Checklist03

Compiling and Building

1. Why Study C?
   - teaches low-level memory and systems concepts
   - produces fast, portable programs
   - foundation for understanding other languages
   1. Why not just **Python** or **C++?**
      - Python hides memory and system details
      - C++ adds abstractions that obscure low-level behavior
   2. Why is **C** considered a **systems programming language?**
      - Direct memory and hardware access via pointers and manual memory management
        - `malloc()`
        - `calloc()`
        - `free()`
      - efficient, minimal runtime
      - used to build OS, compilers, embedded systems
   3. What is **C's** relationship to **Unix**?
      - C was created to rewrite Unix portably
        - Unix originally written in assembly
      - Unix system calls and APIs are accessed through C
      - Learning C is learning how to program Unix-level systems
2. What is the **compiler pipeline**s?
   1. What exactly happens when you compile a program (ie. describe the **compiler pipeline**)
      - The **compiler pipeline** transforms your source code into an executable in several stages:
        1. **Preprocessing:**
           - expands `#include`, `#define`, and conditional compilation
           - copies and pastes all your header files into the program (libraries and headers)
        2. **Compiling**:
           - Translates C code in assembly
           - syntax check is done
           - optimization is done (maybe `puts` instead of `printf`)

3. **Assembly**:
    - converts assembly code to binary machine code in an object file (`.o`)
    - not all binary code is executable
        - it may need to be packaged
4. **Linking**:
    - takes binary code and packages it up to to produce an executable

- NOTE: the first three steps are often referred to as "compile" and the last step as "linking"

2. What is the difference between a **dynamic executable** and a **static executable?**
    - **Dynamic Executable:**
        - Links to shared libraries at **runtime** → smaller file, libraries can be updated independently
    - **Static Executable:**
        - All library code is included at **compile time** → faster startup time, much larger in size
3. What is the difference between a **shared library** and a **static library?**
    - **Shared Library:**
        - **Static Library (`.a`)**: code is copied into the executable during linking
        - **Shared library (`.so`)**: Code stays separate; loaded at runtime
            - key difference:
                - static = compile-time inclusion
                - shared = runtime inclusion

3. How do you write a **Makefile** that utilizes **rules** and **variables** for a program that consists of multiple files?

```
CC=      gcc
CFLAGS=      -Wall -g -std=gnu99 -fPIC
LD=      gcc
LDFLAGS= -L.
LIBS=        -lstr
AR=      ar
ARFLAGS= rcs
TARGETS= libstr.so libstr.a trit.dynamic trit.static

all:         $(TARGETS)

#-------------------------------------------------------------------------
-------
# TODO: Add rules for libstr.a libstr.so
#-------------------------------------------------------------------------
```

```
-------

str.o: str.c str.h
	$(CC) $(CFLAGS) -c str.c -o $@
libstr.so: str.o
	$(LD) $(LDFLAGS) -shared str.o -o $@
libstr.a: str.o
	$(AR) $(ARFLAGS) $@ str.o

#-------------------------------------------------------------------------
-------
# TODO: Add rules for trit.dynamic trit.static
#-------------------------------------------------------------------------
-------

trit.o: trit.c str.h
	$(CC) $(CFLAGS) -c trit.c -o $@
trit.dynamic: libstr.so trit.o
	$(LD) $(LDFLAGS) trit.o $(LIBS) -o $@
trit.static: trit.o libstr.a
	$(LD) $(LDFLAGS) trit.o libstr.a -o $@
```

- `CC = gcc` -> compiler for C code
- `CFLAGS` : flags passed during compilation
  - `-Wall` -> enables all warnings
  - `-g` -> include debug symbols (useful for gdb)
  - `-std=gnu99` -> use GNU99 C standard
  - `-fPIC` -> needed for shared libraries (`.o`)
- `LD = gcc` -> linker (usually same as compiler)
- `LDFLAGS = -L` -> add current directory to library search path when linking
- `LIBS = -lstr` -> link with `libstr` library
- `AR = ar` -> archiver tool to create static libraries
- `ARFLAGS = rcs` -> options for `ar`
  - `r` = insert or replace objects
  - `c` = create archive if it doesn't exist
  - `s` = write an index for faster linking
- `TARGETS` -> list of all things `make all` should build
- `all` is the default target
- `-c` -> compile only, don't link
- `$@` -> target
- `-shared` -> create a shared libarary

- notice the different pattern for `AR`
- key note:
  - for dynamic executable use `$(LIBS)`
  - for static executables pass the `.a` file directly, avoid `$(LIBS)`

Pointers, Arrays, Strings

1. What exactly is a **pointer? array? string?** How are they related?
   - a **pointer** is simply a variable that stores a memory address of another variable
   - an **array** is a contiguous block of memory holding elements of the same type
   - a **string** is a char array ending with NUL
   - An array acts like a pointer to the first element
   - strings are arrays of `char`
   - pointers can be used to iterate over arrays/strings
2. What does it mean to **dereference a pointer?**
   - Accessing the value stored at the memory address the pointer points to
3. How to we get the **address** of a variable?
   - `&`
4. What are multiple ways to access an element of an **array** or **string?**

```c
int arr[3] = {10,20,30};
char str[] = "abc";

arr[0] //10
str[1] // 'b'

*(arr + 2) //30
*(str + 0) // 'a'

int *p = arr;
p[1] // 20, same as *(p+1)
```

Memory Allocation

1. What is the difference between a **struct** and a **union**?
   - Struct:

        - memory layout:
            - each member gets its own memory, total size = sum of members (+ padding)
        - Usage:
            - store multiple values at the same time
        - Access:
            - all members can be used independently
    - Union:
        - memory layout:
            - all members share the same memory, total size = largest member (+padding)
        - usage:
            - stores one value at a time in the same location
        - access:
            - only one member is valid at a time
2. How much memory is allocated when we declare:
    - `int` : 4 bytes
    - `float` : 4 bytes
    - `double` : 8 bytes
    - `char` : 1 bytes
    - `int8_t` : 1 bytes
    - `int32_t` : 4 bytes
    - `int64_t` : 8 bytes
    - `size_t` : 8 bytes
    - **8 BITS IS A BYTE**
3. When declaring a variable, where is the data allocated?
    - stack: local variables
    - heap: `malloc` , `calloc`
    - data: `string literals` , `static variables` . `globals`
    - code: instructions

Memory Management and Linked Lists

1. How do we dynamically allocate memory? How do we deallocate that memory?
    - allocate using:
        - `malloc(4 * sizeof(int))`
        - `calloc(4, sizeof(int))`
        - `realloc(int *ptr, new # of bytes)`

- deallocate using:
  - `free(int *ptr)`

2. When should we allocate on the stack? heap? data? What are the advantages and disadvantages of utilizing each memory segment
   - stack:
     - lifetime: function scope
     - advantages: fast, no manual free, automatic cleanup
     - disadvantages: limited size, can't survive beyond function
   - heap:
     - lifetime: until `free`
     - flexible size, survives beyond function
     - slower, manual free required, fragmentation risk
   - Data:
     - lifetime: program lifetime
     - advantages: always available, easy to access
     - disadvantages: less flexible, occupies memory even if unused
     - DATA SEGMENT IS ~READONLY (i.e string literals)
3. What is a single linked list? doubly linked list?
   - SLL -> each node points to next node
   - DLL -> each node points to next and previous node
     - how would we insert an entry into a linked list?

```
#SLL prepending
Node *insert(Node *head, int value){
  Node *new_node = malloc(1 * sizeof(Node));
  new_node -> data = value;
  new_node -> next = head
  return new_node
}
```

     - how would we search for an entry in a linked list?

```
#SLL traversal

Node *search(Node *head, int value){
  Node *curr = head;
  while (curr != NULL){
       if (curr->data == value){return curr;}
       curr = curr->next;
  }
```

```
  return NULL; //not found
}
```

- how would we remove an entry in a linked list?
  - ```
    #SLL remove (first occurance)

    Node *remove(Node *head, int value){
      Node *curr = head, *prev = NULL;
      while (curr != NULL && curr->data != value){
           prev = curr;
           curr = curr->next;
      }
      if (curr == NULL) return head; //not found
      if (prev == NULL) return curr->next; // remove head
      else prev->next = curr->next;
      free(curr);
      return head;

    }
    ```

Bitsets and Data Representation

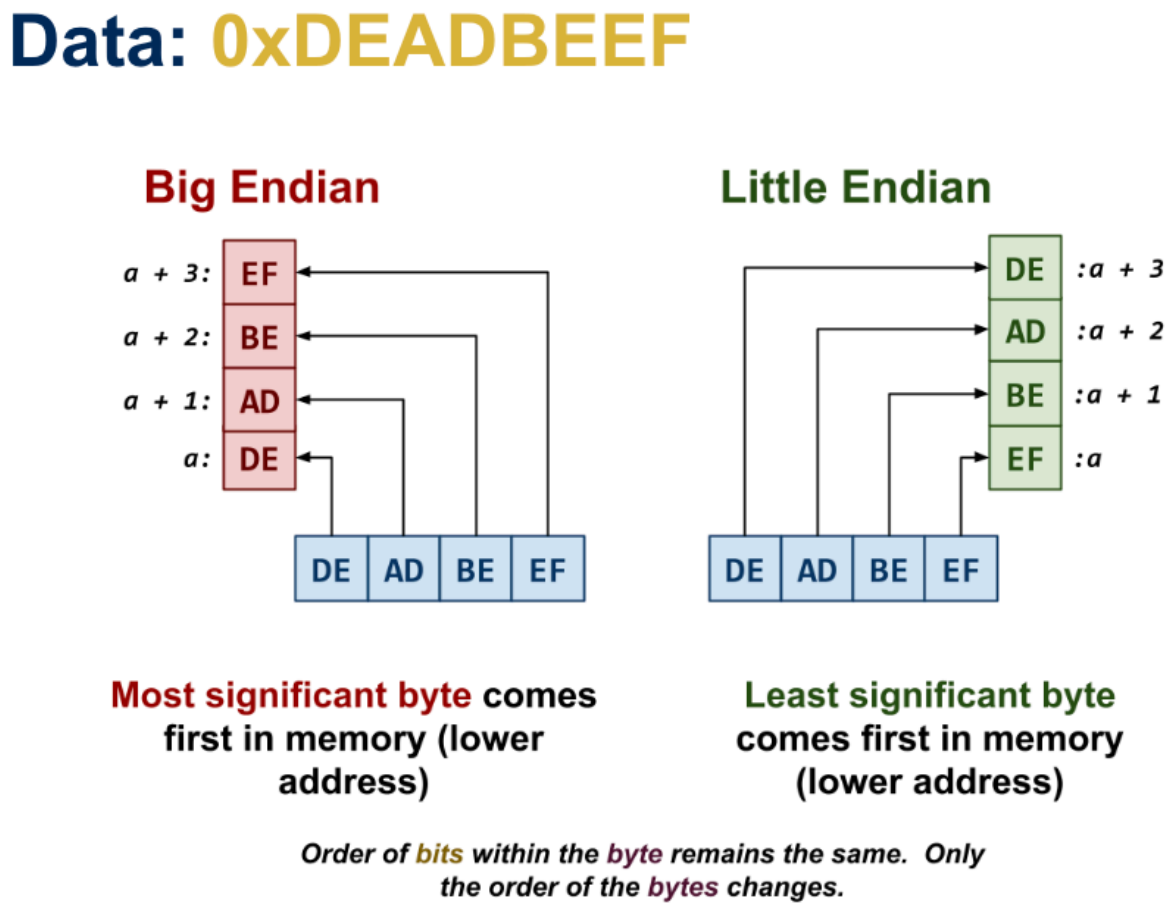
1. How do you implement a **bitset** in **C** that supports insertion, removal, and searching?
   - A bitset is a compact way to store boolean values using bits in integers
   - Representation:
     - Use an array of integers to hold the bits
     - Each bit represents a value (0 or 1)
     - ```
       #define BITS_PER_INT 32
       int bitset[1]; //for up to 32 elements
       ```
   - Operations:
     - Insert (set a bit)
     - ```
       void insert(int *bitset, int i){
         bitset[i/ BITS_PER_INT] |= (1<< (i % BITS_PER_INT));
       }
       ```
       - Remove (clear a bit)
     - ```
       void remove_bit(int *bitset, int i){
         bitset[i / BITS_PER_INT] &= ~(1<< (i % BITS_PER_INT));
       ```

```
}
```

- Search (check if bit is set)
  - ```
    int contains(int *bitset, int i){
      return (bitset[i/BITS_PER_INT] & (1<<(i %
    BITS_PER_INT))) != 0;
    }
    ```

- KEY IDEA:
  - `i / BITS_PER_INT` -> which integer in array
  - `i % BITS_PER_INT` -> which bit inside that integer

2. How is an integer represented on a **little endian** machine vs a **big endian** machine?
   - Little-endian: least significant byte (LSB) stored in first (lowest memory address)
   - Big-endian: most significant byte (MSB) stored first (lowest memory address)



Debugging

1. How do you debug problems such as segmentation faults in a C program using gdb
   1. start gdb `gdb ./myprogram`
    ```

2. run it : `(gdb) run`
    - if seg fault happens, gdb stops at the offending line
3. Insepect where is crashed
    - `(gdb) backtrace #shows the call stack`
    - `(gdb) list #shows source code around current line`
    - `(gdb) print var #inspect variables`
    - `(gdb) info locals #see all local variables`
4. step through code(optional for tricky bugs):
    - `(gdb) next # next line (skip inside function calls)`
    - `(gdb) step # next line (step into function calls)`
5. exit gdb
    - `(gdb) quit`

- COMMON causes of seg faults:
    - dereferencing `NULL` or uninitialized pointer
        - indicated by `variable = 0x0`
    - buffer overflows
    - accessing freed memory

2. How do you debug memory errors such as invalid reads, unitialized memory, and memory leaks in a C progam using valgrind?
    - types of errors valgrind detects:
        - INVALID READ/WRITE -> accessing memory you shouldn't (out of bounds, using freed memory)
        - USE OF UNINITIALIZED MEMORY -> using variables before initializing them, (they are declared)
        - MEMORY LEAKS -> allocated memory not freed before program exits
    - INVALID READ/WRITE:
        - ```
          ==12345== Invalid read of size 4
          ==12345==    at 0x4005A6: main (myprogram.c:12)
          ```
            - you're reading/writing memory you don't own
            - look at file and line (`myprogram.c:12`)
            - Usually caused by:
                - out-of-bounds array access
                - using freed memory
    - USE OF UNINITIALIZED VALUE
        - ```
          ==12345== Use of uninitialized value of size 4
          ==12345==    at 0x4005C7: sum_array (myprogram.c:8)
          ```

        - you used a variable before setting it.
        - fix: initialize variables before use
- MEMORY LEAKS
    - ```
      ==12345== 8 bytes in 1 blocks are definitely lost
      ==12345==    at 0x4C2BBAF: malloc (vg_replace_malloc.c:309)
      ==12345==    by 0x4005A0: main (myprogram.c:10)
      ```
    - "definitely lost" -> memory allocated but never freed (malloc without free)
    - check stack trace -> see where you allocated memory -> add free at the right spot
- SUMMARY SECTION
    - ```
      ==12345== LEAK SUMMARY:
      ==12345==    definitely lost: 8 bytes in 1 blocks
      ==12345==    indirectly lost: 0 bytes in 0 blocks
      ==12345==      possibly lost: 0 bytes in 0 blocks
      ==12345==    still reachable: 0 bytes in 0 blocks
      ```
    - focus on "definitely lost" -> these must be freed
    - "still reachable" is usually okay