C++ Prework

IMC

January 2025

CONTENTS

CHAPTER

ONE

INTRODUCTION

I hope you are as excited as we are about you joining IMC, and we're looking forward to having you in our Software Engineering Global Traineeship (colloquially known as Dev School).

Before we start the traineeship together, we want to ensure that you have a solid understanding of certain widely used open-source technologies. We have a limited amount of time available for the traineeship, so we want to be able to use that time for working directly with instructors to learn advanced topics and IMC-specific tooling. Therefore, it is essential that you spend time familiarizing yourself with these technologies prior to the start of the program. The attached documents describe the contents that you should focus on.

While we will provide you with recommended learning resources as a starting point, please note that independent research and exploration are expected from each of you. Self-directed learning is a crucial skill for success at IMC, and this is an opportunity to practice that!

For each technology, we have compiled a set of recommended learning resources that will serve as a great starting point for your studies. We have found these to be good introductory resources for covering fundamental concepts and providing practical examples to help grasp the essentials. If they do not fit your learning style, you are encouraged to find your own resources on Google, YouTube, or elsewhere. If you are already familiar with any material, feel free to skip the specific content.

Additionally, we are providing you with a checklist of concepts for each technology. This checklist will serve as a valuable tool to ensure that you cover the most important topics within each technology. It is highly recommended that you use this checklist as a guideline to gauge your progress and track your learning.

For some topics, we have prepared exercises to work through, and/or an exam to test your knowledge. Any exams are open-book and untimed - once again, the point is to help double-check your understanding and gauge your progress. However, when noted, please submit your answers via email.

It is very important that you are completely comfortable with these basics, as they are mostly taken for granted in the training.

If you have any questions or need further clarification, please don't hesitate to reach out to us. We are more than happy to assist you.

CHAPTER

TWO

GIT

Git is the dominant VCS in the tech community, and is used by all development teams at IMC.

Additionally, most teams use the Gerrit code review tool, which heavily relies on advanced Git features around rewriting history.

2.1 Recommended Reading Material

2.1.1 Pro Git

The following chapters of Pro Git contain the most useful material about using Git.

- 1. Getting Started
- 2. Git Basics
- 3. Git Branching
- 7. Git Tools
  - 7.1 Revision Selection
  - 7.2 Interactive Staging
  - 7.3 Stashing and Cleaning
  - 7.5 Searching
  - 7.6 Rewriting history
  - 7.7 Reset Demystified
  - 7.10 Debugging with Git

Chapters 4-6 cover Git Servers and Workflows that are applicable to working in open source and on Github, but not immediately relevant to work at IMC.

The other sections of Chapter 7 cover more advanced Git tools that are useful, but will not be used in Dev School.

2.2 Recommended Exercises

Learn Git Branching is a good practical exercise for learning about the most useful Git commands.

2.3 Checklist

2.3.1 Essential Concepts

You should have a solid understanding of the following concepts.

- Git Fundamental Elements
  - Commits
  - Branches
  - Tags
  - References
  - Remotes
  - Repositories
- Commit Workflow
  - Working Directory
  - Stage/Index
  - Stash
- Checking out history
- Rewriting history
- `git reset`

2.3.2 Practical Skills

You must be able to do the following:

- Git commit workflow
  - Stage change and create commits
  - See your uncommitted or staged changes
  - Throw away your un-committed changes
  - Stash and unstash changes, stash multiple changes at a time
- Exploring Git History
  - See the history of commits in a repository
  - See the commit message for any given commit?
  - See what changes went into any given commit, or what are the differences between two arbitrary commits
  - Look at the code from a previous state in the repository history
  - Figure out what commit a bug was introduced in

- Branching and tagging
  - Create, switch, merge and delete branches
  - Copy commits from one branch to another
  - Create and delete tags
- Git Remotes
  - Clone from, pull from, push to a remote
- Undoing
  - Undo a `git commit` (without losing your code).
  - Revert a commit so that the code is in the same state as if the commit wasn't done, but the commit is left in history.
  - Delete a commit so that it is gone from git history entirely.
- Rewriting history
  - Modify the commit message of the most recent commit.
  - Modify the commit message of a previous commit
  - Combine two commits into one commit
  - Split one commit into two commits
  - Reorder two commits
  - Delete a commit
  - Change a branch to point to a completely different commit
- Fixing mistakes
  - Undo a `reset`
  - Recover a "lost" commit

2.3.3 Advanced Git Concepts and Tools

You are **not** required to know anything about these.

- Submodules / Subtrees
- Sparse Checkouts
- Hooks
- Git internals
  - Objects: Blobs, Trees, etc.
  - Packfiles
- `git rerere`

2.4 Additional Material

- An Introduction To Git and Github - Introduction to Git, but in video format.
- Think Like (a) Git
- Confusing git terminology

2.5 Exam

The *Git Exam* can be used to evaluate your understanding of git.

2.5.1 Git Exam

Theory

Instructions

Answer the below questions with short (a few sentences) answers. Keep your answers as precise and concise as possible.

Send your answers as an email, with one blank line in between answers to each question, to Dev School Leads (devschoolleads@imc.com) with the subject "Pre-work Git theory exam submission".

---

1. What do `git add` and `git commit` do?
2. What is a commit in Git? What are the components of a commit?
3. What is a branch in Git? What is the purpose of branches?
4. What does it mean for a commit to have more than one parent?
5. How do you merge changes from branch A into branch B? What does this mean? What happens to each branch?
6. How do you cherry-pick a change onto branch A? What does this mean? What happens when you do this?
7. How do you rebase changes from branch `A` on top of branch `B`? What does this mean? What happens to each branch?
8. What is a 'merge conflict' in Git? What operations can result in a merge conflict? How are merge conflicts resolved?
9. What is a fast-forward merge?
10. What is a 'remote' in Git? What is 'origin'?
11. What is the difference between `git pull` and `git fetch`?
12. What is the difference between a tag and a branch?
13. What are the most common use cases for tags?
14. What is a "commit-ish"? What are a few examples of a "commit-ish"?
15. What is the difference between `HEAD` and a "head" (such as `refs/heads/master`)?
16. What is "Detached HEAD mode"? How do you get into detached HEAD mode? How do you get out of detached HEAD mode?

17. If you find a bug in code that you know was working at some point in the past, but isn't working now, how would you find the commit where a bug was introduced?
18. How do you check out the code for an arbitrary past commit so that you can debug it?
19. What does rewriting history mean? What is a typical reason you will rewrite history?
20. What is the difference between reverting a commit and deleting it?
21. What is the difference between `git checkout 29f5e54` and `git reset 29f5e54`?
22. How do you delete the most recent commit from a branch?
23. How do you delete an older commit from a branch's history?
24. When should you not rewrite history? What can go wrong when you modify history?
25. If you `git commit --amend` a commit, how can you recover the previous version of that commit?
26. What is the `reflog`? When would you use it?

Practical

Instrucitons

Clone the repository from https://github.com/imc-trading/devschool-git-exam and then complete the following steps.

When you are done, zip up your repository and send it, along with the answers to each question, with a blank line in between each answer, to Dev School Leads (devschoolleads@imc.com) with the subject "Pre-work Git practical exam submission".

1. Check out the branch `resume`. Change the commit message from "Add Wok Experience" to "Add Work Experience". What commands did you run?
2. Add this "Work Experience" item in "Resume.md":

```
### The Normal Brand

- Web Developer -- 2019-2021
  - Implemented OAuth login process to support Social Login.
  - Added web analytics
```

Commit it on the branch `resume` with the message "Add Normal Brand item". What command(s) did you run?

3. The commit on the branch `resume` with the message "fill edu" has some issues. Fix it like this: 1. Change the message to "Add content for Education section". 2. The commit has introduced some whitespaces at the end of two lines. Remove those. What command(s) did you run? Explain your actions.
4. The commit with the message "Add empty lines" is not useful. Remove it from the history. What command(s) did you run?
5. There is a branch called `skills`. How can you show the content of the `Resume.md` file on that branch without switching to the branch?
6. On the branch `skills` there is a commit called "Add skills". Add that commit on top of the branch `resume` without switching branches away from `resume`. That should create a conflict. Fix it by incorporating both the previous and the new changes. What commands and actions did you do?

7. In `Resume.md` there is a line `### Illinois State University`. How can you find out what is the last commit that changed/introduced that line? What is the message of that commit? What commands did you run?
8. Create a branch called `letter_of_intent` forking off from the branch `job_applications`. Create a file there called `letter.txt` with the content `I need this job, please!`. Commit this file on the `letter_of_intent` branch with the commit message `Add letter of intent`. Add another commit that adds this line to `letter.txt`: `Sincerely yours, Norman`. Commit it with the message `Sign letter`. What commands did you run?
9. Rebase the `letter_of_intent` onto the `resume` branch without including the commits on the `job_applications` branch. Basically add the commits `Add letter of intent` and `Sign letter` on top of `resume` without changing `resume` and point `letter_of_intent` to the last commit. Do not use `cherry-pick`. What command did you run?
10. Merge the branch `letter_of_intent` into the branch `resume`. What commands did you run? Explain the merge strategy taken.
11. Merge the `job_applications` branch into the branch `resume`. What commands did you run? What merge strategy was taken? What happened to the git history of the `resume` branch?

CHAPTER

THREE

UNIX BASICS

All of our deployment servers are UNIX environments and you will need to be able to interact with them using the command line in order to effectively deploy and manage your apps at IMC.

3.1 Checklist

3.1.1 Essential skills

You must be able to do the following:

- Be familiar with the UNIX command line interface and its basic functionalities.
  - Be able to navigate through a file system, create and remove files and directories. Use the `cd`, `ls`, `mkdir`, `cp`, `mv` and `rm` commands.
- View files using `cat` and `less`
- Edit files using a CLI text editor.
  - We recommend `vim` but `emacs` and `nano` are also available.
    - `vimtutor` is a useful tool for learning.
  - Be able to open, edit and write files from the CLI.

3.1.2 Useful skills

The following skills are extremely useful, but not critical for Dev School.

- Search through files and directories for text patterns with `find` and `grep`
- Redirect output to a new command with the `|` operator.
  - Combine this with `grep`!
- Redirect output to a file with > and >>
- Understand environment variables, `env`, `export`, `echo`
- Monitor a system with `ps`, `top` and `htop`
- Manage processes (`kill`)

3.1.3 Bonus

- Use a CLI editor to create a simple bash script file which prints "Hello World" (`echo`).
- Make it runnable (`chmod +x`) and run it directly from the command line.
- If your script file is called `hello.sh` what is the difference between running `bash hello.sh` and `./hello.sh`? What is a shebang?

CHAPTER

FOUR

C++

4.1 Resources

4.1.1 Recommended Exercises

Write code! The best way to get better at a programming language is by writing code.

One way to do this is through Code Katas. There are many lists of Code Katas online. Here is one:

- Code Katas

4.1.2 Recommended Reading

A Tour of C++, 3rd Edition by Bjarne Stroustrup

Note:

- If you *are not proficient in C++*, please read all the chapters noted below.
- If you *are proficient in C++* but haven't kept up with modern features of the language (since C++17), please read the relevant chapters to ensure familiarity before the traineeship starts.

Chapters

- 1. The Basics
- 2. User-Defined Types
- 3. Modularity
- 4. Error Handling
- 5. Classes
- 6. Essential Operations
- 7. Templates
- 8. Concepts
- 9. Library Overview
- 10. Strings and Regular Expressions - only the following sections:
  - 10.2 Strings
  - 10.3 String Views

    - 10.5 Advice
- 11. I/O Streams
- 12. Containers
- 13. Algorithms
- 15. Pointers and Containers

4.1.3 Recommended Videos

- Master C++ Value Categories

C++ value categories are a topic that many people find difficult to understand, but are essential to professional C++ development. This video does a good job explaining it. Please spend the time to watch it if you can.

CHAPTER

FIVE

C++ LIBRARIES

5.1 Containers

The standard library comes with a collection of containers (such as std::vector or std::map). Useful links -

- All standard containers
- Interesting article on standard and non-standard containers

5.2 Algorithms

The standard library also comes with a collection of algorithms. Some important design choices here are:

- These algorithms are designed to operate on iterators instead of operating on containers. This allows us to have algorithms that are container-agnostic. STL algorithms can work on your custom containers and you can write custom algorithms that work on STL containers.

```
Container elements;
auto it = std::find(elements.begin(), elements.end(), 78);
if (it != elements.end()) {
    // Found 78, it is the iterator for 78
} else {
    // 78 was not in the collection.
}
```

- When relevant, the behavior of algorithms can be parameterized using so-called function objects. For example, by default std::sort() will sort in increasing order, but it can also be passed a function object to sort in any other order

```
template<class RandomIt, class Compare>
void sort(RandomIt first, RandomIt last, Compare comp);
// Will sort in natural order, e.g. 1, 2, 3
std::sort(elements.begin(), elements.end());
// To reverse the order of sorting:
std::sort(elements.begin(), elements.end(), std::greater<>{});
```

- All standard algorithms
- Additional motivation (reasons to use algorithms): Presentation by Sean Parent: C++ Seasoning

5.3 Iterators

Iterators are a generalisation of pointers and are the glue between algorithms and containers

- They "point" to something, can be dereferenced: *it
- There are various categories like `InputIterator`, `ForwardIterator`, `BidirectionalIterator`, `RandomAccessIterator`, `OutputIterator`, `ContiguousIterator`
- Knowing the iterator category of your container is valuable. `std::lower_bound` on a `std::vector` is logarithmic time complexity (`RandomAccessIterator`) whereas the same algo on `std::list` is linear (`BidirectionalIterator`)
- Refer to this tutorial to learn more.
- Warning: There are no bounds checks in operations that change iterator positions. You are expected to do this yourself if necessary. No exceptions will be thrown. Dereferencing an iterator which points outside the bounds of the container results in undefined behavior (UB).

5.4 Third-party libraries

Don't reinvent the wheel. There are many high-quality third-party libraries that we use at IMC. Some examples include:

- Google Test/Google Mock, will be covered later in pre-work
- Abseil - A collection of C++ libraries, by Google. Among other things, it provides a very high-performance hash table implementation
- Boost - A very large collection of C++ libraries

CHAPTER

SIX

C++ SANITIZERS & SETTINGS

Programming is hard and error-prone, and we need all the help that we can get. Luckily, there are tools to help us catch mistakes as early as possible.

6.1 Compiler warnings

The compiler can warn us when it detects potentially harmful/buggy constructs being used in our code. These warnings are controlled by passing flags (such as -Wall) to the compiler.

At IMC, we treat all warnings as errors: Compilation will fail if any warning is reported!

Warnings are enabled by adding a flag to the compile command. In Bazel we can add global warnings in our `WORKSPACE` file, where we configure our `imc_cc_toolchain`, by adding them as a `copt` or adding a copts argument directly to a rule:

```
imc_cc_toolchain(
    copts = [
        "-Wall",
        "-Werror",
        "-Wconversion",
    ],
    gcc_only_cxxopts = [
        "-Wno-maybe-uninitialized",
    ],
)
```

```
cc_library(
    name = "cc-lib",
    copts = [ "-Wsign-conversion" ]
    ...
)
```

Some warnings in `-Wall, -Wextra or -pedantic` are, well, too pedantic. Individual warnings can be disabled pre-pending no- to the warning name, e.g. to disable `unused-parameter`:

```
copts = ["-Wno-unused-parameter"],
```

Reference to warnings for gcc compiler.

6.2 Sanitizers

Sanitizers allows detecting certain classes of errors (such as memory or threading issues) at runtime. The typical workflow is:

- First, during compilation, the sanitizer instruments your code to add certain checks
- Then, at runtime, these checks fire if an error is detected, and an error message is printed

The runtime overhead induced by sanitizers is almost always unacceptable for production usage. Therefore, we tend to run sanitized code as part of our CI/CD pipelines (or locally) instead. Therefore sanitizers can only catch issues if we have sufficient unit tests.

Types of Sanitizers

- Address Sanitizer ASan Docs
- Thread Sanitizer TSan Docs
- Memory Sanitizer MSan Docs
- Undefined Behavior Sanitier UBSan Docs

See gcc reference on how to enable sanitizers. ;tldr - use `-fsanitize=address` for ASan, `-fsanitize=thread` for TSan, and `-fsanitize=undefined` for UBSan.

The IMC Bazel toolchain has first class support for sanitizers, but not all compilers support all sanitizers.

- –features=asan (address)
- –features=lsan (leak)
- –features=ubsan (undefined behavior)
- –features=cast_san (float cast san)
- –features=divide_by_zero_san

Examples -

```
bazel run //engine --features=asan
bazel run //engine --features=lsan
```

CHAPTER

SEVEN

BAZEL

Bazel is a powerful new build system that offers some significant benefits over alternatives like Maven. Its advanced caching features enable faster builds even for large code bases. Furthermore, Bazel's support for multiple languages and technologies makes it a versatile tool for polyglot development systems.

At IMC, we use Bazel for our next-generation execution codebases, and intend to migrate more projects to Bazel in the future.

7.1 Recommended Reading

- Intro to Bazel
- Bazel: Getting Started
- Bazel Tutorial: Build a C++ Project

7.2 Checklist

7.2.1 Concepts

You should understand generally the following concepts:

- Build Tool Basics
  - The purpose and benefits of build automation tools
- Bazel's Vision and Philosophy
- Basic Concepts
  - Workspaces, packages, targets
  - Labels
  - Build files and rules
  - Dependencies
  - Target/Load Visibility
  - Hermeticity
- Bazel extensions
- Remote Caching

- Remote Execution

7.2.2 Practical Skills

Know how to do the following:

- Set up and configure a basic Workspace
- Read and write simple Starlark code
- Bazel CLI
  - `build`, `test`, `run`, etc.
- Bazel queries
  - List all labels in a workspace
  - Find the versions of all dependencies of a target
  - Find all of the targets that depend on a target
  - Find which direct dependency pulls in a specific transitive dependency
- Add a new dependency to a Bazel project

CHAPTER

EIGHT

CLION

An IDE is essential to improving your productivity as a Software Engineer. CLion is a highly recommended C++ IDE, and the IDE with the most internal support.

8.1 Recommended Reading

- CLion - Learn
- Bazel Clion Setup

8.2 Recommended Viewing

These are targeted at IntelliJ, but equivalent functionality exists for most features.

- Youtube - Be More Productive With IntelliJ IDEA
- Youtube - IntelliJ IDEA Debugger Essentials

8.3 Practical skills

Be able to use the following features:

- Reading Code
  - Code Navigation
  - Find Usages
- Writing Code
  - Inspections
  - Live Templates
  - Nullability Analysis and Annotations
  - Local History
  - Code Style and Formatting
- Refactoring
  - Renaming

  - Change Signature
  - Extract/Introduce refactorings
  - Inline
  - Safe Delete
  - Migrate
- Running
  - Running a C++ application
  - Running tests
- Debugging
- Bazel Integration

CHAPTER

NINE

GOOGLETEST

GoogleTest it the preferred testing and mocking library for the majority of C++ development teams at IMC, and what we will be using during Development School.

9.1 Exercise

There is an exercise to complete and submit as part of the pre-work. We recommend you go through the recommended reading materials below before completing the exercise.

Instructions

Clone the repository from GitHub and follow the instructions in the README.md file.

Email your completed OrderTests.cpp file to Dev School Leads (devschoolleads@imc.com) with the subject "Pre-work GoogleTest exercise submission".

9.2 Recommended Reading Material

- GoogleTest Primer
- gMock
- GoogleTest Matchers Reference - Matchers used with `EXPECT_THAT` and `ASSERT_THAT` are preferred over old style GoogleTest assertions. This reference shows the variety available. Get familiar with what is available.

9.3 Checklist

9.3.1 Concepts

Understand these concepts:

- Unit tests vs Integration or End-to-end testing
- Mocks vs Fakes

9.3.2 Practical Skills

Know how to do the following:

- GoogleTest
  - Test Fixtures
    - Run simple unit tests
    - Write assertions using matchers
    - Ignore tests
    - Test setup/teardown test state
      - `SetUp`, `TearDown`
      - Suite/Global-level SetUp/TearDown
    - Test exceptions and failures
  - CLion integration
    - Run tests within CLion
- gMock
  - Mocking
    - Create mocks
    - Stub method calls
- Bazel
  - Run GoogleTest with Bazel (Hint: Quickstart: Bazel with GoogleTest)

9.3.3 Advanced Features

We won't be using these in dev school, but they are useful and used in some places at IMC.

- Parameterized tests
- Typed tests