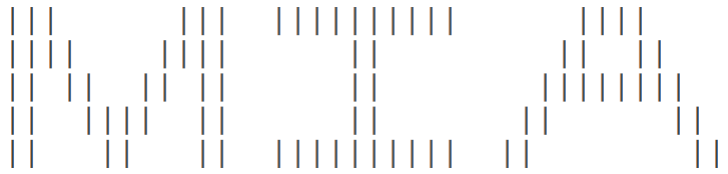

Multiple Integrated Applications (MIA) Manual & Programming Documentation

Developer: Antonius Torode

Version – 2.001
Latest update: May 26, 2025

The Original Maintainer of this work is: Antonius Torode.

The Current Maintainer of this work is: Antonius Torode.

This document is designed for the sole purpose of providing documentation for Multiple Integrated Applications (MIA), a program written for and created for personal use. Throughout this document, "MIA" will be used as a reference to "Multiple Integrated Applications." This acronym is solely designed for use in conjunction with the program and was originally created by the creator of this document. MIA is designed to be used for multiple purposes based on the continual addition of functionality. It can be adapted to work in other situations and for alternate purposes.

This document is continuously under development.
Most Current Revision Date: May 26, 2025

Torode, A.
MIA Manual.
2021.
Includes Source Code and References

Contents

1 Multiple Integrated Applications (MIA) 1
- 1.1 Introduction . . . 1
  - 1.1.1 Repository Overview . . . 1

2 MIA Development 3
- 2.1 Project Dependency Structure . . . 3
- 2.2 Build Script Overview . . . 3
  - 2.2.1 Purpose . . . 3
  - 2.2.2 Key Functionalities . . . 3
  - 2.2.3 Platform Support . . . 4
  - 2.2.4 CMake Integration . . . 4
  - 2.2.5 Installation and Uninstallation . . . 5
  - 2.2.6 Design Considerations . . . 5
- 2.3 CMake Setup . . . 5
  - 2.3.1 Project Definition, Standards and Versioning . . . 5
  - 2.3.2 Build Options . . . 5
  - 2.3.3 Platform-Specific Path Configuration . . . 5
  - 2.3.4 Configuration Constants . . . 6
  - 2.3.5 Project Structure and Targets . . . 6
  - 2.3.6 Installation Rules . . . 7
- 2.4 Application Framework . . . 7
- 2.5 Base Application Class: `MIAApplication` . . . 9
- 2.6 Command Option System . . . 11
  - 2.6.1 CommandOption Class . . . 11
  - 2.6.2 command_parser Namespace . . . 12
- 2.7 Error Handling Framework . . . 13
- 2.8 Global Constants and Paths . . . 15
  - 2.8.1 constants Namespace . . . 15
  - 2.8.2 paths Namespace . . . 15
- 2.9 Configuration System: `MIAConfig` . . . 16
  - 2.9.1 Overview . . . 16
  - 2.9.2 Key Features . . . 16
  - 2.9.3 Class Hierarchy and Responsibilities . . . 16
  - 2.9.4 Usage Pattern . . . 17
  - 2.9.5 Quick Example . . . 18
  - 2.9.6 Extensibility . . . 18

3 MIAConfig Files 19
- 3.1 Introduction to MIAConfig . . . 19
- 3.2 Structure, Format, and Syntax . . . 19
- 3.3 How MIAConfig Files Are Handled at Runtime . . . 20
- 3.4 Extensibility and Maintainability . . . 20
- 3.5 MIA Configuration (MIAC) . . . 20

4 D0sag3 Command (D3C) Integration **21**
4.1 D3C Introduction and Overview . . . 21
4.2 d0s1 Encryption . . . 21
4.3 d0s2 Encryption . . . 22
4.4 d0s3 Encryption . . . 22

5 Workout Generation **23**
5.1 MIA Workout Generation Overview and Introduction . . . 23
5.2 Workout Application Installation . . . 23
5.3 Input File and Defining Workouts . . . 23
5.3.1 Input File . . . 23
5.3.2 Workout Generation Parameters . . . 24
5.3.3 Defining Exercises . . . 25
5.4 Generation Algorithm . . . 25
5.4.1 Number of Sets Per Workout . . . 25
5.4.2 Number of Exercises Per Set . . . 26
5.4.3 Number of Reps Per Exercise . . . 27
5.5 Real World Difficulties . . . 28
5.6 Notes on Appropriate MIA Parameters For Usage . . . 28

6 World of Warcraft (WoW) Features in MIA **30**
6.1 WoW Fishbot . . . 30
6.1.1 Setup and Configuration . . . 30
6.2 Mailbox Management . . . 32
6.2.1 Sending Duplicates of a Letter . . . 33
6.2.2 Unloading Duplicated letters . . . 33

7 Sequencer **35**
7.1 Using the Sequencer . . . 35
7.2 Defining a Sequence . . . 36
7.3 Notes on Using The Sequencer . . . 37

This page intentionally left blank.
(Yes, this is a contradiction)

Chapter 1

Multiple Integrated Applications (MIA)

1.1 Introduction

MIA is designed to be a collection of scripts, tools, programs, and commands that have been created in the past and may be useful in the future. It's original idea was a place for the original author to combine all of his previous applications and codes into one location that can be compiled cross platform. MIA is written in C++ but will contain codes that were originally designed in C#, Java, Python, and others. MIA is created for the authors personal use but may be used by others if a need or desire arises under the terms of Antonius' General Purpose License (AGPL).

The MIA acronym was created by the original author for the sole purpose of this application. The design of MIA is a terminal prompt that accepts commands. There are no plans to convert MIA into a GUI application as there is currently no need; however, some elements may be programmed in that produce a GUI window for certain uses such as graphs. The MIA manual is designed to be an explanation of what MIA contains as well as a guide of how to utilize the MIA program to it's fullest.

As MIA is continually under development, this document is also. Due to this, it may fall behind and become slightly outdated as I implement and test new features into MIA. I will attempt to keep this document up to date with all of the features MIA contains but I can only do so if time permits.

1.1.1 Repository Overview

This repository is a modern recreation of the original MIA (Multiple Integrated Applications) project available at `https://github.com/torodean/Antonius-MIA`. It serves as a centralized platform for storing various utilities, scripts, and applications developed over time, allowing easy reuse and accessibility. The project is modular and designed for cross-platform compatibility.

For a comprehensive list of commands, features, and usage guidelines, refer to the MIA manual located in this file or accessible online at `https://github.com/torodean/MIA/docs/`.

Project Structure

- `bin/`
  Contains the main source code for all integrated applications and core libraries, organized by module and functionality.
- `docs/`
  Houses all project documentation, including manuals, design notes, and usage instructions.
- `scripts/`
  Stores utility scripts for tasks such as building, installing, and managing the project.
- `build/` (generated)
  Temporarily holds build artifacts and platform-specific outputs during the compilation process.
- `release/` (generated)
  Contains release-ready binaries and configuration files for distribution or standalone execution.

- `resources/`
  Includes static resources and configuration files required by MIA tools and executables.

Additional directories may be introduced as the project evolves to support new functionality or organizational needs.

Chapter 2

MIA Development

2.1 Project Dependency Structure

The project is organized into layers to manage dependencies clearly:

- Core: Contains fundamental types, error codes, and logging interfaces. It has no dependencies on other modules.
- Utilities: Depend on Core and provide reusable helper functions like file and string operations.
- Libraries: Depend on Core and Utilities, implementing domain-specific logic such as math functions.
- Applications: Depend on Core, Utilities, and Libraries to build and run the final executable.

Modules within the same layer may depend on each other as needed. The key rule is to avoid circular dependencies between layers to maintain clear and manageable dependency flow. This layering ensures a unidirectional dependency flow, promoting modularity and easier maintenance.

2.2 Build Script Overview

The build process for MIA is orchestrated through a Bash script, `build.sh`, located in the root directory of the project. This script automates common tasks involved in building, installing, and managing the application.

2.2.1 Purpose

The build script serves as a centralized interface for developers and users to:

- Configure and build the MIA project using CMake.
- Install the application on the system.
- Manage dependencies for supported platforms.
- Generate release builds.
- Clean the build environment for fresh compilation.
- Uninstall installed components.

2.2.2 Key Functionalities

The script supports a range of command-line options to control its behavior:

```
$ ./build.sh -h

build: build.sh [options...]
This build script will automate the build and install of MIA.
```

```
Options:
    -h    Display this help message.
   -S    Run the initial setup to install dependancies and such. then exit.
   -C    Perform a clean build by removing the build directory first.
   -v    Enable verbose output during build process.
   -D    Attempt to Install dependencies.
   -I    Install MIA after building (requires admin). Use a clean build if errors occur.
   -R    Update the release files. Use a clean build if errors occur.
   -U    Uninstall all MIA files. This will uninstall then quit without other actions.
   -T    Run all tests.
   -d    Add flags to cmake for a debug build (useful if gdb is needed).
```

This build script is continually evolving and new options are added.

Adding New Build Script Options

To add a new option to the build script, a few things need considered. First, options must be unique. Each option should contain a short option argument (i.e. `-v, -h, -d`) as well as a description. These new options need added to the `usage` function and the `getopts` loop.

```
usage() # Create the usage output.
{
  # ...
  echo "  Options:"
  echo "    -f    Adding a new option with the -f flag."
  # Other entries...
}
```

```
# Define the build script options and create variables from options.
while getopts "hCvDIfRU" opt; do # Add the flag to the list here.
  case $opt in
    # Other entries...
    f) flagVariable=1 # Add flag actions here.
      ;;
    # Other entries...
  esac
done
```

The flag actions should be added to the getopts list (as seen above). These actions are typically setting a variable to store that the flag is used, then called later in the script. But functions or lists of other variables can be added here too.

2.2.3 Platform Support

The script contains conditional logic to perform platform-specific setup, with support primarily targeted at Linux environments. At the time of writing this, dependency installation scripts are stubbed for other platforms (e.g., macOS, Windows) but are not fully implemented.

2.2.4 CMake Integration

The build is managed via CMake, which is invoked with appropriate flags based on script options. A separate `build` directory is used to isolate generated files, and Make is used to compile the project with support for parallel jobs. See section 2.3 for more details. The build script handles adding flags to the cmake configuration via the `cmakeArgs`. To add a flag to be used during the cmake build, add the appropriate flag to the `cmakeArgs` via the following:

```
cmakeArgs = "$cmakeArgs -DNEW_FLAG_HERE"
```

2.2.5 Installation and Uninstallation

Installation is optionally triggered using `cmake -install` followed by a custom installation script. Uninstallation is handled early in the script flow via a dedicated script and bypasses other operations. Although installation is handled via cmake, there are also separate install and uninstall scripts which are located in the scripts folder. These are setup to be called during the build script and are meant to implement additional install or uninstall steps. these are called via the following

```
$rootDirectory/scripts/install.sh
$rootDirectory/scripts/install.sh
```

2.2.6 Design Considerations

This script abstracts complexity away from the end user, provides a repeatable and configurable build process, and facilitates both development and deployment workflows. It is structured to be easily extendable for future platform support or additional build modes.

2.3 CMake Setup

The MIA project utilizes CMake to manage the build configuration, compilation, and installation processes. The setup is designed to be flexible and portable across supported platforms while allowing customization for release or system installation builds.

2.3.1 Project Definition, Standards and Versioning

The CMake configuration begins by specifying a minimum required CMake version and defining the project name `MIA` along with its version. The version is also passed as a preprocessor definition (`MIA_VERSION_VAL`) for use in the codebase. At the time of writing this, C++20 is set as the required language standard, and compiler warnings are enabled globally using the `-Wall` flag. The build directory path is stored in a variable for reference.

2.3.2 Build Options

Two primary options control build behavior:

- `RELEASE_BUILD`: When enabled, modifies install paths and triggers release-specific build behavior.
- `SYSTEM_INSTALL`: When enabled, configures installation paths and behaviors suitable for system-wide deployment.
- NOTE: The two build behaviors are somewhat incompatible and a clean build needs done when switching between them.

2.3.3 Platform-Specific Path Configuration

Installation paths and default directories for configuration files and logs are set depending on the operating system:

- **Windows:** Uses `C:/Program Files` for the install and `C:/ProgramData` for supporting files.
- **Linux/Unix:** Uses `/usr/local` for the install, `/etc` for supporting resource files, and `/var/log` for logging.
- Unsupported platforms produce a configuration error.

The cmake section that contains these configurations will give the exact file paths chosen for the various files. The actual CMake file should be used for gathering the current used paths (this documentation may become out-dated) which will be in a section similar to the following:

```cmake
# OS-Specific Path Configuration
if(WIN32)
    set(APP_INSTALL_LOCATION "C:/Program Files/mia")
    set(DEFAULT_SYSTEM_CONFIG_FILE_DIR "C:/ProgramData/mia")
    set(DEFAULT_SYSTEM_LOG_DIR "C:/ProgramData/mia/logs")
elseif(UNIX AND NOT APPLE)
    set(APP_INSTALL_LOCATION "/usr/local/mia")
    set(DEFAULT_SYSTEM_CONFIG_FILE_DIR "/etc/mia")
    set(DEFAULT_SYSTEM_LOG_DIR "/var/log/mia")
else()
    message(FATAL_ERROR "Unsupported platform")
endif()
```

2.3.4 Configuration Constants

Both system-level and repository-level default paths for configuration and log files are defined and exposed to the compiler through compile-time definitions. This centralizes resource location information for consistent access across the codebase. To add a constant value defined in cmake such that it is accessible in the source code (c++), the following method can be used.

Defining C++ Accessible CMake Variables

First, the value needs defined in the CMake using the `set` and `add_definiition` methods:

```cmake
# The version of the current MIA code.
set(MIA_VERSION_VAL "2.001")

# Add MIAVERSION as a preprocessor variable.
add_definitions( -DMIA_VERSION_VAL=\"${MIA_VERSION_VAL}\" )
```

These values can then be accessed within the source C++ code directly via

```cpp
// The MIA Version value gathered from CMake.
inline const std::string MIA_VERSION = MIA_VERSION_VAL;
```

These values are typically added within the `Constants_LIB` files which contains system related MIA constants.

2.3.5 Project Structure and Targets

The project is organized into subdirectories for executables, libraries, resources, and tests:

- **Executables:** Targets like `MIATemplate` are defined with source and header files, linked against core libraries, and have conditional install rules based on build options.

```cmake
# Create the MIATemplate executable.
set(MIATemplate_SRC MIATemplate.cpp MIATemplate_main.cpp )
set(MIATemplate_INC MIATemplate.hpp )
add_executable(MIATemplate ${MIATemplate_SRC} ${MIATemplate_INC} )
target_link_libraries(MIATemplate PRIVATE Core_LIB )

if(SYSTEM_INSTALL)
 install(TARGETS MIATemplate DESTINATION ${APP_INSTALL_LOCATION})
endif()

if(RELEASE_BUILD)
 install(TARGETS MIATemplate DESTINATION ${RELEASE_INSTALL_LOCATION
    ↪ })
endif()
```

- **Libraries:** Utility libraries such as `Types_LIB` are created from source files and expose their include directories for dependent targets. Libraries can link to other internal utilities or libraries.

```
# Create the Types_LIB
set(Types_SRC StringUtils.cpp )
set(Types_INC StringUtils.hpp )
add_library(Types_LIB ${Types_SRC} ${Types_INC})
target_link_libraries( Types_LIB PUBLIC BasicUtilities_LIB )

# Expose this library's source directory for #include access by
    ↪ dependent targets
target_include_directories(Types_LIB PUBLIC ${
    ↪ CMAKE_CURRENT_SOURCE_DIR})
```

- **Tests:** Test code is included via a dedicated subdirectory.

```
# Include the test directory.
add_subdirectory( test )
```

- **Resources:** Static resources are managed in their own subdirectory. These are primarily for configuration files and other resources. These are installed via the `FILES` Cmake keyword as follows:

```
set(MIA_Config_Files
    MIATemplate.MIA
    # Other files here...
)

if(SYSTEM_INSTALL)
    install(FILES ${MIA_Config_Files} DESTINATION ${
        ↪ DEFAULT_SYSTEM_CONFIG_FILE_DIR})
endif()

if(RELEASE_BUILD)
    install(FILES ${MIA_Config_Files} DESTINATION ${
        ↪ RELEASE_CONFIG_INSTALL_LOCATION})
endif()
```

2.3.6 Installation Rules

Conditional install commands allow targets to be installed either to system-wide locations or release-specific directories based on build flags (seen in the above shown cmake blocks). This supports flexible deployment workflows for development, testing, and production release. Overall, the CMake setup provides a structured, modular, and configurable build environment that adapts to different platforms and build scenarios while maintaining centralized control over important variables and paths.

2.4 Application Framework

Overview

The application framework provides a minimal structure for C++ applications that follow a consistent life cycle. It enforces a standard interface consisting of initialization and execution phases, and offers utilities to streamline application entry point definition. The design leverages C++20 concepts and templates for compile-time enforcement of interface contracts.

Life Cycle Model

Applications using this framework must implement the following life cycle methods:

- `void initialize(int argc, char** argv)`
  Performs startup logic, such as argument parsing and resource allocation.
- `int run()`
  Contains the main execution logic of the application. The return value is propagated as the process exit code.

Interface Enforcement

The framework uses a C++20 concept, `AppInterface`, to statically constrain application types. This ensures that any type passed to the launcher function conforms to the expected interface, eliminating runtime errors due to missing methods. Other types can be added to this interface if they are in the future and should conform to the same format as the currently existing ones.

```
template<typename App>
concept AppInterface = requires(App app, int argc, char** argv)
{
    { app.initialize(argc, argv) } -> std::same_as<void>;
    { app.run() } -> std::same_as<int>;
};
```

Application Launch

Applications are launched using the `runApp` template function:

- Accepts any type satisfying `AppInterface`.
- Calls `initialize` followed by `run`, in that order.
- Returns the result of `run` as the application's exit code.
- Other method calls can be added in this sequence if they are needed in the future.

```
template<AppInterface App>
int runApp(int argc, char** argv)
{
    App app;
    app.initialize(argc, argv);
    return app.run();
}
```

Entry Point Definition

To reduce boilerplate (code needing to be implemented by the end user) and unify application entry points, the framework provides an application macro entry point:

- `MIA_MAIN(AppClass)` defines a `main()` function that delegates to `runApp<AppClass>`.
- This enables consistent startup across applications while preserving type safety and clarity.

```
#define MIA_MAIN(AppClass)                    \
int main(int argc, char** argv)               \
{                                             \
    return runApp<AppClass>(argc, argv);      \
}
```

This macro (along with the above defined interface) makes creating a main for an application simple and straight forward and provides consistent behavior across apps. Typically, for clarity, the user should define a separate main cpp file which contains the implementation of this interface.

```cpp
// The MIA Application Framework templates
#include "AppFramework.hpp"
// The file containing the fishbot app.
#include "MIATemplate.hpp"

MIA_MAIN(MIATemplate)
```

Some tips and considerations follow:

1. Define an application class that implements the required `initialize` and `run` methods.
2. Use `MIA_MAIN(YourAppClass);` in a translation unit to define the application entry point.
3. Avoid manual `main()` definitions to preserve uniform lifecycle management.

Benefits

- Promotes a consistent lifecycle across applications.
- Enables interface enforcement at compile time.
- Minimizes boilerplate in entry point logic.
- Facilitates cleaner and more maintainable application architecture.

2.5 Base Application Class: `MIAApplication`

Overview and Purpose

The base MIA Application (see `MIAApplication.hpp`) serves as the foundational base class for applications built using the framework (See section 2.4). It defines a standardized interface for application life cycle methods and provides shared functionality for command-line argument parsing, particularly handling common flags such as `-v` (verbose) and `-h` (help). This class is intended to be sub-classed by MIA applications. It provides default behavior for argument parsing and flag handling, while enforcing the implementation of core logic via a pure virtual `run()` method. This ensures that the application framework templates and required concepts are enforced.

Initialization and Execution Interface

- `virtual void initialize(int argc, char* argv[])`
  Handles initial setup, including parsing common command-line arguments. This method is intended to be overloaded to handle app-specific command line arguments and app loading. Derived classes should override this method to extend parsing logic but should still invoke the base implementation to preserve flag handling.

- `virtual int run() = 0`
  A pure virtual function that must be implemented by all derived classes. It defines the main operational logic of the application and must return an integer exit code.

Flag Handling

- Verbose mode is handled internally and can be queried via `getVerboseMode()`.
- Help flag status is stored and can be utilized to trigger usage messages. When the help flag is specified, an application is automatically set to print the potentially-overloaded printHelp() method, print the help message, then exit the application.)
- `printHelp()` provides a virtual method to emit shared help information; it can be extended or overridden by subclasses. Derived classes should override this method to extend help output to be app specific but should still invoke the base implementation to preserve base command help options.

- A `logger::Logger logger` object is created with the optional `-logfile` flag setting a custom log file. This allows for easy logging by applications. This depends on application and user permissions in the case where the file path does not exist. The `logger` is stored as a private data member and meant to be hidden from the end-user. The logging functionality should be called via a `log(string)` method, which automatically handles verbose handling. See section **??** for more details.

Protected Members

- `bool verboseMode`
  Indicates whether verbose output was requested.

- `bool helpRequested`
  Set to `true` if the user requested help via the command-line.

Usage Guidelines

An example application exists in `bin/apps/template` which demonstrates how the base application class is used.

1. Inherit from `MIAApplication`.

```
class MIATemplate : public MIAApplication
{ // Class details... }
```

2. Override `initialize()` to add application-specific argument parsing; call the base method to retain common flag handling at the start of the application initialize() implementation.

```
void MIATemplate::initialize(int argc, char* argv[])
{
    try
    {
        MIAApplication::initialize(argc, argv);
        // Setup app-specific command line arguments here...
    }
    catch (const error::MIAException& ex)
    {
        std::cerr << "Error during MIATemplate::initialize: " << ex.what()
            << std::endl;
        // Handle error appropriately...
    }

    // Other init code...
}
```

3. Implement the `run()` method with the application's main logic.

```
void MIATemplate::run()
{
    // Perform app functions...
}
```

4. Use `getVerboseMode()` to conditionally enable verbose output (if the verbose flag was used when running the application).

```
if (getVerboseMode())
    std::cout << "This is verbose output!" << std::endl;
```

5. Override `printHelp()` to add help output specific to the application. Call the base method to retain base application help menu output.

```cpp
void MIATemplate::printHelp() const
{
    MIAApplication::printHelp();

    std::cout << "MIATemplate specific options:" << std::endl
              // Add app-specific help output here...
              << std::endl;
}
```

6. Call the `log()` method in order to log information.

```cpp
log("This is a message to log");
```

In the rare case that the user wants to overload the standard verbose handling, an overloaded `log(string, bool)` method exists which accepts a verbose flag as the second argument.

```cpp
log("This is a message to log", true);
```

Benefits

- Standardizes life cycle structure across applications.
- Centralizes command-line flag logic.
- Reduces redundancy in application setup.
- Encourages consistent help and verbose interfaces.

2.6 Command Option System

The command option system provides structured parsing and management of command-line arguments. It is composed of two main components: the `CommandOption` class and the `command_parser` namespace. Together, they offer a consistent and type-safe interface for defining, parsing, and validating command-line options. An example application exists in `bin/apps/template` which demonstrates how the command options can be used.

2.6.1 CommandOption Class

The `CommandOption` class encapsulates metadata and behavior for individual command-line arguments. It supports a variety of data types and provides a standardized help display format.

Key Features

- **Supported Types:** These define the type of command option that is expected. Each command option should only have one expected type which is defined by the `commandOptionType` enum:
  - `boolOption`: A single command flag to store true when used (false otherwise).
  - `intOption`: A command flag followed by an integer.
  - `doubleOption`: A command flag option followed by a double.
  - `stringOption`: A command flag option followed by a string.
- **Constructor:** Accepts short/long argument forms, a description, data type, and a `required` flag. This constructor should be called during construction for the application class. This constructor sets the command option flags, description, and the type; all of which are needed during initialization to enable command option output and parsing.

```
MIATemplate::MIATemplate() :
    configFileOpt("-c", "--config", "Specify a config file to use.",
        CommandOption::commandOptionType::stringOption),
    testOpt("-t", "--test", "A test command option.",
        CommandOption::commandOptionType::boolOption)
{ };
```

- **Help Output:** `getHelp()` returns a formatted string of the form:

```
-s, --long        Description
```

This method is intended to return a string for each command option in a consistent format in order to make constructing the overridden printHelp() message of the base application class simpler. The getHelp() method can be used in constructing the help message for an app in a way similar to the following:

```
void MIATemplate::printHelp() const
{
 MIAApplication::printHelp();

 // This is a dump of the help messages used by the various command
     options.
 std::cout << "MIATemplate specific options:" << std::endl
           << configFileOpt.getHelp() << std::endl
           << testOpt.getHelp() << std::endl
           << std::endl;
}
```

- **Value Parsing:** The `getOptionVal<Type>` method retrieves the typed value from the command-line using appropriate dispatch. This is typically done during app initialization and serves to simplify the process of parsing command line options by allowing a single method to get and set the appropriate variables.

```
void MIATemplate::initialize(int argc, char* argv[])
{
    try
    {
        MIAApplication::initialize(argc, argv);

        // Set the values from the command line arguments.
        testOpt.getOptionVal<bool>(argc, argv, testMode);

        std::string configFile = defaultConfigFile;
        configFileOpt.getOptionVal<std::string>(argc, argv, configFile);
        config.setConfigFileName(configFile, constants::ConfigType::
            KEY_VALUE); // handles config.initialize().
    }
    catch (const error::MIAException& ex)
    {
        std::cerr << "Error during MIATemplate::initialize: " << ex.what()
             << std::endl;
    }
}
```

- **Error Handling:** Throws `MIAException` if a type mismatch occurs or parsing fails.

2.6.2 command_parser Namespace

This namespace implements the actual logic for extracting typed values from `argc`/`argv`. Each function accepts short and long option names, and a reference to the variable where the parsed value should be stored.

Parsing Functions

- `void parseBoolFlag(int argc, char* argv[], const std::string& shortArg, const std::string& longArg, bool& outValue);`
- `void parseIntOption(int argc, char* argv[], const std::string& shortArg, const std::string& longArg, int& outValue, bool required = false);`
- `void parseDoubleOption(int argc, char* argv[], const std::string& shortArg, const std::string& longArg, double& outValue, bool required = false);`
- `void parseStringOption(int argc, char* argv[], const std::string& shortArg, const std::string& longArg, std::string& outValue, bool required = false);`

Behavior and Integration

Each parser:

- Validates the presence of the option.
- Converts the argument to the correct type.
- Throws a `MIAException` on error, such as type mismatch or missing required value.

The `CommandOption` class acts as a high-level interface, while `command_parser` performs the low-level parsing. Together, they ensure robustness, type safety, and consistent error reporting across the command-line interface.

2.7 Error Handling Framework

Overview

The MIA error handling system provides a structured and consistent mechanism for reporting, categorizing, and propagating errors throughout MIA applications. It is centered around a custom exception type (`MIAException`) and an extensible set of error codes defined via the `ErrorCode` enumeration.

Error Codes

The `ErrorCode` enumeration defines a comprehensive list of standardized error values used throughout the MIA framework. These include:

- System-aligned codes (e.g., `Access_denied = 5`)
- Application-specific codes starting at 31415 (i.e., `int`($\pi \times 10000$)), e.g., `Feature_In_Dev`
- Custom fatal and configuration errors (e.g., `FATAL_File_Not_Found`, `Config_File_Not_Set`)

Each error code is associated with a human-readable description defined in `ErrorDescriptions.hpp`, accessible via:

```
const std::string& getErrorDescription(ErrorCode code);
```

MIAException Class

The primary mechanism for structured error propagation is the `MIAException` class:

- Inherits from `std::exception`.
- Encapsulates an `ErrorCode` and an optional detailed message.
- Supports integration with standard C++ `try/catch` error handling.

Interface Summary:

- `MIAException(ErrorCode, const std::string& details = "")`
  Constructs an exception with a specific code and optional extra context.
- `const char* what() const noexcept`
  Returns a descriptive error string combining code-based and contextual information.
- `ErrorCode getCode() const noexcept`
  Retrieves the error code associated with the exception.

Deprecated Legacy Functions

Several older functions exist but are marked deprecated in favor of `MIAException`:

- `returnError()`, `errorInfo()`, and `errorInfoRun()` are retained for backward compatibility but should be avoided.
- These are flagged with `[[deprecated]]` attributes.

Usage Guidelines

1. Throw `MIAException` in place of manual error returns. An appropriate error code from `error::ErrorCodes` should be used. If one does not exist, one should be added.

```
try
{
    // Do some code...
}
catch (const error::MIAException& ex)
{
    std::string err = "Details of this error!";
    throw error::MIAException(error::ErrorCode::Catastrophic_Failure, err)
        ;
}
```

2. Use `getErrorDescription()` to log or report known error codes.
3. Check exception codes via `getCode()` in catch blocks to take context-specific actions.

```
catch (const error::MIAException& ex)
{
    if (ex.getCode() == error::ErrorCode::Catastrophic_Failure)
    {
        // Do something specific for this error.
    }
}
```

4. Add new error codes and descriptions for error-specific handling and information gathering to ensure that apps remain transparent and easy to use.

```
// In bin/core/Error.hpp
namespace error
{
    enum ErrorCode
    {
        New_Error = 31450 // Add error codes here
    }
}
```

```
// In bin/core/Errordescriptions.cpp
namespace error
{
    const std::unordered_map<ErrorCode, std::string> errorDescriptions = {
        // Other error code descriptions here...
```

```
6           { New_Error, "Add error description here..." },
7       };
```

Benefits

- Unifies error handling across the application ecosystem.
- Enables richer debugging and logging through descriptive errors.
- Supports granular response logic based on typed error codes.

2.8 Global Constants and Paths

This section documents the centralized constants and path utilities used across the application. These components standardize access to configuration values, resource locations, and metadata such as the application version. See section 2.3.4 for some related information on adding global paths and compile time through CMake.

2.8.1 constants Namespace

The `constants` namespace provides globally accessible constant values compiled into the application.

Defined Constant

- `MIA_VERSION`: A `std::string` representing the version of the MIA application, injected from CMake via the `MIA_VERSION_VAL` macro.

2.8.2 paths Namespace

The `paths` namespace offers a centralized interface for accessing directory and file paths used by the system. These include installation-specific and repository-relative locations, as well as runtime-determined paths.

Static Path Constants

- `SYSTEM_CONFIG_FILE_DIR`, `SYSTEM_CONFIG_FILE`: Paths to the configuration directory and file for system installations.
- `REPO_CONFIG_FILE_DIR`, `REPO_CONFIG_FILE`: Paths used during development and testing.
- `SYSTEM_LOG_DIR`, `SYSTEM_LOG`: Default logging directory and file when installed.
- `REPO_LOG_DIR`, `REPO_LOG`: Logging paths for repository use.
- `INSTALL_LOCATION`: The root directory of the system installation.

Runtime Utilities

- `getExecutableDir()`: Returns the absolute directory of the running executable using platform-specific APIs.
- `isInstalled()`: Determines whether the application is running from the system-installed location by comparing the executable path with the install directory.
- `getDefaultConfigDirToUse()`: Selects the configuration directory based on execution context:
  - If installed: returns `SYSTEM_CONFIG_FILE_DIR`.
  - If a `resources` folder exists in the executable directory: returns that path.
  - Otherwise: returns `REPO_CONFIG_FILE_DIR`.

This is particularly useful when determining the full path of a configuration file to use. The application will automatically try to detect which file to use based on its location and which files are available. This allows for portability and flexibility.

This design ensures portability and flexibility across different environments (development, testing, deployment), with consistent fallback logic and platform-agnostic path resolution.

2.9 Configuration System

The `MIAConfig` class provides a flexible and extensible interface for managing configuration data in the MIA system. It replaces the legacy `Configurator` from the previous MIA project. The system is designed using the PIMPL (Pointer to IMPLementation) idiom, encapsulating implementation details and allowing for multiple configuration types such as key-value pairs or raw-line formats (with the flexibility to add more later).

2.9.1 Overview

- `MIAConfig` is the main public interface that users interact with.
- `ConfigData` is an abstract base class defining the interface for internal configuration storage.
- `KeyValueData` is a concrete implementation of `ConfigData::KEY_VALUE`, supporting standard key-value format parsing and access.
- `RawLinesData` is a concrete implementation of `ConfigData::RAW_LINES`, supporting raw line format parsing and access.
- Configuration types are enumerated via `constants::ConfigType`.

2.9.2 Key Features

- Supports setting and retrieving configuration files and types dynamically.
- Provides typed accessors: `getInt()`, `getDouble()`, `getString()`, `getBool()`, and vector versions.
- Supports verbose logging during file parsing for debugging.
- Can return all configuration entries as key-value pairs, raw lines, or other formats depending on the format and implementation details.

2.9.3 Class Hierarchy and Responsibilities

`MIAConfig`
: Acts as the front-end API. It delegates all data handling to the `ConfigData` object through a `std::unique_ptr`.

`ConfigData`
: Abstract base class defining the required interface for data handling implementations. Includes methods for loading, retrieving, and dumping configuration data.

`KeyValueData`
: Implements `ConfigData`. Parses files formatted with `key=value` lines into an internal map and supports type conversion and retrieval.

`RawLinesData`
: Implements `ConfigData`. Parses files formatted with `any` format (parsed by line) into an internal vector and supports retrieval for custom parsing.

2.9.4 Usage Pattern

1. Include a configuration object as a private member of the application class for each configuration file which is needed (used) by the application.

```
/// The configuration loader for this app.
MIA_System::MIAConfig config;
```

2. Instantiate `MIAConfig` with a file name and configuration type. This should be done at application construction so that the configuration is available to parse and use during initialization. A default configuration file name can be defined and used at object construction or defined later. The file name can contain a full file path, but if it does not, the MIAConfig class will attempt to identify an appropriate configuration directory during config initialization using the `paths::getDefaultConfigDirToUse()` method (see section 2.8.2 for more details).

```
MIATemplate::MIATemplate() :
    config(defaultConfigFile, constants::ConfigType::KEY_VALUE)
{ };
```

3. Call `initialize()`, `reload()` or `setConfigFileName(..)` during application initialization to parse and load the base configuration file into the appropriate config implementation (this is based on the specified configuration type).

```
void MIATemplate::initialize(int argc, char* argv[])
{
    try
    {
        // Other code...
        std::string configFile = defaultConfigFile;
        configFileOpt.getOptionVal<std::string>(argc, argv, configFile);

        // This will call config.initialize().
        config.setConfigFileName(configFile, constants::ConfigType::
            KEY_VALUE);
    }
    // Optionally print the config values after it is loaded.
    if (getVerboseMode())
        config.dumpConfigMap();

    loadConfig(); // An app-specific config loader.
}
```

4. Use the typed getter methods to retrieve settings from the config class and store them into application-specific variables.

```
void MIATemplate::loadConfig()
{
    try
    {
        // Load configuration here.
        configFileVals.boolValue = config.getBool("MyBoolValue");
        configFileVals.intValue = config.getInt("myIntValue");
        configFileVals.doubleValue = config.getDouble("myDoubleValue");
        configFileVals.stringValue = config.getString("myStringValue");
        configFileVals.listValue = config.getVector("myListValue", ',');
    }
    catch (error::MIAException& ex)
    { // Handle errors... }
}
```

2.9.5 Quick Example

```
MIAConfig cfg("settings.conf", constants::ConfigType::KEY_VALUE);
cfg.initialize();
int timeout = cfg.getInt("network.timeout");
```

2.9.6 Extensibility

New configuration formats can be supported by deriving new classes from `ConfigData` and implementing the required interface. `MIAConfig` will manage the polymorphic pointer and delegate appropriately based on the `ConfigType`.

2.10 Logging Framework

2.10.1 Overview

The `Logger` component provides centralized logging functionality for all MIA applications. It supports simple, thread-safe log message recording to either a default or user-defined log file, with optional verbosity to print messages to `stdout`.

2.10.2 Free Logging Functions

Two free-standing functions are provided for lightweight, context-independent logging:

- `logToDefaultFile(message, verbose = false)`
  Logs a message to a default log file (typically `MIA.log`). If `verbose` is `true`, the message is also printed to `stdout`.
- `logToFile(message, filename, verbose = false)`
  Logs a message to a specified file. If only a filename is provided (not a full path), the path is resolved using `paths::getDefaultLogDirToUse()`. Assumes the target directory exists. Supports optional verbosity.

2.10.3 The `Logger` Class

For more structured applications, the `Logger` class provides an object-oriented interface for managing log state:

- `Logger(filename = DEFAULT_LOG_FILE)`
  Constructor that initializes logging to the specified file. If not provided, defaults to `MIA.log`.
- `setLogFile(filename)`
  Changes the log file during runtime. Closes the existing stream and opens a new one.
- `log(message, verbose = false)`
  Logs a message to the current file. If `verbose` is enabled, also prints to `stdout`.
- `getLogFile()`
  Returns the current log file name in use.

2.10.4 Internals

Internally, the `Logger` class maintains:

- `currentLogFileName` – User-specified or default log filename.
- `currentLogFileFullPath` – Resolved full path for the log file.
- `logStream` – A mutable `std::ofstream` used for log writes.

The method `openLogFile()` is called during construction and when switching log files, ensuring the stream is always open and ready for use.

2.10.5 Summary

This logging framework simplifies and unifies log message handling across MIA tools. It supports both procedural and object-oriented usage styles, allows runtime control over log destinations, and integrates with the system's standard output for interactive diagnostics.

Chapter 3

MIAConfig Files

3.1 Introduction to MIAConfig

The `MIAConfig.MIA` file is a plaintext configuration file used to define the runtime behavior of the MIA application. It serves as a centralized source for main program variables and environment-specific settings. This file allows the behavior of the program(s) to be adjusted without requiring recompilation, making it suitable for use cases where the end-user either lacks access to the build system or needs to fine-tune behavior in different execution environments.

Variables stored in this file affect initialization, input/output paths, user preferences, and runtime flags. Because the file is read during program startup, any changes made to it will be reflected in subsequent executions unless the program is specifically designed to reinitialize the configuration.

3.2 Structure, Format, and Syntax

The format of the configuration file is simple and strictly line-based. Each setting must follow a key-value structure on a single line, with the key and value separated by an equals sign `=`. The correct format is essential for successful parsing. The configuration classes are designed so this can be expanded to other formats at a later time if needed.

General Rules

- Comment lines begin with the `#` character. These lines are ignored during parsing.
- Empty lines are allowed and ignored.
- Whitespace around the equals sign is not permitted and may result in incorrect key parsing.
- Values may contain spaces and special characters, and they will be interpreted as-is.
- All keys and values are treated as strings during parsing; type conversion occurs at access time.

Example

```
#============================================================================
# Name         : MIAConfig.MIA
# Author       : Antonius Torode
# Date         : 1/10/18
# Copyright    : This file can be used under the conditions of Antonius'
#                General Purpose License (AGPL).
# Description : MIA settings for program initialization.
#============================================================================

# File path variables
inputFilePath=Resources/InputFiles/
cryptFilePath=Resources/EncryptedFiles/
decryptFilePath=Resources/EncryptedFiles/
```

```
workoutsFilePath=Resources/InputFiles/exercises.txt
sequencesFilePath=Resources/InputFiles/MIASequences.txt
workoutOutputFilePath=Resources/OutputFiles/workout.txt
excuseFilePath=Resources/Excuses.txt

# Program behavior flags
verboseMode=false
MIATerminalMode=true

# ... Additional runtime variables below
```

3.3 How MIAConfig Files Are Handled at Runtime

At runtime, the MIA system utilizes the `MIAConfig` class to locate, read, and parse the configuration file. This class stores all configuration values internally in a key-value map structure and provides type-safe access methods to retrieve values in common formats (e.g., `int`, `bool`, `double`, `string`, and vectors of each type).

The following mechanisms are supported:

- If a full file path is provided to the `MIAConfig` object, that path is used directly.
- If only a filename is provided, the system will search default paths for a matching file.
- Once loaded, the configuration can be queried using accessor functions like `getString()`, `getInt()`, or `getVector()`.
- The configuration can be reloaded dynamically using the `reload()` method, allowing runtime updates.

Example Usage (C++)

```
MIA_System::MIAConfig config("MIAConfig.MIA");

std::string path = config.getString("inputFilePath");
bool verbose = config.getBool("verboseMode");
int retryCount = config.getInt("maxRetries");
std::vector<std::string> names = config.getVector("userList", ',');
```

3.4 Extensibility and Maintainability

The configuration system was designed to be flexible and extensible. Internally, all configuration is maintained as string-based key-value pairs, but typed access is provided by the interface. This design allows for future integration with other configuration formats (e.g., JSON or environment variables) or backend sources with minimal changes to consuming code. Additional configuration maps or derived classes can be introduced later without altering the core interface or behavior.

3.5 MIA Configuration (MIAC)

Previously, all configuration variables for MIA were contained within a single file called the MIAC (MIA Configuration file). This meant that settings for all applications and modules were centralized in one place. In newer versions of MIA, configuration has been modularized: each application now uses its own standalone configuration file. This separation improves organization, simplifies customization, and reduces the risk of conflicts between different application settings. At the time of writing this, this change is relatively new, so this section is here to explain this change in case the MIAC acronym is used.

Chapter 4

D0sag3 Command (D3C) Integration

4.1 D3C Introduction and Overview

The D3C encryption was an incorporation of an old encryption program I created many years ago as part of the D3C (d0sag3 command) program. The original code was made when I was first learning C++ and this was used as a project for educational purposes. The encryption algorithm utilizes random numbers, bit analysis, variable type conversions, and more.

4.2 d0s1 Encryption

The d0s1 encryption algorithm was the first implementation of encryption within the D3C program. The d0s1 algorithm is programmed solely to encrypt an input string value. To outline the algorithm that d0s1 uses, we will start with an example string "hello." The algorithm follows.

```
# Start with an input string
Hello

# Each character is examined individually.
H e l l o

# The string get's converted to integers based on the ascii value of each character.
72 101 108 108 111

# The integers are converted to a binary representation.
1001000 1100101 1101100 1101100 1101111

# A random number is generated for each character that existed.
103 70 105 115 81

# The random numbers are converted to binary representations.
1100111 1000110 1101001 1110011 1010001

# The string and random binary numbers are added to a trinary number.
   1001000 1100101 1101100 1101100 1101111
+  1100111 1000110 1101001 1110011 1010001
-----------------------------------------
=  2101111 2100211 2202101 2211111 2111112

# The random numbers selected before are converted to base 12 numbers.
103 70 105 115 81 = 87 5A 89 97 69

# The base 12 random numbers are placed at the end of the trinary string.
210111187 21002115A 220210189 221111197 211111269

# The ouput of the encryption is then these values.
2101118721002115A220210189221111197211111269
```

The encryption was meant to have a final stage to decrease the length of the output by assigning different characters to the number sequences output; however, this was never finished.

4.3 d0s2 Encryption

d0s2 encryption is a very similar algorithm to d0s1 with one major difference. The encryption of d0s2 requires a user input password that is added into the encryption process. The password and string are both encrypted and then added together in a way that the password is needed for quick decryption.

4.4 d0s3 Encryption

The d0s3 encryption algorithm was (as of the time writing this) never finished. The d0s3 was the actual D3C encryption that was originally desired with d0s1 and d0s2 being practice runs for the creator to experiment with C++ first before employing an actual complicated algorithm. MIA currently has parts of the d0s3 encryption programmed in but they are still in development and not yet deployed. The d0s3 encryption algorithm is not related to d0s1 and d0s3 but will instead have a unique and complicated algorithm that can encrypt entire files instead of just string values. The D3C encryption utilities, including d0s1, d0s2, and the partial implementation of d0s3, are included in MIA as general-purpose utilities. These components are designed to be accessible for use across various MIA applications and libraries, providing a consistent and centralized encryption interface where needed.

Chapter 5

Workout Generation

5.1 MIA Workout Generation Overview and Introduction

The workout generation is an application designed within MIA and is created for producing a workout with customization from the user. The generation of a workout within MIA has some dependencies on random value generation and thus is capable of creating different workouts each run. MIA is also capable of creating an entire weeks worth of workouts and outputting it to a file for the user. The entire generation process depends on a few input values, such as maximum number of sets, maximum number of exercises per set, and more which are all defined in a exercises file. Upon running the workout generation, the user enters a difficulty and MIA generates a workout with appropriate difficulty based on this number and the input file (see section 5.4 for more details).

Throughout this section, workout can be defined as the complete output generated by MIA containing some number of sets, some number of exercises per set, and some number of reps per exercises.

5.2 Workout Application Installation

The MIAWorkout application is built from source files `MIAWorkout.cpp` and `MIAWorkout_main.cpp`, with headers such as `MIAWorkout.hpp`. It links to core libraries including `Core_LIB`, `Types_LIB`, and `Math_LIB`.

During installation, if `SYSTEM_INSTALL` is enabled, the executable is installed to the directory specified by `$APP_INSTALL_LOCATION`. For release builds, the executable is placed in the path defined by `$RELEASE_INSTALL_LOCATION`. This configuration ensures the workout app is installed in the correct location depending on the build and system environment.

5.3 Input File and Defining Workouts

5.3.1 Input File

The MIA workout generation utilizes an input file to determine exercises, exercise weighted values and various generation values. By default, this file is `resources/MIAConfig.MIA` but this file name and path can be changed via the configuration file parameters. The contents of this input file look similar to the following.

```
#================================================================================
# Name        : MIAConfig.MIA
# Author      : Antonius Torode
# Date        : 1/10/18
# Copyright   : This file can be used under the conditions of Antonius'
#                General Purpose License (AGPL).
# Description : MIA settings for program initialization.
#================================================================================

# Path for output files.
workoutOutputFilePath=resources/outputFiles/workout.txt
```

```

# Increasing this value provides a global increase to difficulty. Recommended value is 0.1.
# The maxNumOfExercises cannot exceed the number of exercises defined. Default values is inf (which picks the maximum allowed).
# Put these variable before all defined exercises.
toughness = 0.1
minNumOfExercises = 3.0
maxNumOfExercises = inf
minNumOfSets = 1.0
maxNumOfSets = 10.0

##############################################
#### Define exercises below this point
##############################################

# Exercises and weights.
push_up = 10.0; reps
push_up_mixed = 8.0; reps
push_up_weighted = 5.0; reps
push_up_diamond = 7.0; reps
push_up_jump = 5.0; reps
sit_up = 15.0; reps
sit_up_inclined = 10.0; reps
crunch = 5.0; reps
leg_lift = 10.0; reps
pull_up = 2; reps
pull_up_weighted = 1; reps
split_jump = 5.0; reps
squat = 3.0; reps
squat_weighted = 1.0; reps
jumping_jack = 30.0; reps
running = 0.1; miles
dips = 5.0; reps
wall_sit = 20.0; seconds
plank = 30.0; seconds
lunge = 3.0; reps
knee_jump = 5.0; reps
burpee = 3.0; reps
squirpee = 3.0; reps
chin_up = 3.0; reps
chin_up_weighted = 2.0; reps
punches = 10.0; seconds
russian_twist = 10.0; seconds
```

See chapter 3 for more specific details on the configuration files. This file must be in the correct format in order for the MIA workout generation to function properly. First, commented lines are created using the '#' character. These lines are ignored by MIA when running internal algorithms. Within this input file, spaces are not important. Upon initialization, the MIA program will ignore all spaces within this file. Next, there are a few variables that the user can customize and define within this file which are below.

5.3.2 Workout Generation Parameters

```
toughness = 0.1
minNumOfExercises = 3.0
maxNumOfExercises = inf
minNumOfSets = 1.0
maxNumOfSets = 10.0
```

These values must appear in the input file before any defined exercises. To begin, toughness is a global variable that helps define the number of reps MIA will output per workout chosen. Increasing

this value is a global increase to the workout generation difficulty. The default value for toughness is 0.1 (see section 5.4 for more details). Next, There are minNumOfExercises and maxNumOfExercises variables which are used to determine the minimum number of exercises MIA will choose per set and the maximum number of exercises MIA can choose per set. The maxNumOfExercises value is read in such that a value of 'inf' is allowed. If 'inf' is selected, MIA will set the total number of exercises within the input file to be the maximum. Similarly, there are minNumOfSets and maxNumOfSets values which work in identical ways to minNumOfExercises and maxNumOfExercises only defining a minimum and maximum for number of sets per generated workout instead of number of exercises per set.

Note. At the time of writing this, the MIA program is not designed to account for a value of maxNumOfExercises that is larger than the actual number of exercises defined in the input file.

5.3.3 Defining Exercises

Following these program variables, the main part of the input file is the defined exercises. The exercises are defined similar to below.

```
# Exercises and weights.
push_up = 8.0; reps
sit_up = 15.0; reps
pull_up = 0.75; reps
squat = 3.0; reps
running = 0.1; mile
jumping_jack = 30.0; reps
```

Each exercise is defined using a common form. As shown below, the line must begin with an exercise name. Following this comes an equal sign and then a weighted value. This weighted value is defined to be relative to all other weighted values. This mean that in the above example, the file is claiming 8.0 push ups are equivalent to 15.0 sit ups, and similarly, 0.75 pull ups, etc. The MIA program will assume and each weight value for each exercise is of the same real world difficulty to the user. Following the weighted value must come a semi-colon and then a unit. The equal sign and semi-colon are important because they define how MIA separates the values.

```
# Proper format for defininD an exercise in the input file.
exercise_Name = exercise_Weighted_Value; exercise_Unit
```

5.4 Generation Algorithm

This section contains an outline of the algorithm used to generate the MIA workouts. The MIA generation is based on creating two curves (maximum and minimum) for a parameter and then deciding upon which parameter to use for a workout by taking a random value between both curves. For the purposes of this section, we will denote a random number between two values q_1 and q_2 as $rand[q_1, q_2]$. We will denote a complete workout with W.

5.4.1 Number of Sets Per Workout

The number of sets, denoted $S(s_{min}, s_{max}, d) \equiv S$ is dependent on three variables. The first two are from the input file which are minNumOfSets, denoted s_{min} and minNumOfSets, denoted s_{max}. The last is the difficulty d which is input by the user upon generation. The maximum number of sets was originally based on a linear increase, however for better optimization of the real world workout difficulties, a custom algorithm was created. The maximum $S_{max}(s_{min}, s_{max}, d)$ and minimum $S_{min}(s_{min}, s_{max}, d)$ number of sets per workout are given by

$$S_{max}(s_{min}, s_{max}, d) \equiv S_{max} = \frac{s_{max} - s_{min}}{10^{4/3}} d^{2/3} + s_{min} \tag{5.1}$$

$$S_{min}(s_{min}, s_{max}, d) \equiv S_{min} = \frac{s_{max} - 1.9 \times s_{min}}{1.9 \times 10^{4/3}} d^{2/3} + s_{min}. \tag{5.2}$$

Thus since $S(s_{min}, s_{max}, d)$ is a random value between these curves, we have

$$S_{ave}(s_{min}, s_{max}, d) \equiv S_{ave} = \frac{S_{max} + S_{min}}{2} \tag{5.3}$$

$$= \frac{(2.9s_{max} - 3.8s_{min})}{3.8 \times 10^{4/3}} d^{2/3} + s_{min} \tag{5.4}$$

$$S(s_{min}, s_{max}, d) = rand[S_{min}(s_{min}, s_{max}, d), S_{max}(s_{min}, s_{max}, d)]. \tag{5.5}$$

These are shown in Figure 5.1. The algorithm used to determine the sets per workout is identical to that of determining the number of exercises per set.

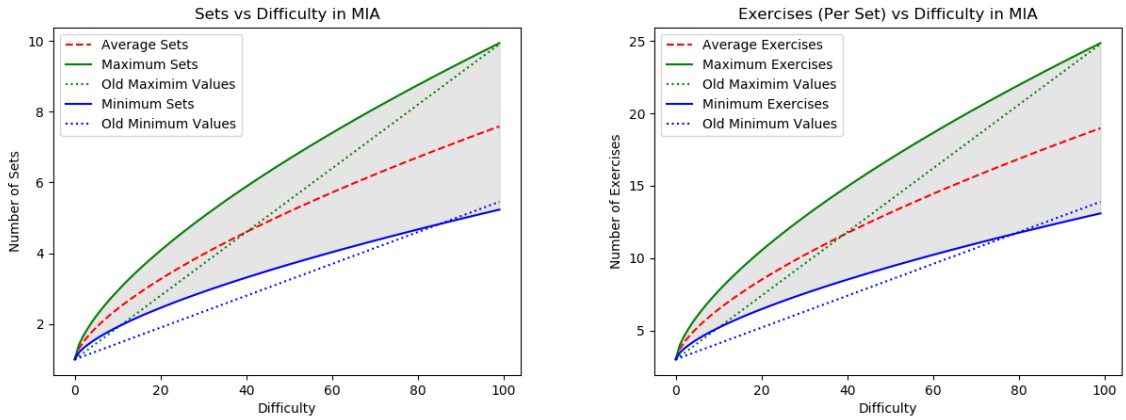

Figure 5.1: Number of sets per workout (left) and number of exercises pet set (right) based on the user input difficulty. The small dotted lines represent the original algorithm which was a simple linear increase in difficulty for each parameter. For the above two figures, values of $s_{min} = 1.0$, $s_{max} = 10.0$, $e_{min} = 3.0$ and $e_{max} = 25.0$ were used. The possible values for S and E are shown via the gray shaded areas.

5.4.2 Number of Exercises Per Set

The number of exercises, denoted $E(e_{min}, e_{max}, d) \equiv E$ is dependent on three variables. The first two are from the input file which are minNumOfExercises, denoted e_{min} and maxNumOfExercises, denoted e_{max}. The last is the difficulty d which is input by the user upon generation. The maximum number of sets was originally based on a linear increase, however for better optimization of the real world workout difficulties, a custom algorithm was created. The maximum $E_{max}(e_{min}, e_{max}, d)$ and minimum $E_{min}(e_{min}, e_{max}, d)$ number of sets per workout are given by

$$E_{max}(e_{min}, e_{max}, d) \equiv E_{max} = \frac{e_{max} - e_{min}}{10^{4/3}} d^{2/3} + e_{min} \tag{5.6}$$

$$E_{min}(e_{min}, e_{max}, d) \equiv E_{min} = \frac{e_{max} - 1.9 \times e_{min}}{1.9 \times 10^{4/3}} d^{2/3} + e_{min}. \tag{5.7}$$

Thus since $E(e_{min}, e_{max}, d)$ is a random value between these curves, we have

$$E_{ave}(e_{min}, e_{max}, d) \equiv E_{ave} = \frac{E_{max} + E_{min}}{2} \tag{5.8}$$

$$= \frac{(2.9e_{max} - 3.8e_{min})}{3.8 \times 10^{4/3}} d^{2/3} + e_{min} \tag{5.9}$$

$$E(e_{min}, e_{max}, d) = rand[E_{min}(e_{min}, e_{max}, d), E_{max}(e_{min}, e_{max}, d)]. \tag{5.10}$$

These are shown in Figure 5.1. Since this value depends on each set, and there are generally numerous sets per workout, we denote the set within the workout with i such that $1 \leq i \leq S$, where S is the total

number of sets within a workout. Using this index, E_i becomes

$$E_{i,max}(e_{min}, e_{max}, d) \equiv E_{max} = \frac{e_{max} - e_{min}}{10^{4/3}} d^{2/3} + e_{min} \tag{5.11}$$

$$E_{i,min}(e_{min}, e_{max}, d) \equiv E_{min} = \frac{e_{max} - 1.9 \times e_{min}}{1.9 \times 10^{4/3}} d^{2/3} + e_{min} \tag{5.12}$$

$$E_{i,ave}(e_{min}, e_{max}, d) \equiv E_{ave} = \frac{E_{max} + E_{min}}{2} \tag{5.13}$$

$$= \frac{(2.9e_{max} - 3.8e_{min})}{3.8 \times 10^{4/3}} d^{2/3} + e_{min} \tag{5.14}$$

$$E_i(e_{min}, e_{max}, d) = rand[E_{i,min}(e_{min}, e_{max}, d), E_{i,max}(e_{min}, e_{max}, d)]. \tag{5.15}$$

Following this, the average number of exercises done for a given workout would be

$$E_W = \frac{1}{S} \sum_{i=1}^{S} E_i. \tag{5.16}$$

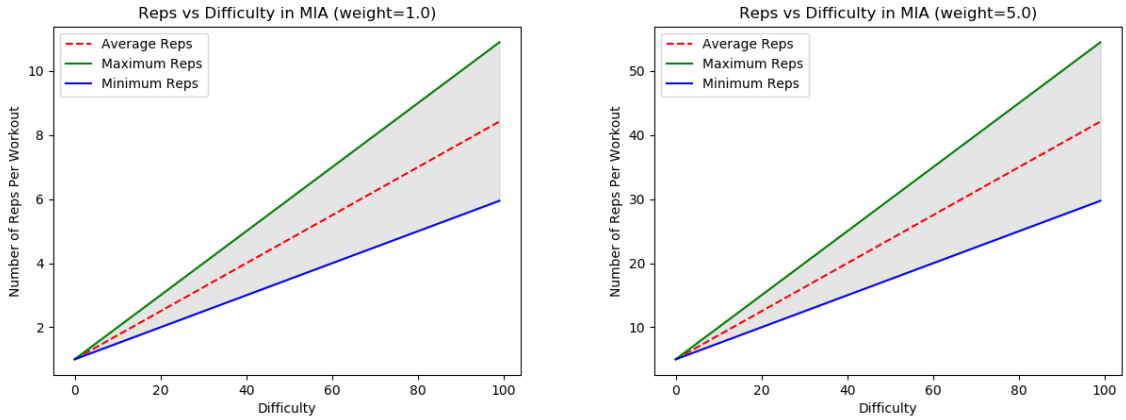

Figure 5.2: Number of reps per exercise. On the right, we have $R(0.1, d, 1)$ and on the left, $R(0.1, d, 5)$. The possible values for R are shown via the gray shaded area.

5.4.3 Number of Reps Per Exercise

The number of reps, denoted $R(t, d, w) \equiv R$ is dependent on three variables. The first is toughness t which is gathered from the input file or defaults to 0.1. The second is difficulty d which is input by the user upon generation. Lastly, the weight w of a given exercise is needed to provide the total reps that are output. The reps are determined by a linear trend given by

$$R_{max}(t, d, w) = tdw + w \tag{5.17}$$

$$R_{min}(t, d, w) = \frac{tdw}{2} + w \tag{5.18}$$

$$R_{ave}(t, d, w) = \frac{R_{max} + R_{min}}{2} = \frac{3tdw}{4} + w \tag{5.19}$$

$$R(t, d, w) = rand[Rmin(t, d, w), R_{max}(t, d, w)]. \tag{5.20}$$

These are shown in Figure 5.2. Since this value depends on each exercise, and there are generally numerous exercises per set, we denote the exercises within the set by an index j such that $1 \leq j \leq E_i$, where E_j is the total number of exercises within the given set i. Using this index, R becomes

$$R_{j,max}(t, d, w_j) = tdw_j + w_j \tag{5.21}$$

$$R_{j,min}(t, d, w_j) = \frac{tdw_j}{2} + w_j \tag{5.22}$$

$$R_{j,ave}(t, d, w_j) = \frac{R_{max} + R_{min}}{2} = \frac{3tdw_j}{4} + w_j \tag{5.23}$$

$$R_j(t, d, w_j) = rand[R_{j,min}(t, d, w_j), R_{j,max}(t, d, w_j)]. \tag{5.24}$$

Following this, the average number of normalized reps done per set would be

$$R_{i,ave} = \frac{1}{E_i} \sum_{j=1}^{E_i} \frac{R_j}{w_j}. \tag{5.25}$$

Then, the average number of reps (normalized by the weights) done per workout would be given by

$$R_W = \frac{1}{S} \sum_{i=1}^{S} \frac{1}{E_i} \sum_{j=1}^{E_i} \frac{R_j}{w_j} \tag{5.26}$$

5.5 Real World Difficulties

Due to the way that we set up the weighted system for each exercise, we can easily determine how difficult, denoted D a workout is in reality based upon the generation. First, a workout is directly proportional to the number of sets it contains and thus $D \propto S$. Similarly, the difficulty of each set is proportional to the number of exercises are contained within each set and thus we use $D \propto E_W$. Lastly, the difficulty of each exercise is proportional to the number of normalized reps, and thus $D \propto E_W$. By combining all of these components we get

$$D \equiv SE_W R_W = S \left(\frac{1}{S} \sum_{i=1}^{S} E_i \right) \frac{1}{S} \sum_{i=1}^{S} \frac{1}{E_i} \sum_{j=1}^{E_i} \frac{R_j}{w_j}. \tag{5.27}$$

To demonstrate the possible values that can be output by D we can examine the maximum and minimum values. The maximum value would be when S, E, and R are at their maximums. Thus, we have

$$D_{max} = S_{max} \left(\frac{1}{S_{max}} \sum_{i=1}^{S_{max}} E_{i,max} \right) \frac{1}{S_{max}} \sum_{i=1}^{S_{max}} \frac{1}{E_{i,max}} \sum_{j=1}^{E_{i,max}} \frac{R_{j,max}}{w_j}. \tag{5.28}$$

Since each $E_{i,max}$ and $R_{j,max}/w_j$ are the same for all i, j respectively, then we can simplify this to

$$D_{max} = S_{max} \left(\frac{1}{S_{max}} S_{max} E_{max} \right) \frac{1}{S_{max}} S_{max} \frac{1}{E_{max}} E_{max} \frac{R_{max}}{w} \tag{5.29}$$

$$= S_{max} E_{max} \frac{R_{j,max}}{w_j}. \tag{5.30}$$

Similarly, the minimum value is

$$D_{min} = S_{min} E_{min} \frac{R_{j,min}}{w_j}, \tag{5.31}$$

where $R_{j,min}/w_j$ represents the normalized reps for each exercise in both of the above two cases. The possible values of D then lie between these two curves and can be seen in figure 5.3.

The real world difficulty D has an exponential increase. This is desired for a specific reason. As improvements are made, meaning as ones physique and ability to perform improves, a greater challenge is needed. Similarly, there is a much larger variance in the real world difficulty D increases with respect to the input difficulty d. This is because as workout intensities increase, there is a desire to keep the body both guessing and not straining too often. Therefore, by increasing the variability of a workout, there is a greater effectiveness and an improved 'rest' or 'slow' period between intensive workouts.

5.6 Notes on Appropriate MIA Parameters For Usage

Based on the way MIA generates workouts (as described in the above sections), there are a few things to keep in mind when determining the proper settings. First, if a maxNumberOfExercises value of 'inf' is used, then the real world difficulty output will be proportional to the number of exercises defined in the input file. Therefore, if one places a thousand different workouts in this file, each difficulty will be drastically more intense than if there were only 25. Thus, it is important to experiment with this value and adjust accordingly based on the number of defined exercises you have in the input file.

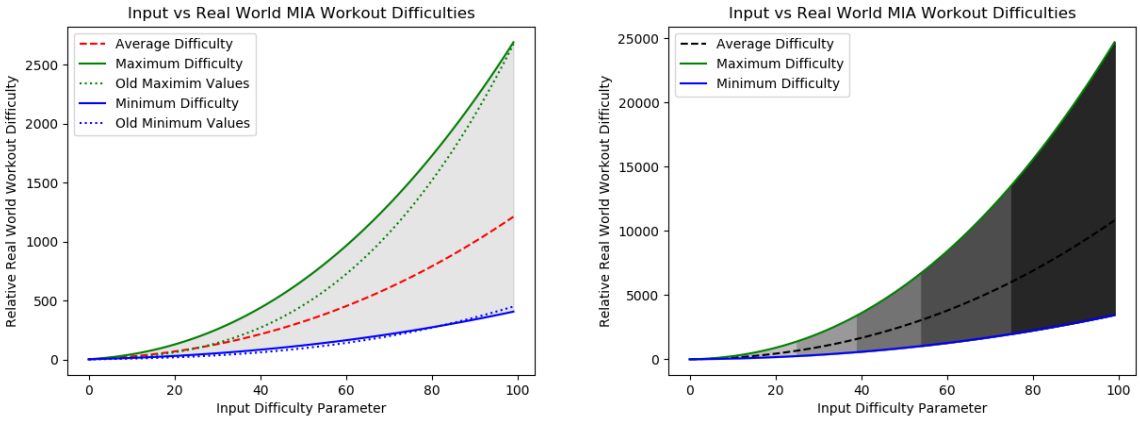

Figure 5.3: The real world difficulties D output by the MIA workout generation program versus the input difficulty parameter (LEFT). The possible values for D are shown via the gray shaded area. The old min/max values represent D based on the old S and E values (see figure 5.1). On the right is the same plot only depicting the difficulty ranges. The ranges run from very easy (light gray), to very hard (dark gray).

Chapter 6

World of Warcraft (WoW) Features in MIA

6.1 WoW Fishbot

Note. This fishbot was made for educational purposes only! Do not use this in the game or you may be banned as it violates the license agreement! As of World of Warcraft (WoW) version 7.2, this fishbot is fully functional.

The World of Warcraft (WoW) fishbot implementation within MIA was created solely for educational purposes. This fishbot violates the terms of use of WoW and should only be used at your own risk. The fishbot is designed to simulate fishing for ones character in WoW.

6.1.1 Setup and Configuration

Before running the fishbot, there are a few things that need to be properly configured, otherwise the bot will either not work or not work efficiently. First, one must disable the hardware cursor in the system settings. This can be done by pressing escape, system, advanced, then setting hardware cursor to disable (see figure 6.1). Be sure to click apply before closing out of the settings.

```
ESC > System > Advanced > Hardware Cursor > DISABLE > Apply
```

Figure 6.1: Visual representation of the location of the option for disabling the hardware cursor. In order for the WoW fishbot to work, this option must be disabled.

After disabling the hardware cursor, the coordinates for cast locations need to be set. It is recommended that the user zoom into first person when using the fishbot though this is not required for functionality. First, place the cast button on the action bar (see figure 6.2). By default, the fishbot uses button 3 for cast, but upon running the fishbot the user will be prompted to change the settings. Similarly, a lure can be added to the bar, which by default is placed in action bar slot 8. A lure is optional and the fishbot program will ask if one is being used upon runtime.

Figure 6.2: Visual representation of the location of the cast option on the action bar.

The WoW-specific configuration variables are now contained in a separate configuration file named `WoWConfig.MIA`. This file includes several settings related to WoW apps and tools within MIA, including variables for the fishbot implementation. Unlike the previous defaults embedded in MIAC (in prior MIA versions), these values are centralized here for easier customization. Some of the fishbot-related variables, along with their current defaults, are shown below:

```
# Variables relating to the fishbot implementation inside MIA.
WoWFishBotStartX=791
WoWFishBotStartY=443
WoWFishBotEndX=1082
WoWFishBotEndY=581
```

Figure 6.3: The red square represents the region in which the casted bobber lands. The two points labeled A and B are positions that one would want to set as the proper coordinates in the configuration file.

These values, `WoWFishBotStartX, WoWFishBotStartY, WoWFishBotEndX, WoWFishBotEndY`, define the area that the fishbot will search for the fish bobber within. The first two parameters, `WoWFishBotStartX, WoWFishBotStartY` define the coordinates of the point A in figure 6.3. The second two parameters, `WoWFishBotEndX, WoWFishBotEndY` define the coordinates of the point B in figure 6.3. The MIA fishbot will search this area for the bobber during operation. The best method to determine the proper coordinates to use is to spam the cast option and observe the area that the bobber lands. The suggested coordinates for A and B are near where they are located in figure 6.3, however any coordinates can be used. To determine the proper coordinates, one can use the `find mouse` command in MIA which will determine the coordinates at the location of ones cursor.

After these parameters are set, the fishbot can be run in MIA by using the `fishbot` command. There are a few other variables and parameters that can be set by the user but the others are optional. These other optional parameters that are contained in the configuration file are as follows.

```
# Variables relating to the fishbot implementation inside MIA.
WoWFishBotIncrement=40
WoWFishBotNumOfCasts=10000
WoWFishBotDelay=10000
```

First, the `WoWFishBotIncrement` variable defines the step size that the MIA program will search for the bobber by. This can be seen in figure . A smaller step size will cause the fishbot to find the bobber slower but more accurate, whereas a faster step size will cause a faster search but is less accurate. The default value for this is 40, but should be decreased if the bobber is missed by the fishbot. The next parameter, `WoWFishBotNumOfCasts` is how many times the fishbot will cast before ending it's program. This can be whatever the user desires.

Figure 6.4: The red square similar to that shown in figure 6.3 partitioned into squares of size defined by the `WoWFishBotIncrement` value.

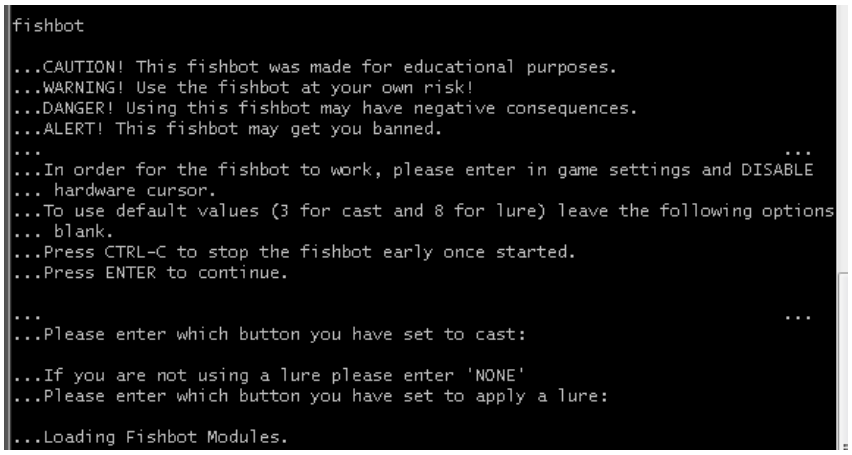

Figure 6.5: A snapshot of the MIA fishbot upon runtime. As of MIA version 0.041.

The last parameter, `WoWFishBotDelay` is what defines how fish are caught. The MIA fishbot catches fish by chance. There is a plan to improve the method to catch fish by, but for now the program waits a specific number of milliseconds after finding the bobber before clicking it. Thus, there is a certain probability that a fish is caught. By default this value is 10000ms.

To run the fishbot simply use the `fishbot` command in the MIA terminal. Upon running this command, the fishbot will ask the use to enter the required information (see figure 6.5)

6.2 Mailbox Management

Note. This program technically violates the blizzard license agreement and should be used with caution. However, it is no more complicated than an addon. I just happen to know C++ and not lua.

MIA contains some in game mail management for World of Warcraft (WoW) in order to assist in the process of creating in game letters in bulk. In game letters can't be sent between characters

and then looted and stored in the character inventories. In some cases, such as role playing (RP) or guild recruitment, these letters may be desired by bulk. The process of creating these letters consists of entering in a letter recipient, subject, the contents of the letter, and then hitting a send button (see figure 6.6). After sending these letters, however many times it is done, they then need to be looted on the character that they were sent to. This consists of opening up the mailbox, clicking on the letter received, acquiring the physical copy by looting it from the opened letter, then deleting the letter to clean up the mailbox (see figure 6.7). MIA is designed to automate this entire process.

6.2.1 Sending Duplicates of a Letter

As described above briefly, MIA has the capability of automating the sending of duplicate letters to a recipient in WoW. To do this, there are a few settings that need to be configured on the user end to ensure proper functionality. There are two variables contained in the configuration file that are used in conjunction with the `wow dup letter` function in MIA. These are listed below.

```
# WoW related variables.
WoWMailboxSendLetterLocationX=279
WoWMailboxSendLetterLocationY=647
```

By default, these values are set for the environment that was used when originally programmed into MIA. Both of these variables are used to represent the location of the send button shown in figure 6.6. The first, `WoWMailboxSendLetterLocationX` represents the x coordinate and the latter `WoWMailboxSendLetterLocationY` represents the y coordinate. Both of these values can be found by using the `find mouse` function in MIA.

Figure 6.6: A screen shot of the WoW in game outgoing mailbox menu. The mail management within MIA utilizes the fields indicated by red squares in this menu. The red box around the send button represents the `WoWMailboxSendLetterLocation` location.

Once the proper coordinates are set, one can automate this process of sending a duplicate letter by using the `wow dup letter` command in the MIA terminal. The `wow dup letter` has a few limitations when used. First, the recipient of the letter can only contain normal characters such as a, e, o, etc. The recipient cannot contain special characters such as ä, é, õ, etc. However, the contents of the message that will be sent can contain special characters. The desired message contents should be copied to the clipboard before running the `wow dup letter` command. Upon running the command, the terminal will prompt the user to do so as well. The subject field will automatically be filled in with "subject."

6.2.2 Unloading Duplicated letters

MIA contains a command `wow unload` which is designed to be used in conjunction with the `wow dup letter` command. This command will loot the letters from the incoming mailbox that are sent using the `wow dup letter` command. In order to use this command properly, there are six different variables within the configuration file that need to be determined for the users environment. These variables are below.

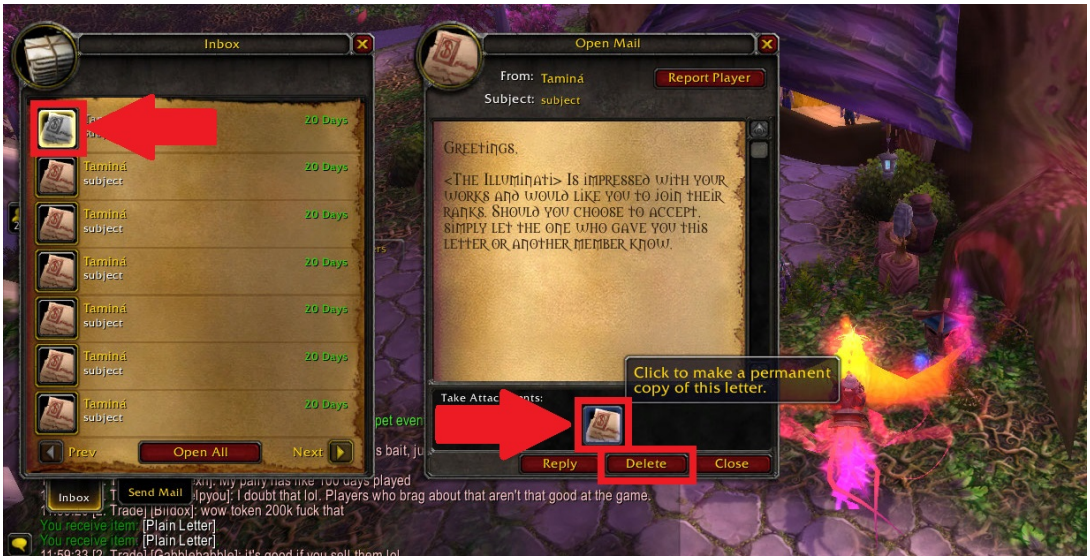

Figure 6.7: A screen shot of the WoW in game incoming mailbox menu. The mail management within MIA utilizes the fields indicated by red squares in this menu. The left arrow indicates the `WoWMailboxFirstLetterLocation` position. The right arrow indicates the `WoWMailboxLootLetterLocation` position. The square around the delete button represents the `WoWMailboxDeleteLetterLocation` location.

```
# WoW related variables.
WoWMailboxFirstLetterLocationX=55
WoWMailboxFirstLetterLocationY=265
WoWMailboxLootLetterLocationX=675
WoWMailboxLootLetterLocationY=600
WoWMailboxDeleteLetterLocationX=700
WoWMailboxDeleteLetterLocationY=650
```

By default, these variables are set to values that were specific to the programmers environment. The variables need to be set based off of three different coorindates. The first, `WoWMailboxFirstLetterLocation` corresponds to the location of the first letter in the inbox of the user (see figure 6.7). The second, `WoWMailboxLootLetterLocation` corresponds to the location of the letter to loot from the user inbox (see figure 6.7). The last, `WoWMailboxDeleteLetterLocation` represents the locations of the delete button on a letter in the WoW inbox (see figure 6.7). For all three coordinates, there is an x and y value (represented by the variables in the configuration file) which can be determined through MIA by using the `find mouse` command.

Chapter 7

Sequencer

The sequencer is an app contained within MIA for performing key/button simulation sequences. This can be used for Performing mundane tasks automatically and repeatable tasks on a timer while away from the computer. More advanced uses can also be for botting and other repeatable tasks.

7.1 Using the Sequencer

The sequencer acquires its programmable sequence functionality from the `MIASequences.MIA` file. This file will appear similar to the following.

```
#================================================================================
# Name        : MIASequences.txt
# Author      : Antonius Torode
# Date        : 12/26/2019
# Description : MIA combinations for button sequences.
#================================================================================

# This file is formatted similar to the other MIAConfig files.
# Create a commented line using the '#' character.
# Comments must be on their own line.
# This file must be of the proper format to work with MIA.
# Create a combination name using 'SEQUENCENAME=name'.
# Define the DELAY between sequence actions with 'DELAY=3000', units are
    milliseconds.
# After declaring a name for the command sequence, define the command sequence
    using
# any number of the following possible actions in the desired order.
#----------------------------------------
# TYPE=abc123
# SLEEP=200
# MOVEMOUSE=xxx,yyy
# CLICK=RIGHTCLICK
#----------------------------------------
# Actions and program variables should be capitalized.
# The sequence name must be defined at the start of a sequence.
# The end of a sequence must be defined by ENDOFSEQUENCE.
# See the documentation for a full list of valid types and options.

#Sequence definitions below...
```

The sequencer app has two main modes of operation. If ran without the `-sequence` flag specified, the sequencer will load a front end UI where it prompts the user for a sequence to run. The sequence names are defined when the sequence is created at app initialization based on the `MIASequences.MIA` file input. If the `-sequence` flag is specified, the program will instead default to running the sequence entered via the flag. Other options are handled appropriately.

```
$ ./MIASequencer -h
```

```
Base MIA application options:
   -v, --verbose           Enable verbose output.
   -h, --help              Show this help message

MIATemplate specific options:
   -c, --config            Specify a config file to use (default = /etc
      ↪ /mia/MIASequences.MIA)
   -s, --sequence          Run a sequence, then exit.
   -t, --test              Enables test mode. This mode will only
      ↪ output the sequence to terminal.
```

Using the `-test` flag enables a dry-run mode. In this mode, no actual input events are triggered. Instead, each parsed sequence and action is printed to the terminal for review. This is useful for verifying sequence definitions without affecting the system.

7.2 Defining a Sequence

A sequence is defined by first creating a sequence name using the `SEQUENCENAME` keyword and ending with the `ENDOFSEQUENCE` keyword. Each line in the configuration should represent a valid *action*. The following example will demonstrate all of the valid actions available in a sequencer file. The descriptions of what each line do are below.

```
# This combination is for testing.
SEQUENCENAME=test
DELAY=3000
TYPE=abcdefg
SLEEP=500
MOVEMOUSE=145,887
CLICK=RIGHTCLICK
ENDOFSEQUENCE
```

1. `SEQUENCENAME=test` This line creates a sequence with the name “test”.
2. `DELAY=3000` This sets the timing between each event in the sequence. The units are in milliseconds.
3. `TYPE=abcdefg` This command will type the text entered. In this case “abcdefg” will be typed.
4. `SLEEP=500` This defines the time to sleep when the hover keyword is used.
5. `MOVEMOUSE=145,887` This is a command signaling to move the mouse to the coordinates (145,887).
6. `CLICK=RIGHTCLICK` This is a command signaling to perform a right click.
7. `ENDOFSEQUENCE` This ends the sequence and prepares the sequencer for a new defined sequence.

The below table shows the valid action keywords and their valid values.

Keyword	Input Type	Valid Values / Description
`SEQUENCENAME`	String	A unique identifier for the sequence. Must be defined at the start of each sequence.
`DELAY`	Integer (ms)	Delay between actions in milliseconds.
`TYPE`	String	Any string of characters to be typed as keystrokes.
`SLEEP`	Integer (ms)	Sleep/pause time in milliseconds before next action.
`MOVEMOUSE`	Integer pair	X,Y screen coordinates to move the mouse to. Format: `MOVEMOUSE=x,y`, i.e., `MOVEMOUSE=145,887`
`CLICK`	Enum (ClickType)	One of: `LEFTCLICK`, `RIGHTCLICK`, `MIDDLECLICK`
`ENDOFSEQUENCE`	None	Terminates the current sequence definition. Must appear at the end of a sequence.

7.3 Notes on Using The Sequencer

At the time of writing this, the sequencer is completely new and yet to be fully tested. It will continue to be improved as it is used. It is important to follow the scheme above for defining sequences above closely as errors and bugs are not yet determined. It is also important that all needed parameters be defined properly and no sequences have a duplicate name. The program is not programmed to handle this as of yet.