significant bit are the instruction type. The remaining 14 bits are the instruction data.

Instruction Type (bits 15, 14)	Description
`0b00`	Store a positive integer. The last 14 bits are the integer.
`0b01`	Store a negative integer. The last 14 bits are the *positive* integer. (Assume the machine converts the positive integer into a negative integer when storing the value.)
`0b10`	Perform an opcode-based instruction. (See next slide.)
`0b11`	Unsupported.

Project Specification – Stack Machine – Specification

Opcodes for opcode-based instructions are as follows. The opcode value starts from the least significant bit.

Opcode (bits 13 - 0)	Description
`0x0`	Halt. Stop executing a program.
`0x1`	Add. Pop the top two values off the stack, add them, and push the result back onto the stack.
`0x2`	Subtract. Pop the to two values off the stack, subtract the first popped value from the second, and push the result back onto the stack.
`0x3`	Multiply. Pop the top two values off the stack, multiply them, and push the result back onto the stack.
`0x4`	Integer division. Pop the top two values off the stack, divide the first popped value into the last popped value, and push the result back onto the stack.

Project Specification – Stack Machine – API

The next slide shows an API for the stack machine.

Project Specification – Stack Machine – API

```c
/**
 * @return A new instance of a VM.
 */
struct VM vm_new(void);

/**
 * Load a program into a VM.
 *
 * @param vm The virtual machine to load the program into.
 * @param program An array of instructions.
 * @param program_length The number of instructions in @c program.
 */
void vm_load_program(struct VM* vm, uint16_t* program, size_t program_length);

/**
 * Run a loaded program.
 *
 * @param vm The VM containing the program to run.
 */
void vm_run(struct VM* vm);

/**
 * Get the last value pushed to the VM's stack.
 *
 * @param vm The VM holding the stack to inspect.
 * @return The last value pushed to the VM's stack.
 */
int16_t vm_peek(struct VM* vm);
```

Project Specification – Stack Machine – CLI

The command-line interface to the stack machine is a program that takes a single command-line argument which is a binary file containing bytecode to run. The program instantiates a virtual machine, runs the program in the machine, and when the program finishes, it prints the result of the last computation (add, sub, mul, div) performed.

Project Specification – Language Specification

The language specification for this project is the following simple grammar.

```
expression: (number | operator)*
number: [+|-]?[0-9]+
operator: ('+' | '-' | '*' | '/')
```

Programs are expressions written in postfix notation. Whitespace separates numbers and operators and is otherwise ignored.

The next slide features some example programs.

Project Specification – Language Specification

An empty source file is a valid program. It simply halts.

```
0
```

The program above pushes 0 to the stack and halts.

```
0 -20 + 5
/
```

The program above pushes 0 to the stack, pushes -20 to the stack, performs addition (pops -20, pops 0, adds them, stores the result on the stack), pushes 5 to the stack, and then performs integer division. The final state of the stack is -4.

Project Specification – Language Specification

```
1 +
```

The above program pushes 1 to the stack and then attempts addition.

There are not enough values on the stack to perform addition. The virtual machine behavior would be undefined.

Project Specification – Code Generator

The code generator is a command-line program that takes a single command-line argument which is a path to a file containing source code. The tool converts the source code into bytecode and writes the bytecode to a file in the current working directory.

The code generator does not need to make determinations about whether a program would cause a virtual machine to enter an undefined or error state. For example, the code generator can still emit code for the previous program `1 +` even though the virtual machine will enter an undefined state.

Project Specification – Tasks

1. Implement the VM API.
   a. You are free to modify and extend the API as you please.
   b. The API may have issues (ex: missing documentation or insufficient information in function signatures to handle errors).
2. Implement the VM CLI.
3. Implement the code generator.

Project Specification – Hints

The slides before this do not cover everything you need to elegantly implement the project.

The slides after provide hints in the form of code snippets, additional techniques, and design notes to help you in your implementation.

Project Specification – Hints – Virtual Machine – API

`size_t` is often used in place of `int` or `unsigned int` to count the number of bytes in some collection. See its documentation for more.

A stack-based virtual machine uses a stack as its memory space. In C, a stack is easily implemented with an array and an index into the array keeping track of the top element.

When processing instructions, consider implementing a simplified version of the "instruction cycle," namely `fetch`, `decode`, and `execute`. `fetch` gets the next instruction to process. In our case, `decode` breaks the instruction into two parts: instruction type and instruction data. `execute` actually "runs" the instruction, updating the state of the virtual machine's stack.

Project Specification – Hints – Virtual Machine – API

The format specifiers for fixed-width integer types are in the table for the "`fprintf` family of functions" here. These format specifiers are strings, so you can create a format string using string concatenation, like:

```
printf("top of stack: %" PRId16 "\n", vm_top_of_stack(vm));
```

You will need to use `for` and maybe `while` loops.

You will need to use `if`.

You will likely need to use the `switch` statement.

Project Specification – Hints – Virtual Machine – API

You will likely have to use bitwise operators, such as bitwise AND.

For example, here is how to test whether the 3rd bit (from 0) in a `uint8_t` is set.

```
bool is_third_bit_set(uint8_t i) {
  uint8_t mask = 0x08; // binary: 0000 1000
  bool is_set = i & mask; // if set, result is non-zero (true)
  return is_set;
}
```

Project Specification – Hints – Virtual Machine – API

You may have to use bit shifting, such as bitwise right shift.

For example, here is how to shift the top 2 bits to the position of the bottom 2 bits in a `uint8_t`.

```
void shift_down(uint8_t i) {
  i = i >> 6; // i is shifted 6 bits right,
              // so `bbxx xxxx` becomes `0000 00bb`.
              // The bits that "fall off" the right are lost.
              // The bits that shift in from the left are 0.
              // Note that right shift works differently for signed types,
              //   remember the sign bit!
}
```

Project Specification – Hints – Virtual Machine – API

The `string.h` header file includes various functions to work with memory, such as performing memory copies. See this documentation for more.

Project Specification – Hints – Virtual Machine – CLI

File IO in C is done through a subset of the API provided in `stdio.h`.

Study at least `fopen`, `fread`, `fwrite`, and `fclose`.

When you want to read and write binary files, take note of `fopen`’s file access flag `'b'`.

Opening a file returns a file handle of type `FILE*`, a file identifier that you use in calls to other parts of the file IO API.

When you’re done reading or writing to a file, call `fclose`. This flushes (writes) any remaining content of the backing buffer to the file and then releases the file management resources back to the operating system.

Project Specification – Hints – Virtual Machine – CLI

When reading content from a file, you may want to prepare a buffer beforehand that is large enough to store the contents of the file. Here is how you can determine the number of bytes in a *binary* file. Error handling is not shown.

```c
// Open a file for reading in binary mode.
FILE* fp = fopen("file_name", "rb");
// Move file position indicator to the end of the file.
fseek(fp, 0, SEEK_END);
// Get the number of bytes between start of file and position indicator.
size_t num_bytes = (size_t) ftell(fp);
// Move file position indicator back to the beginning.
rewind(fp);
// Make a buffer of size num_bytes.
char* bytes = malloc(...);
// Clean up the file.
fclose(fp);
```

Project Specification – Hints – Code Generator

As we saw in an earlier slide, code generation often involves translating input source code into assembly and then assembly into machine code or bytecode.

For the sake of simplicity and time, we will skip assembly (although you can implement an assembly language if you'd like).

Our code generator may be implemented in two broad parts: a lexical analyzer and a code generator.

A lexical analyzer reads raw source code and returns a collection of tokens. This is called "tokenization," and the thing that does it is called a "tokenizer."

The code generator then processes the tokens and emits bytecode.

Project Specification – Hints – Code Generator

Tokens are strings, and our tokenizer returns a collection of tokens. If individual tokens are typed as `char*`, the collection may be typed as `char**`.

Consider the example program `0 -20 +`.

Because our language is so simple, it turns out that each non-whitespace character is a part of a token. The tokens in this case are `["0", "-20", "+"]`.

It turns out each of our tokens corresponds directly to a machine instruction:

1. `0` : push positive number to the stack
2. `-20`: push negative number to the stack
3. `+` : opcode based instruction

Project Specification – Hints – Code Generator

If tokens are strings, how much space should we reserve for each string?

Let’s consider the limitations of our VM. Instructions are 16 bits, and a number’s magnitude is stored in 14 bits. Consider the maximum and minimum decimal value representable in 14 bits.

When allocating memory for your token, don’t forget to allocate space for the sign character and the null terminator.

Project Specification – Hints – Code Generator

You will find string operations useful. The API for working with strings in C is broad. See here for a list of functions.

You may particularly find `strcmp`, `strlen`, and `atoi` useful.

You can test individual characters in a string directly. For example, to find if the letter `a` is the third element (counting from 0) in a `char* str`: `str[3] == 'a'`.

Project Specification – Hints – Code Generator

Here is an API idea for a lexer library:

```c
/**
 * Tokenize source code.
 *
 * @p source A pointer to the source to tokenize. Cannot be @c NULL.
 * @p num_bytes The number of bytes in the @c source buffer.
 * @return A pointer to a null-terminated collection of tokens represented
 *         as null-terminated strings. The caller must free the collection
 *         using @c lexer_free_tokens .
 */
char** lexer_tokenize(char* source, size_t num_bytes);

/**
 * Free the collection of tokens.
 *
 * @p tokens A collection of tokens; must have been prepared by
 *         @c lexer_tokenize.
 */
void lexer_free_tokens(char** tokens);
```

Project Specification – Hints – Code Generator

Here is an API idea for a code generator library:

```c
/**
 * Generate bytecode from a collection of tokens.
 *
 * @p tokens A collection of tokens to generate bytecode from.
 * @return Bytecode. The caller must clean up the memory up using
 *         @c code_gen_free_bytecode when done with the bytecode.
 */
uint16_t* code_gen_generate_bytecode( char** tokens);

/**
 * @p bytecode The bytecode to free.
 */
void code_gen_free_bytecode( uint16_t* bytecode);
```