Lecture 13: Git Objects, Compression Algorithms, and Low-Level C Concepts

1. Git Internal Object Representation

1.1 Git Directory Structure

- Git repositories contain a `.git/` subdirectory.
- `.git/objects/` stores Git objects as files, named by SHA-1 hashes.
- The `.git/objects` directory contains subdirectories named by the first two characters of the SHA-1 hash, each containing files named by the remaining 38 characters.

ASCII Diagram: Git Object Storage

```
.git/
  objects/
    6d/
      cb09e...   # File containing the object data
    7a/
      1f2e3...
    ...
```

1.2 Git Object Types and File Naming

- Git objects include:
  - `blob`: file data
  - `tree`: directory structure
  - `commit`: snapshot and metadata
  - `tag`: annotated reference to a commit
- Each object is named by a 40-character SHA-1 hexadecimal string.
  - First 2 characters: directory name.
  - Remaining 38 characters: file name inside that directory.
- Example:

  ```
  SHA-1: 6dcb09e... (40 hex digits)
  Directory: 6d/
  File: cb09e...
  ```

1.3 Blob Format

- Git objects include: `blob`, `tree`, and `commit`. Focus here: `blob`.
- Format: `blob <size><null byte><binary contents>`
  - ASCII "blob", then a space.
  - Decimal count of number of bytes (e.g., `196`).
  - Null byte (`\0`), ASCII value 0.
  - Then `<size>` bytes of content.
- **Why include size in header?**
  - Blob format is self-describing, not dependent on external bookkeeping.
  - Useful for packed structures.
- **Why use decimal for length?**
  - Makes blobs more readable for debugging.
  - Not worth the complexity to use binary for a few bytes of savings.

Example: Creating and Inspecting a Blob

```
echo "hello world" > file.txt
git hash-object file.txt  # Creates a blob and prints its SHA-1
SHA=($(git hash-object -w file.txt))
# Inspect the blob:
git cat-file -p $SHA
```

1.4 Compression in Git

- Git compresses objects using Zlib (same as Gzip).
- When examining files directly, decompression is required.

---

2. Limits and Theory of Data Compression

2.1 Fundamental Limits

- No "universal" compression algorithm can always reduce file size for all inputs.
- **Information-theoretic argument (Pigeonhole Principle):**
    - If input has 2^N possible values, you need N bits to represent all possibilities.
    - If you try to compress every possible input, some must get larger (since there aren't enough shorter outputs for all inputs).
- **Compression only works when input has redundancy or patterns.**
- **Compression behavior:**
    - Highly redundant text: major size reduction.
    - Random bytes: may get larger due to headers.

Example: Why Random Data Can't Be Compressed

Suppose you have all possible 8-bit values (0-255). Any mapping to fewer than 8 bits would require collisions (loss of information), so true random data cannot be compressed.

---

3. Gzip/Zlib Compression Algorithms

- Gzip uses two main ideas:
    1. **Huffman Coding** (1950s)
    2. **Dictionary Compression** (1970s)
- These are often combined (as in DEFLATE, used by gzip/zlib).

---

4. Huffman Coding

4.1 Overview

- Assigns variable-length bit strings to symbols based on frequency.
- Frequent symbols get shorter codes.
- Optimal for symbol-by-symbol compression with known probabilities.

4.2 Example: Symbol Probabilities

Character	Frequency
' ' (space)	0.2
e	0.06
t	less than e
z/q	very rare

4.3 Encoding Table Example

Symbol	Encoding
space	00
e	01
t	10
q	11100101

- **Prefix-free property:** No code is a prefix of another.

4.4 Huffman Tree Construction

- Start with all symbols as leaves, ordered by frequency.
- Merge two least frequent symbols at each step.
- Repeat until one node (the root) remains.
- Path from root to leaf = codeword for symbol.
- For 256 leaves (bytes): 255 internal nodes.

Step-by-Step Example: Huffman Tree

Suppose we have symbols A (5), B (9), C (12), D (13), E (16), F (45):

```
Step 1: Combine A(5) + B(9) = 14
Step 2: Combine C(12) + D(13) = 25
Step 3: Combine 14 + E(16) = 30
Step 4: Combine 25 + F(45) = 70
Step 5: Combine 30 + 70 = 100
```

ASCII Diagram:

```
        [100]
       /    \
    [30]    [70]
   /   \    /  \
[14] [16] [25] [45]
/  \      /  \
A   B    C    D
```

4.5 Optimality & Limitations

- **Pros:** Minimizes total bits if symbol probabilities are known and fixed.
- **Cons:**
  - Sender and receiver must share the encoding table.
  - Table size adds overhead.
  - Inflexible if input corpus changes (e.g., language switch mid-text).

---

5. Adaptive Huffman Coding

5.1 Overview

- Sender and receiver start with a trivial, balanced Huffman tree.
- Tree is updated dynamically as input is processed.
- No need to transmit the full table upfront.
- Rarely used in practice due to speed and memory tradeoffs; more common in academic settings.

5.2 Process

1. Start with uniform tree (equal frequency for all symbols).
2. Transmit first few symbols in full (e.g., 8-bit bytes).
3. After each symbol:
   - Update tree.
   - Both sender and receiver maintain identical updated trees.

Example: Tree Evolution

```
Initial tree: all symbols equal frequency
After 'A': 'A' node frequency increases
After 'B': 'B' node frequency increases
Tree structure changes as frequencies change
```

5.3 Advantages

- Only one pass over input required.
- Tree adapts to input statistics over time.
- Handles changing symbol distributions (e.g., English to Portuguese).

5.4 Trade-Offs

- Inefficient at the start (no compression until tree adapts).
- Slightly suboptimal compared to full-table Huffman.
- Sender/receiver must synchronize tree updates exactly.
- Requires extra RAM for tree (but small for byte-oriented data).

---

6. Dictionary Compression

6.1 Basic Idea

- Replace frequent sequences of bytes (e.g., words) with short codes.
- Instead of encoding each byte, encode entire sequences.

6.2 Dictionary Setup

- Build a dictionary mapping sequences to indexes.

```
Index | Value
------|-------------------
1     | "the "
2     | "and"
3     | "understandable"
```

- Data stream encodes sequences as index references (e.g., "understandable" → index 3, 16-bit code).

6.3 Issues & Considerations

1. **Dictionary size:**
   - Full English dictionary > 100,000 entries.
   - If too large, can't fit in 16 bits.
2. **Non-word characters:**
   - Dictionary includes all common byte sequences, not just words.
3. **Language switching:**
   - Static dictionary may not compress well if input language changes.

---

7. Adaptive Dictionary Compression (Sliding Window)

7.1 Problem with Static Dictionaries

- Non-adaptive method requires two passes: build dictionary, then encode.
- Not ideal for large files or streaming data.

7.2 Adaptive Approach

- Uses a **sliding window** dictionary (e.g., 64KiB of recent data).
- Sender and receiver both maintain a history of previous N bytes.
- Dictionary dynamically reflects recent input.
- Format for reference:

```
{Offset (16 bits), Length (8 bits)}
```

ASCII Diagram: Sliding Window Compression

```
Input Stream:  ... | W | I | N | D | O | W |  ...
                   <------ Sliding Window ------>
Current Position:   ^
```

The sliding window moves forward as new data is processed, always containing the most recent N bytes.

Step-by-Step Example: Adaptive vs. Static Dictionary

Suppose the input is: `the cat and the dog`

- **Static Dictionary:**
  - Pre-built dictionary: {1: "the", 2: "cat", 3: "dog"}
  - Encoded as: [1] [2] [and] [1] [3]
- **Adaptive (Sliding Window):**
  - Start with empty window.
  - As "the" is seen, it's added to the window.
  - When "the" appears again, it's referenced by offset/length in the window.

    - "cat" and "dog" are also added as they appear.

Example: LZ77 (Sliding Window) Encoding

Suppose the input is: `abcabcabc`

- First "abc": output as raw bytes
- Second "abc": reference offset=3, length=3
- Third "abc": reference offset=3, length=3

Table: Static vs. Adaptive Dictionary Compression

Feature	Static Dictionary	Adaptive (Sliding Window)
Dictionary	Pre-built, fixed	Built dynamically as input read
Passes Needed	Two (build, then encode)	One (encode as you go)
Handles New Data	Poorly (if not in dict)	Well (adapts to new patterns)
Memory Usage	Fixed (dict size)	Sliding window size
Use Case	Known, repetitive data	Streaming, unknown data

7.3 Compression Technique

1. Start with empty dictionary.
2. At each input point:
    - Find longest match in dictionary.
    - Emit (offset, length).
3. If no match:
    - Emit raw byte.

7.4 Search Optimization

- Naive search: $O(n^2)$.
- Efficient implementations use tries or suffix trees ($O(n \log n)$).

7.5 Limitations

- If a matching sequence is not in the window, must send as raw byte.
- Random data compresses poorly.
- Sensitive to data loss—losing sync causes downstream errors.
- More RAM can improve compression, but at a cost.

---

8. C vs. C++: What C Leaves Out

8.1 Simplified View: C = C++ - High-Level Features

Feature	C++	C
Classes & Objects	Yes	No
Polymorphism	Yes	No
Encapsulation	Yes	Limited
Inheritance	Yes	No
Abstract Data Types	Yes	Less supported
Static Members	Yes	Not in structs
Namespaces	Yes	No
Function Overloading	Yes	No
Exceptions	Yes	No (discouraged)
New/Delete	Yes	No (use malloc/free)
`cin`/`cout`	Yes	No (use `stdio.h`)

- In C, an "object" is just a piece of memory with a type.
- You still have `struct`, but not `class`.

- No built-in memory allocation operators (`new`, `delete`); use `malloc`/`free`.
- No namespaces; must manage names manually.
- No function overloading; use unique names.
- Exception handling exists but is discouraged.

8.2 Low-Level Control in C

- C provides direct access to memory via pointers and pointer arithmetic.
- Manual memory management: allocate with `malloc`, free with `free`.
- No automatic garbage collection.
- Allows direct manipulation of hardware and system resources.

Table: Memory Management in C vs. C++

Feature	C++ (Modern)	C
Manual Allocation	`new`/`delete`	`malloc`/`free`
Automatic (RAII)	Yes (constructors)	No
Smart Pointers	Yes (`std::unique_ptr`, etc.)	No
Garbage Collection	No (but can use libs)	No

---

9. C Compilation Process

9.1 Steps

Step	Input	Output	GCC Flag
Preprocessing	.c	.i	`-E`
Compilation	.i	.s	`-S`
Assembly	.s	.o	`-c`
Linking	.o	(executable)	-o (with `gcc`)

ASCII Flowchart: C Compilation Pipeline

```
[Source.c] --(Preprocessing: -E)--> [Source.i] --(Compilation: -S)--> [Source.s] --(Assembly: -c)-->
[Source.o] --(Linking: -o)--> [Executable]
```

9.2 Example

```
#define INTMAX 2147483647
int main() { return INTMAX; }
```

- Preprocessing expands macros:

```
int main() { return 2147483647; }
```

- Compilation to assembly:

```
main:
    movl $2147483647, %eax
    ret
```

- Linking resolves library calls (e.g., `abs`).

---

10. Process Management & Developer Tools

10.1 Process Management Commands

Command	Purpose
`ps`	Show running processes
`ps -ef`	All processes (full view)
`ps -ejH`	Show hierarchy of processes
`kill <PID>`	Send default signal (TERM)
`kill -9 <PID>`	Force kill using SIGKILL signal
`kill -STOP`	Pause process
`kill -CONT`	Resume process

Table: Common Signals

Signal	Number	Description
SIGTERM	15	Graceful termination
SIGKILL	9	Forceful kill (cannot be caught)
SIGSTOP	19	Pause process
SIGCONT	18	Resume process

10.2 Debugging Tools

- `time <command>`: Shows real/user/sys time for a command.
- `strace <command>`: Traces system calls (kernel interactions).
- `ltrace <command>`: Traces library calls (e.g., `printf`, `malloc`).
- `valgrind <command>`: Memory error detection (invalid access, leaks, uninitialized use). Slower but thorough.

Example: Using Valgrind

Suppose you have a buggy C program `buggy.c`:

```c
int main() {
    int *p = malloc(sizeof(int));
    return *p; // uninitialized read
}
```

Compile and run with valgrind:

```
gcc buggy.c -o buggy
valgrind ./buggy
```

Valgrind output will report use of uninitialized value.

Summary

This lecture explored Git's internal object model, especially blob storage and format, and explained how Git compresses data using Gzip/Zlib. It covered the theory and practical limits of compression, the two main strategies in Gzip (Huffman coding and dictionary compression), and both static and adaptive versions of these algorithms. The lecture then transitioned to low-level C programming, contrasting it with C++ by highlighting omitted features for greater control and performance. It included a walk-through of the C compilation pipeline and the tools developers use to observe and manage processes, debug, and track system and memory behavior. These concepts are essential for understanding system-level programming and preparing for assignments that work closely with system internals.