Self-Study Summary Collection

Volume 2

Computer Sciense

Anton Augustsson

August 31, 2025

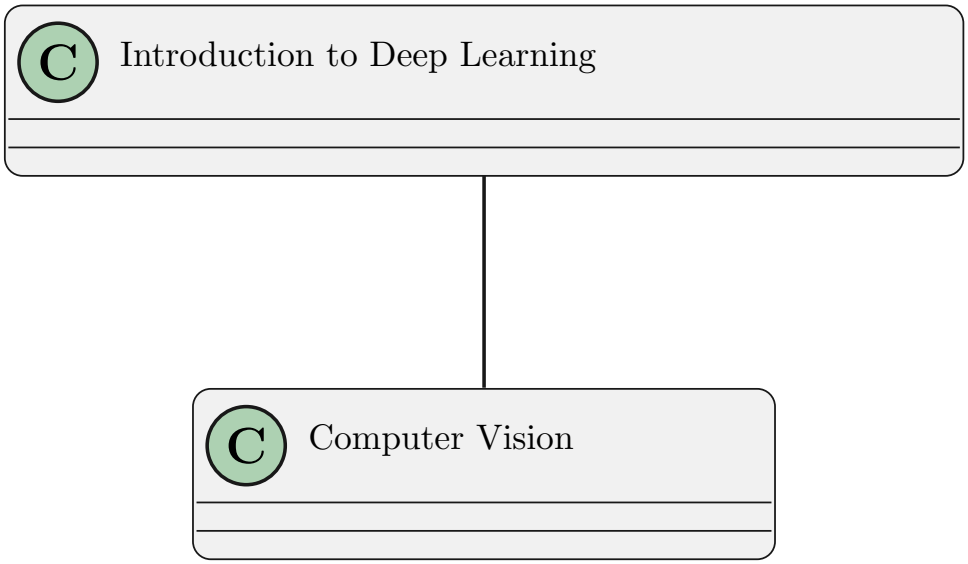
Contents

Chapter 1

Study Plan

There are several course topics summarized in this document. They are related in someways but can be regarded as isolated and there for have to correlation to between topics. Each of the courses is summarized in each own chapter and is mostly based on a course from MIT, Yale, or Stanford. MIT in particular has a great selection of open courses in various scientific topics. Each of the chapters starts with a general info of the course it is based on and various relevant links for the course material. The chapters are in chronological order the courses was taken and as no relation to topics.

1.1 Computer Science



Chapter 2

Embedded C++

Introduction

C++ is widely used in embedded systems programming due to its balance of high-level features and low-level control. It provides object-oriented programming while maintaining efficiency, critical for resource-constrained environments.

2.1 Key Features of C++ for Embedded Systems

- **Low-Level Memory Control:** Access to pointers, memory manipulation, and hardware registers.
- **Object-Oriented Programming (OOP):** Encapsulation, inheritance, and polymorphism enhance modularity and code reuse.
- **Templates:** Generic programming with minimal overhead.
- **Inline Functions:** Eliminate function call overhead for performance-critical code.
- **RAII (Resource Acquisition Is Initialization):** Ensures safe resource management.
- **Namespaces:** Prevent naming conflicts in large projects.

2.2 Essential C++ Concepts for Embedded Systems

2.2.1 Memory Management

Embedded systems often lack dynamic memory (heap) due to limited resources. C++ allows manual memory management with careful use of:

- Stack-based allocation (preferred).
- Avoiding the use of dynamic memory (e.g., `new`/`delete`).

2.2.2 Hardware Access

Accessing hardware registers and memory-mapped I/O is crucial. C++ provides:

```
#define GPIO_PORT  (*(volatile uint32_t*)0x40020C00)
void setPinHigh() {
    GPIO_PORT |= (1 << 5); // Set bit 5
}
```

2.2.3 Inline Functions

Inline functions eliminate the overhead of function calls:

```
inline void setBit(volatile uint32_t& reg, uint8_t pos) {
    reg |= (1 << pos);
}
```

2.2.4 Classes and Encapsulation

Encapsulation simplifies peripheral drivers:

```
class GPIO {
public:
    GPIO(uint32_t* port) : port_(port) {}
    void setPin(uint8_t pos) { *port_ |= (1 << pos); }
    void clearPin(uint8_t pos) { *port_ &= ~(1 << pos); }
private:
    volatile uint32_t* port_;
};

GPIO gpioA(reinterpret_cast<uint32_t*>(0x40020C00));
gpioA.setPin(5);
```

2.3 Best Practices for Embedded C++

- Use **const** and **constexpr** to avoid runtime overhead.
- Avoid exceptions and RTTI (Run-Time Type Information) due to memory and performance costs.
- Prefer fixed-size types (e.g., `uint8_t`, `uint16_t` from `<cstdint>`).
- Use static memory allocation to avoid fragmentation.
- Leverage compiler optimizations (e.g., `-O2`, `-O3` flags).
- Use lightweight libraries or standard subsets (e.g., Embedded STL).

2.4 Comparison: C vs C++ for Embedded Systems

Feature	C	C++
Encapsulation	Manual (structs)	Classes and objects
Generic Programming	Macros	Templates
Error Handling	Return codes	Exceptions (avoid in embedded)
Memory Management	Manual	RAII, smart pointers
Namespace Management	None	Namespaces
Code Reusability	Limited	OOP, inheritance

2.5 Conclusion

C++ offers significant advantages for embedded systems, combining performance, safety, and modern programming techniques. By adhering to best practices, developers can write clean, efficient, and maintainable embedded software using C++.

2.6 Compilation

C++ source code goes through multiple stages before becoming an executable program. Understanding the compilation process helps developers write better code, debug efficiently, and optimize performance.

The C++ compilation process can be broken down into the following stages:

1. Preprocessing
2. Compilation
3. Assembly
4. Linking

Each of these stages is discussed in detail below.

2.7 Stages of the C++ Compilation Process

2.7.1 1. Preprocessing

The preprocessor processes the source code before actual compilation. It handles all directives starting with `#`, such as:

- `#include`: Include header files.
- `#define`: Define macros.
- `#ifdef`, `#ifndef`, `#endif`: Conditional compilation.

The preprocessor outputs an expanded version of the source file, which is sent to the next stage. Example:

```
#include <iostream>
#define PI 3.14

int main() {
```

```
    std::cout << "Value of PI: " << PI << std::endl;
    return 0;
}
```

After preprocessing:

```
int main() {
    std::cout << "Value of PI: " << 3.14 << std::endl;
    return 0;
}
```

Command to see preprocessed output:

```
g++ -E main.cpp -o main.i
```

2.7.2 2. Compilation

The compiler takes the preprocessed file and translates it into **assembly language**, which is specific to the target architecture.
Example:

```
g++ -S main.i -o main.s
```

This produces an assembly file `main.s`. The assembly file contains low-level instructions that can be understood by the assembler.
Example (Snippet of `main.s`):

```
.LC0:
    .string "Value of PI: %f"
main:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
```

2.7.3 3. Assembly

The assembler converts the assembly code into **machine code**, which consists of binary instructions the CPU can understand. The output of this stage is an **object file** (`.o` or `.obj`).
Example:

```
g++ -c main.s -o main.o
```

The `main.o` file contains machine code but is not yet executable. At this stage, the object file may contain unresolved references to external functions or libraries.

2.7.4 4. Linking

The linker combines the object file with:

- Other object files (if any).
- Standard libraries (e.g., `libstdc++` for C++ Standard Library).

It resolves function calls and creates an executable file.
Example:

```
g++ main.o -o main
```

The final executable, `main`, can now be run:

```
./main
```

2.8 Summary of Stages

The C++ compilation pipeline can be summarized as follows:

- **Source Code:** `.cpp`
- **Preprocessing:** Expands macros, includes headers → `.i`
- **Compilation:** Translates to assembly code → `.s`
- **Assembly:** Converts to machine code → `.o`
- **Linking:** Combines object files and libraries → Executable

2.9 Compiler Tools

Here are some common tools used for the compilation process:

- **GCC/G++**: The GNU Compiler Collection.
- **Clang**: A compiler based on LLVM.
- **MSVC**: Microsoft Visual C++ Compiler for Windows.

2.10 Conclusion

Understanding the C++ compilation process helps developers write more efficient code and troubleshoot issues like:

- Missing header files during preprocessing.
- Syntax or type errors during compilation.
- Linking errors due to unresolved references.

By using tools like `g++` or `clang++`, you can observe each step of the compilation pipeline.

2.11 Pointers in C++

Pointers are fundamental in C++ for managing memory, accessing data, and working with low-level hardware. While raw pointers provide direct access to memory, they can lead to issues like memory leaks. Modern C++ introduces smart pointers to manage memory safely and automatically.

2.12 Raw Pointers in C++

A pointer is a variable that stores the memory address of another variable. Raw pointers require manual memory management.

2.12.1 Syntax and Example

```cpp
#include <iostream>
int main() {
    int x = 10;
    int* ptr = &x; // Pointer stores the address of x

    std::cout << "Value: " << *ptr << std::endl; // Dereference pointer
    std::cout << "Address: " << ptr << std::endl;

    return 0;
}
```

2.12.2 Dynamic Memory Allocation

Raw pointers can allocate memory dynamically using `new` and release it using `delete`.

```cpp
#include <iostream>
int main() {
    int* ptr = new int(5); // Allocate memory on the heap
    std::cout << "Value: " << *ptr << std::endl;

    delete ptr; // Free the allocated memory
    return 0;
}
```

Caution:

- Forgetting to call `delete` leads to memory leaks.
- Dereferencing null or dangling pointers leads to undefined behavior.

2.13 Smart Pointers in Modern C++

Smart pointers, introduced in C++11, are wrappers around raw pointers that provide automatic memory management. They are defined in the `<memory>` header.

2.13.1 Types of Smart Pointers

1. `std::unique_ptr`: A pointer that has sole ownership of an object.
2. `std::shared_ptr`: A pointer that shares ownership of an object with other smart pointers.
3. `std::weak_ptr`: A non-owning pointer that observes `std::shared_ptr`.

2.13.2 `std::unique_ptr`

`std::unique_ptr` ensures single ownership of a resource. When the pointer goes out of scope, the memory is automatically released.

```cpp
#include <iostream>
#include <memory>

```

```cpp
int main() {
    std::unique_ptr<int> ptr = std::make_unique<int>(10); // Create unique_ptr
    std::cout << "Value: " << *ptr << std::endl;

    // Unique ownership; cannot be copied
    // std::unique_ptr<int> ptr2 = ptr; // Error!

    return 0;
} // Memory is released automatically
```

2.13.3 std::shared_ptr

`std::shared_ptr` allows multiple smart pointers to share ownership of an object. It uses reference counting to track ownership.

```cpp
#include <iostream>
#include <memory>

int main() {
    std::shared_ptr<int> ptr1 = std::make_shared<int>(20); // Create shared_ptr
    std::shared_ptr<int> ptr2 = ptr1; // Shared ownership

    std::cout << "Value: " << *ptr1 << ", " << *ptr2 << std::endl;
    std::cout << "Reference Count: " << ptr1.use_count() << std::endl;

    return 0;
} // Memory is released when the last shared_ptr goes out of scope
```

2.13.4 std::weak_ptr

`std::weak_ptr` observes a `std::shared_ptr` without contributing to the reference count. It is useful for breaking cyclic references.

```cpp
#include <iostream>
#include <memory>

int main() {
    std::shared_ptr<int> shared = std::make_shared<int>(30);
    std::weak_ptr<int> weak = shared; // Observes shared_ptr

    if (auto ptr = weak.lock()) { // Check if resource still exists
        std::cout << "Value: " << *ptr << std::endl;
    } else {
        std::cout << "Resource no longer exists." << std::endl;
    }

    return 0;
}
```

2.14 Comparison of Pointers

Pointer Type	Ownership	Memory Management
Raw Pointer	None	Manual (new/delete)
`std::unique_ptr`	Single Ownership	Automatic
`std::shared_ptr`	Shared Ownership	Reference Counting
`std::weak_ptr`	Observing (Non-owning)	No ownership

2.15 Best Practices for Pointers

- Prefer `std::unique_ptr` for single ownership.
- Use `std::shared_ptr` when multiple owners are needed.
- Avoid raw pointers unless working with legacy code or hardware.
- Use `std::weak_ptr` to break cyclic references.
- Avoid manual memory management (use smart pointers instead).

2.16 Conclusion

Pointers are a powerful tool in C++ for managing memory and accessing resources. While raw pointers offer flexibility, they come with risks like memory leaks. Smart pointers in modern C++ provide safer alternatives for automatic and efficient memory management.

2.17 Lessons from Scott Meyers's Effective C++

Effective C++ by Scott Meyers is a classic book that provides practical guidelines and principles for writing robust, maintainable, and efficient C++ programs. This document summarizes the key guidelines into a concise format for quick reference.

The book is organized into 50 guidelines across several key areas of C++ programming.

2.18 Design and Declarations

1. **View C++ as a federation of languages.**
   Understand C++ as a mix of procedural programming, object-oriented programming, templates, and the STL.

2. **Prefer consts, enums, and inlines to `#define`.**
   Use `const` for constants, `inline` for functions, and `enum` for integral constants instead of macros.

3. **Use const whenever possible.**
   Declare objects, pointers, and member functions as `const` to improve safety and clarify intent.

4. **Prefer `++i` over `i++` for iterators and other user-defined types.**
   Pre-increment is generally more efficient than post-increment for iterators and custom types.

5. **Declare a virtual destructor in polymorphic base classes.**
   Without a virtual destructor, deleting derived objects through a base class pointer causes undefined behavior.

6. **Avoid returning handles to object internals.**
   Don't expose raw pointers or references to private data members as it compromises encapsulation.

2.19 Constructors, Destructors, and Assignment Operators

1. **Prevent resource leaks in constructors.**
   Use smart pointers or RAII (Resource Acquisition Is Initialization) to manage resources.

2. **Explicitly disallow the use of compiler-generated functions you do not want.**
   Declare unwanted copy constructors and assignment operators as `private` or delete them in C++11.

3. **Prefer initialization to assignment in constructors.**
   Initialize member variables in the member initializer list to improve performance.

4. **Handle self-assignment in operator=.**
   Ensure assignment operators check for self-assignment to avoid corruption of the object state.

2.20 Resource Management

1. **Use objects to manage resources.**
   Implement RAII for resource management (e.g., smart pointers for dynamic memory).

2. **Avoid using raw pointers.**
   Use smart pointers such as `std::unique_ptr` or `std::shared_ptr` instead of raw pointers to manage dynamic memory.

3. **Minimize resource acquisition in constructors.**
   Use helper classes or functions to avoid constructor failures due to resource allocation.

2.21 Templates and Generic Programming

1. **Understand the behavior of templates.**
   Templates are instantiated with types at compile time; understand how type deduction works.

2. **Prefer function objects over functions for STL algorithms.**
   Function objects (functors) can store state and inline better than function pointers.

3. **Familiarize yourself with STL containers and iterators.**
   Use the right container for your task (e.g., `vector`, `list`, `map`).

4. **Write generic code with templates.**
   Use templates to write code that works for a wide variety of types.

2.22 Inheritance and Object-Oriented Design

1. **Distinguish between public and private inheritance.**
   Use public inheritance for "is-a" relationships; use private inheritance for implementation reuse.

2. **Avoid slicing.**
   Always pass objects by reference or pointer to avoid slicing when working with polymorphic classes.

3. **Prefer composition over inheritance.**
   Composition provides more flexibility and avoids the complexities of inheritance.

2.23 Miscellaneous Guidelines

1. **Avoid unnecessary inclusion of header files.**
   Use forward declarations when possible to reduce compilation dependencies.

2. **Use inline functions carefully.**
   Inline functions can improve performance, but overuse can lead to code bloat.

3. **Minimize casting.**
   Avoid explicit casts; use safer alternatives like `dynamic_cast`, `static_cast`, or C++11's `auto`.

4. **Pay attention to compiler warnings.**
   Enable and resolve all compiler warnings for better code quality.

2.24 Conclusion

Effective C++ offers valuable, practical advice for writing clean, efficient, and maintainable C++ code. Following these principles can help developers avoid common pitfalls, manage resources effectively, and utilize the full power of C++.

2.25 Summary of MISRA C++:2008 Guidelines

MISRA (Motor Industry Software Reliability Association) provides coding guidelines for developing reliable, maintainable, and safe C++ code. Originally designed for embedded and automotive systems, these rules apply to all safety-critical systems.

This document summarizes key MISRA C++ rules and organizes them into categories for clarity.

2.26 General Rules

1. **Avoid implementation-defined behavior.**
   Code must not rely on behavior that varies between compilers or systems.

2. **Avoid undefined behavior.**
   Writing code that leads to undefined behavior (e.g., dereferencing null pointers) is strictly prohibited.

3. **Do not rely on unspecified behavior.**
   Code behavior that depends on compiler or execution order should be avoided.

2.27 Language Restrictions

1. **No use of non-standard libraries.**
   Only standard C++ libraries and approved third-party libraries should be used.
2. **Dynamic memory allocation is forbidden.**
   Avoid `new`, `delete`, `malloc`, and `free` due to fragmentation and reliability concerns.
3. **Avoid implicit conversions.**
   Implicit type conversions can result in data loss or unexpected behavior.
4. **No use of uninitialized variables.**
   Always initialize variables before use to avoid undefined behavior.
5. **No usage of `goto`.**
   The `goto` statement reduces code readability and maintainability.

2.28 Object-Oriented Programming Rules

1. **Use polymorphism safely.**
   Virtual functions in base classes must be declared with a virtual destructor.
2. **Do not slice derived class objects.**
   Pass objects by pointer or reference, not by value, to avoid object slicing.
3. **Use `explicit` for single-argument constructors.**
   Prevent implicit conversions that occur from single-argument constructors.
4. **Avoid multiple inheritance.**
   Multiple inheritance increases complexity and risks ambiguity.
5. **Do not hide inherited names.**
   Ensure that base class functions are not unintentionally hidden by derived class functions.

2.29 Templates and STL Rules

1. **Avoid template instantiation errors.**
   Templates must compile cleanly without instantiation errors for all intended types.
2. **Use STL carefully.**
   Standard Template Library (STL) containers and algorithms must be used only when their behavior is well-understood and compliant with MISRA guidelines.
3. **Avoid exceptions.**
   Exception handling (e.g., `throw`, `try`, `catch`) should be minimized or avoided, as it adds runtime overhead and complexity.
4. **No partial specialization.**
   Partial specialization of templates is forbidden, as it can lead to ambiguous or hard-to-maintain code.

2.30 Resource Management

1. **Ensure proper resource cleanup.**
   Use RAII (Resource Acquisition Is Initialization) to manage resources and avoid leaks.

2. **Avoid resource exhaustion.**
   Ensure that resource allocation (e.g., files, memory) is properly bounded.

3. **No memory leaks.**
   Avoid situations where dynamically allocated memory is not deallocated.

4. **Use smart pointers instead of raw pointers.**
   Prefer `std::unique_ptr` and `std::shared_ptr` to manage dynamic memory.

2.31 Concurrency and Synchronization

1. **Avoid data races.**
   Ensure that concurrent threads do not access shared resources without proper synchronization.

2. **Use thread-safe mechanisms.**
   Use mutexes, locks, and other thread-safe constructs to synchronize access to shared resources.

3. **Minimize the use of global variables.**
   Global variables lead to hidden dependencies and make code harder to maintain.

2.32 Error Handling

1. **Use error codes instead of exceptions.**
   Return error codes for predictable error handling instead of relying on exceptions.

2. **Check return values of all functions.**
   Always validate the return value of functions that may fail.

3. **Use assertions sparingly.**
   Assertions (`assert`) should only be used to check invariants, not for error handling.

Conclusion

MISRA C++ rules ensure safe, reliable, and maintainable code for critical systems. By following these guidelines, developers can avoid common pitfalls, undefined behavior, and performance issues while ensuring compliance with industry standards.

Chapter 3

CMake

3.1 Introduction

CMake is a cross-platform, open-source build system generator. It generates native build configurations (e.g., `Makefiles`, `ninja`, Visual Studio projects) to build, test, and package software. It supports both in-source and out-of-source builds and is highly configurable.

3.2 Core Concepts

- **CMakeLists.txt:** The main configuration file used to define the build process.
- **Targets:** Executables or libraries generated by the build process.
- **Generator:** A tool that CMake uses to produce native build scripts (e.g., `Unix Makefiles`, `Ninja`).
- **Out-of-source Build:** Keeping build artifacts separate from source code.

3.3 Basic Commands in `CMakeLists.txt`

`cmake_minimum_required(VERSION X.Y)` Specifies the minimum version of CMake required.

`project(NAME LANGUAGES ...)` Defines the project name and supported languages (e.g., C, C++).

`add_executable(target source1 ...)` Defines an executable target.

`add_library(target source1 ...)` Defines a library target.

`target_include_directories(target ...)` Adds include directories to a target.

`target_link_libraries(target ...)` Links libraries to a target.

`find_package(NAME ...)` Locates external dependencies (e.g., Boost, OpenCV).

`install(...)` Specifies installation rules for targets or files.

`option(NAME "Description" DEFAULT)` Defines a build-time configuration option.

`if(condition)` Adds conditional logic.

`set(NAME VALUE)` Sets variables for configuration or build logic.

3.4 Example CMake Project

3.4.1 Directory Structure

3.4.2 CMakeLists.txt

```
cmake_minimum_required(VERSION 3.15)
project(MyProject LANGUAGES CXX)

# Set C++ standard
set(CMAKE_CXX_STANDARD 17)

# Add include directories
include_directories(include)

# Add executable target
add_executable(my_app src/main.cpp)
```

3.5 Build Steps

1. Create a build directory:

   ```
   mkdir build && cd build
   ```

2. Run CMake to configure the project:

   ```
   cmake ..
   ```

3. Build the project:

   ```
   cmake --build .
   ```

3.6 Advanced Features

3.6.1 Variables and Options

- Use `set()` to define variables:

  ```
  set(VARIABLE_NAME value)
  ```

- Define options with `option()`:

  ```
  option(BUILD_TESTS "Build the test suite" ON)
  ```

Dependency Management

- Use `find_package()` to locate external dependencies:

```
find_package(Boost REQUIRED COMPONENTS system filesystem)
target_link_libraries(my_app Boost::system Boost::filesystem)
```

- Use `FetchContent` to manage dependencies:

```
include(FetchContent)
FetchContent_Declare(
    googletest
    URL https://github.com/google/googletest/archive/release-1.10.0.zip
)
FetchContent_MakeAvailable(googletest)
```

Cross-Compilation

CMake supports cross-compilation by specifying a toolchain file with `-DCMAKE_TOOLCHAIN_FILE`:

```
cmake -DCMAKE_TOOLCHAIN_FILE=toolchain.cmake ..
```

Generators

CMake provides support for different build tools via generators:

- `Unix Makefiles`: Generates `Makefiles` for `make`.
- `Ninja`: Generates build files for `ninja`.
- `Visual Studio`: Generates Visual Studio project files.

Choose a generator with the `-G` flag:

```
cmake -G "Ninja" ..
```

Best Practices

- Use **out-of-source builds** to keep build artifacts separate from source code.
- Modularize your build system with subdirectories and subprojects.
- Always specify a `CMakeLists.txt` in each subdirectory.
- Use `target_include_directories()` instead of global include directories.

References

- CMake Documentation
- Modern CMake Guide

Chapter 4

Git

Abstract

Git is a distributed version control system designed to handle everything from small to large projects efficiently. This document provides a detailed and fundamental explanation of how Git works, including its architecture, core concepts, and the mechanics behind its operations.

4.1 Introduction

Git is a powerful tool for managing source code and collaborating on software development projects. Unlike centralized version control systems, Git is distributed, meaning each developer has a complete copy of the repository, including its history. This section introduces the architecture of Git and explains its core concepts.

4.2 Key Concepts in Git

4.2.1 Snapshots vs. Deltas

Unlike many version control systems that track changes (deltas) between file versions, Git records the state of the repository at each commit. Each commit in Git is a snapshot of the project at a given point in time. If a file remains unchanged, Git stores a reference to the previous file rather than duplicating it.

4.2.2 Three States in Git

Git operates on three primary states:

- **Working Directory:** The local directory where files are edited.
- **Staging Area:** An index where changes are staged before committing.
- **Repository:** The .git directory where committed snapshots and metadata are stored.

4.2.3 Git's Data Model

Git's data model is built on three core components:

- **Blobs:** Store the content of files.
- **Trees:** Represent directory structures and reference blobs.
- **Commits:** Point to trees and contain metadata, such as author, timestamp, and parent commits.

4.3 How Git Works

4.3.1 Commits and Hashes

Each commit in Git is identified by a unique SHA-1 hash, which is generated based on the commit's content, including:

- The tree object representing the project's directory structure.
- The parent commit(s).
- Metadata such as author, timestamp, and commit message.

This ensures that any change to a commit's content results in a completely new hash, maintaining integrity.

4.3.2 Branches

Branches are pointers to specific commits. The default branch in Git is `main` (formerly `master`).

- Creating a branch creates a new pointer to a commit.
- Switching branches updates the working directory to match the snapshot of the commit the branch points to.
- Merging combines changes from one branch into another.

4.3.3 Staging and Committing

1. **Staging:** Use `git add` to move changes to the staging area.
2. **Committing:** Use `git commit` to save a snapshot of staged changes. Git creates a commit object that references the current tree and parent commit(s).

4.3.4 Distributed Architecture

Each Git repository is complete and self-contained. This allows developers to work offline and perform all operations locally. Collaboration is achieved through pushing and pulling changes to/from remote repositories.

4.4 Git Workflow

4.4.1 Cloning a Repository

To start working on a project, clone the repository using:

```
git clone <repository_url>
```

4.4.2 Typical Workflow

A typical workflow in Git includes the following steps:

1. **Modify Files:** Make changes in the working directory.
2. **Stage Changes:** Use `git add` to stage the changes.
3. **Commit Changes:** Create a snapshot using `git commit`.
4. **Push Changes:** Share changes with a remote repository using `git push`.
5. **Pull Updates:** Fetch and integrate changes from the remote repository using `git pull`.

4.5 Common Git Commands

- `git status` - Check the status of files in the working directory and staging area.
- `git diff` - View changes in the working directory or staging area.
- `git log` - View the history of commits.
- `git branch` - Manage branches.
- `git merge` - Combine branches.
- `git rebase` - Reapply commits on top of another base.

4.6 Conclusion

Git is a robust and versatile version control system that underpins modern software development workflows. Its distributed architecture, snapshot-based model, and powerful branching capabilities make it an essential tool for developers.

References

- Git Official Documentation
- Atlassian Git Tutorials