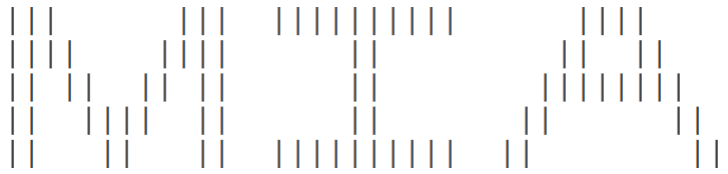

Multiple Integrated Applications (MIA) Manual & Programming Documentation

Developer: Antonius Torode

Version – Version 2.030
Latest update: June 28, 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: June 28, 2025

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

Contents

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.

1.2 MIA Features

1.2.1 MIA Application Framework

One of the main features of MIA is the core framework and internal setup. The MIA project's core framework and modular architecture form the backbone of its design, providing a robust, extensible, and developer-friendly ecosystem for building and managing applications. This framework enforces a standardized application life cycle, ensures clear dependency management, and integrates seamlessly with supporting systems such as error handling, configuration, and logging. By leveraging modern C++20 features, the core framework promotes type-safe, maintainable, and scalable application development, making it a cornerstone feature of MIA.

The core framework is built around several interconnected components that collectively enhance modularity and usability:

- **Layered Dependency Structure:** The project is organized into distinct layers—Core, Utilities, Libraries, and Applications—to ensure a unidirectional dependency flow. The Core layer provides fundamental types, error codes, and logging interfaces with no external dependencies. Utilities depend on Core, offering reusable functions like file and string operations. Libraries build upon Core and Utilities for domain-specific logic, such as mathematical functions, while Applications integrate all layers to create executable programs. This structure prevents circular dependencies, enhancing maintainability and scalability. See section 2.1 for more information.
- **Application Framework:** The framework defines a consistent life cycle for applications, requiring `initialize` and `run` methods, enforced at compile-time using C++20 concepts (`AppInterface`).

The `MIAApplication` base class standardizes command-line argument parsing (e.g., `-v` for verbose, `-h` for help) and provides a template function (`runApp`) and macro (`MIA_MAIN`) to streamline application entry points, reducing boilerplate code and ensuring uniform behavior. See section 2.6 for more information.

- **Supporting Systems:** The framework integrates with robust subsystems:
  - **Error Handling:** The `MIAException` class and `ErrorCode` enumeration provide structured error propagation and descriptive reporting (see section 2.9).
  - **Configuration:** The `MIAConfig` class supports multiple formats (e.g., key-value, raw lines) with typed accessors and extensible design (see section 2.11).
  - **Logging:** The `Logger` class and free functions enable thread-safe logging to customizable files with optional verbose output (see section 2.12).
  - **Global Constants and Paths:** The `constants` and `paths` namespaces centralize version information and resource locations, adapting to installed or repository-based environments (see Section **??**).
- **Build and Deployment:** A flexible Bash script (`build.sh`) and CMake integration automate building, installing, and managing dependencies, supporting platform-specific configurations (primarily Linux, with planned expansions) and release builds (see Sections 2.2 and 2.5).

Benefits

The core framework offers several advantages for developers and end-users:

- **Modularity and Maintainability:** The layered architecture ensures clear separation of concerns, making it easier to modify or extend individual components without affecting others.
- **Type Safety and Consistency:** C++20 concepts enforce interface compliance at compile-time, reducing runtime errors and ensuring uniform application behavior.
- **Developer Productivity:** Standardized interfaces, macros, and automated build tools minimize boilerplate code and streamline development workflows.
- **Extensibility:** The framework's design supports adding new features, configuration types, and platform support with minimal changes to existing code.
- **Robustness:** Integrated error handling, logging, and configuration systems provide reliable diagnostics and flexible runtime behavior.

Summary

The MIA core framework and modular architecture provide a powerful foundation for building scalable, maintainable applications. By combining a layered dependency structure, a standardized application life cycle, and robust supporting systems, it enables developers to create feature-rich tools with minimal overhead while ensuring reliability and extensibility for future growth.

1.3 Repository Overview

The current project repository can be found at the following link:

`https://github.com/torodean/MIA`

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/`.

1.3.1 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 & Framework Features

This chapter provides an overview of the core features, architectural design, and internal conventions of the MIA development framework. It is strongly recommended that all developers read this chapter to gain a clear understanding of the various components available within the framework and how they interact. Familiarity with these features will streamline development, ensure consistency across modules, and enable effective use of the utilities and services provided by the MIA suite.

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. Details on utility modules can be found in Chapter 4. Details on domain-specific libraries are in Chapter 5.

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 2.5 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 C++ Macros

Macros are used sparingly within the framework, only when necessary to achieve concise and consistent code; whenever possible, inline functions or templates are preferred to maintain type safety and improve debuggability. Macros are employed within the application framework to provide concise, reusable code snippets that simplify common tasks and improve maintainability. They allow:

- **Code Reusability:** Encapsulate repetitive code patterns, reducing duplication and the chance for errors.
- **Consistent Behavior:** Enforce standardized procedures (e.g., logging conventions) across multiple classes and methods.
- **Ease of Use:** Provide simple interfaces for complex operations, enabling developers to invoke functionality with minimal syntax.
- **Conditional Logic:** Enable compile-time or runtime conditional behavior without adding verbose control flow code in the method body.

While macros can have drawbacks such as reduced type safety and debugging complexity, careful design and limited scope within the framework help mitigate these issues. The framework favors macros for lightweight abstraction where inline functions or templates would be overly verbose or less practical.

2.4 Cross-Platform Support

The `MIA` project is designed to be cross-platform, with support for both Windows and Linux systems (and maybe more at later dates). This portability is achieved in part through the use of preprocessor macros that enable platform-specific compilation paths where necessary.

A key component of this system is the use of centralized platform detection macros defined in the `constants.hpp` file, which is included in the `Framework_CORE` CMake library. These macros simplify platform checks throughout the codebase by abstracting away verbose compiler-defined flags. This code (which will look something like the example below) defines platform-specific macros, `IS_WINDOWS` and

`IS_LINUX`, based on standard compiler-defined macros. The purpose is to centralize platform detection logic in one place for easier maintenance and extension when supporting additional platforms.

By defining `IS_WINDOWS` when any common Windows-related macros are detected, and `IS_LINUX` when the Linux macro `__linux__` is detected, the code enables simpler conditional compilation. Developers can then write platform-specific code using straightforward checks like `#ifdef IS_WINDOWS` or `#if defined(IS_LINUX)` rather than repeating complex or lengthy preprocessor conditionals throughout the codebase.

```
#if defined(WIN32) || defined(_WIN32) || defined(__WIN32__) || defined(__NT__)
    || defined _WIN32 || defined _WIN64 || defined __CYGWIN__
    #define IS_WINDOWS
#elif defined(__linux__)
    #define IS_LINUX
#endif
```

By defining `IS_WINDOWS` and `IS_LINUX` in a single, shared header, platform-specific logic in the rest of the code can be expressed clearly and concisely using `#ifdef` or `#if defined(...)` blocks.

2.5 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.

2.5.1 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.

2.5.2 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.

2.5.3 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:

```
# 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()
```

2.5.4 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:

```
# 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

```
// 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.

2.5.5 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.

```
# 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.

```cmake
# 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.

```cmake
# 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:

```cmake
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()
```

2.5.6 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.

2.6 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. This value should be chosen from one of the valid return codes defined in the `Constants.hpp` file (see section 2.13).

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.

```
// 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.

2.7 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 2.6). 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), `-d` (debug level) 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()`.
- Debug level is handled internally and can be queried via `getDebugLevel()`.
- 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 2.12 for more details.

Runtime Context

RuntimeContext is a struct designed to encapsulate shared runtime configuration and services used across application components. It holds common data such as a global `logger` instance for consistent logging, a `verboseMode` flag to control verbose output, and a `debugLevel` indicating the debug verbosity. This context struct is intended to be passed by reference or pointer to subsystems and utilities that require uniform access to these application-wide settings during execution.

```
struct RuntimeContext
{
    logger::Logger logger;      // The application logger.
    bool verboseMode{false};    // Stores verboseMode.
    unsigned int debugLevel{0}; // Stores the debuglevel.
};
```

The `MIAApplication` class provides a `getContext()` method which returns a pointer to this context object which can be used by other classes. If another class desires the information from context, it should store a pointer to the context object of the application using something similar to the below:

```
void setContext(const RuntimeContext& ctx)
{ context = &ctx; }

const RuntimeContext* context{nullptr};
```

Protected Members

- `RuntimeContext context`
  The runtime context members used by the application.
- `bool helpRequested`
  Set to `true` if the user requested help via the command-line.
- `std::string executableName`
  Store the executable name for use in the help message or other areas.

Logging Macros

To promote consistency and reduce boilerplate in logging method entry points, the application framework defines reusable macros for standardized logging. These macros improve traceability during debugging, especially when verbose mode is enabled.

- `LOG_METHOD_CALL(...)` logs the name of the calling method when it is entered. It uses the internal logger and respects the verbose mode flag. Optional parameters can be entered which allow the caller to specify a string of parameters or context data to include in the log output.

These macros are designed for use at the beginning of methods to ensure clear trace logging with minimal code overhead. They automatically utilize the `__func__` identifier for method name capture and rely on the internal `logger::Logger` instance.

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.

```cpp
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.

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

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

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

5. Use `getdebugLevel()` to conditionally enable debug output (if the debug flag was used when running the application).

```cpp
if (getdebugLevel() >= 1)
    std::cout << "This is debug output!" << std::endl;
if (getdebugLevel() >= 2)
    std::cout << "This is higher level debug output!" << std::endl;
```

6. 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;
}
```

7. 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);
```

8. Use the `LOG_METHOD_CALL()` macro at the start of methods to automatically log method entry points. These macros use the internal logger and respect the verbose mode flag.

```cpp
void MyClass::compute()
{
    LOG_METHOD_CALL(); // Logs method entry if verbose mode is enabled
    // ...
}

void MyClass::computeWithInput(int x)
{
    LOG_METHOD_CALL("x=" + std::to_string(x));
    // ...
}
```

Benefits

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

2.8 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.

2.8.1 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.

```cpp
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.

2.8.2 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.

2.9 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.

Exception Throwing Macro

To streamline exception throwing and ensure consistent inclusion of contextual metadata, the framework defines a macro `MIA_THROW(code, ...)`. This macro reduces boilerplate when throwing exceptions while ensuring that the error message includes file, line, and function information. This contextual data is critical for debugging and tracing issues during runtime. Instead of manually constructing exception messages with source location details, developers can use this macro for clarity and convenience:

```
if (!configLoaded)
{
    MIA_THROW(error::ErrorCode::Config_File_Not_Set, "Configuration missing");
}
```

The macro assists with the following features:

- Automatically captures and appends `__FILE__`, `__LINE__`, and `__func__`.
- Accepts an optional message string describing the error context.
- Safely wrapped in a `do {...} while(0)` block for predictable macro expansion.

This macro enhances code readability, enforces consistent exception formatting, and aids postmortem analysis or log review. It should be preferred over manual `MIAException` construction unless special behavior is required.

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.

```
if (someErrorCondition)
{
    std::string err = "Details of this error!";
    throw error::MIAException(error::ErrorCode::Catastrophic_Failure, err)
        ;
}
```

2. Throw `MIAException` using the pre-defined macro if file name, line number, and method name are desired in the output. An appropriate error code from `error::ErrorCodes` should be used. If one does not exist, one should be added.

```
if (someErrorCondition)
{
    std::string err = "Details of this error!";
    THROW_MIA_EXCEPTION(error::ErrorCode::Catastrophic_Failure, err);
}
```

3. Use `getErrorDescription()` to log or report known error codes.

4. 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.
    }
}
```

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

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

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

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.

2.10 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 2.5.4 for some related information on adding global paths and compile time through CMake.

2.10.1 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.

2.10.2 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.

2.11 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).

2.11.1 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`.

2.11.2 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.

2.11.3 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.

2.11.4 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.
config::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 2.10.2 for more details).

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

   Important: If you create a private data member within a class for a default configuration file, that variable must be constructed before the `MIAConfig` object.

```
// This is the CORRECT usage.
class WoWFishbot {
public:
    WoWFishbot() : config(defaultConfigFile, constants::ConfigType::
        KEY_VALUE)
private:
    std::string defaultConfigFile{"WoWConfig.MIA"};
    config::MIAConfig config;
}
```

```
// This is the INCORRECT usage.
class WoWFishbot {
public:
    WoWFishbot() : config(defaultConfigFile, constants::ConfigType::
        KEY_VALUE)
private:
    config::MIAConfig config;
    std::string defaultConfigFile{"WoWConfig.MIA"};
}

// Throws this error:
terminate called after throwing an instance of 'std::bad_alloc'
what():  std::bad_alloc
```

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... }
}
```

2.11.5 Quick Example

Consider a configuration file containing an integer value with a key of 'network.timeout'

```
# This is the settings.conf file.
network.timeout=15
```

You can collect that value using the following:

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

2.11.6 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`.

2.12 Logging Framework

2.12.1 Overview

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

2.12.2 Free Logging Functions

Three 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.
- `void logMethodCallToFile(methodName, filename, params = "", verbose = false)`
  Logs a message to the specified file which contains the method name and optional parameters. This method is intended to be called with `__func__` as the methodName from some other method call in order to track and log that method entry. This message is somewhat redundant alongside `logToFile`, though it's formatted specifically to be used with some various application framework macros which pre-format some of the arguments (i.e. methodName). See section 2.7 to see how these macros are (and should be) called. 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.

2.12.3 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.

2.12.4 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.

2.12.5 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.

2.13 Return Codes

To standardize and simplify status reporting across the framework, the `ReturnCode` enumeration defines a consistent set of return values that functions and applications can use to indicate the outcome of their execution. This allows for clearer control flow, easier debugging, and predictable integration behavior.

The `ReturnCode` values are primarily intended to be returned from the `run()` method of application classes conforming to the framework's interface (see section 2.6). This ensures that exit codes are standardized across all applications and compatible with the expectations of the `runApp` launcher and `MIA_MAIN` macro. Developers should select an appropriate `ReturnCode` value to reflect the result of their application logic.

```
namespace constants
{
    enum ReturnCode: int
    {
        SUCCESS = 0,   ///< Indicates that the operation completed successfully.
        FAILURE = 1    ///< Indicates that the operation failed due to a general error.
        // Other codes...
    };
}
```

Use of the `ReturnCode` enum improves type clarity and encourages consistent exit handling across the MIA ecosystem. The main return codes are `SUCCESS` and `FAILURE`. This enumeration can be extended with additional codes to cover specific operational contexts (e.g., `FILE_NOT_FOUND`, `INVALID_INPUT`, etc.) if the need exists.

Threading: BackgroundTask Class

The `BackgroundTask` class defines an abstract base for executing background operations on a separate thread, enabling asynchronous processing within an application framework. Derived classes must implement the `run()` method, which contains the task-specific logic. This method is repeatedly called in the internal `threadLoop()` until a stop is requested.

Key features include:

- **Thread Management:** The class encapsulates a `std::thread` object that runs the background task. The `start()` method launches the thread, and `stop()` signals termination by setting an atomic `stopRequested` flag, then joins the thread to ensure clean shutdown.
- **Thread-Safe Stop Signaling:** Uses `std::atomic<bool>` for the stop flag to allow safe, lock-free communication of stop requests between threads.
- **Condition Flag:** An additional atomic `conditionMet` flag allows the background task to track or toggle a condition without stopping the thread, supporting scenarios like toggling state based on external events.
- **Runtime Context Access:** The task holds a pointer to a shared `RuntimeContext` instance, providing access to global application settings such as logging facilities and verbosity flags. The context is injected via `setContext()` to allow flexible runtime configuration without copying.
- **Safety and Correctness:** Copy and move constructors and assignment operators are deleted to prevent accidental thread object duplication or transfer, which could lead to undefined behavior.
- **Extensibility:** A protected `doWhenStopped()` hook can be overridden to perform cleanup or finalization logic after the task stops.

Developers implementing subclasses should ensure that `run()` periodically checks `stopRequested` to enable responsive shutdown. The class design facilitates robust and reusable background task management suitable for long-running or repeating operations within multi-threaded applications.

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++)

```cpp
config::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

Utilities

This chapter documents reusable utility components that support common development needs across the project. All utility modules are defined in CMake using the naming convention *Name*_UTIL. These modules can be linked into your targets via CMake's `target_link_libraries` command. This allows modular inclusion of specific utility functionalities based on your build requirements. To use a utility, simply add the corresponding utility target to your CMake target, for example:

```
target_link_libraries(MyTarget PRIVATE Types_UTIL Math_UTIL)
```

Below is a list of available utility modules.

- **Types_UTIL** – This contains utilities related to type manipulation and handling.
- **Math_UTIL** – This contains mathematical utilities.
- **System_UTIL** – This contains utilities related to system commands.
- **Files_UTIL** – This contains utilities related to file handling.
- **Audio_UTIL** – This contains utilities related to audio handling.
- **Encryption_UTIL** – This contains various encryption utilities.
- **ML_UTIL** – This contains various Machine-learning utilities.

4.1 Markov Model Generation Module

These features are all part of the `ML_UTIL`.

4.1.1 Overview

In a previous project I was working on for Dungeons & Dragons, I created a model for generating random names and character sequences. That project code can be found here:

`https://github.com/torodean/DnD/blob/main/templates/creator.py`.

The functionality was based on transition patterns between characters of preexisting names. I only later found out that this was referred to as a Markov model. The features related to this which are being added to MIA are a more generalized version of that work, which will follow the naming convention relating them to Markov Models.

The `MarkovModels.hpp` header defines a templated C++ module for constructing and manipulating first-order discrete-time Markov models of arbitrary order. The implementation is generic and supports any comparable data type, including characters, strings, or custom state representations, etc.

4.1.2 First-Order Markov Model

This section contains an explanation of how the markov model functions. Consider some set of sequences S for some arbitrary type, where each element is denoted by a capitalized letter, $S = \{ABC, ABD, BAD\}$. The probability matrix P is constructed by determining all of the elements which follow another, and at

what probability that element has of following the others (The probabilities of elements following some element K is P_K). that is $P(S) = \{K : P_K \forall K \in S\}$.

The set of elements which exist in S are $A, B, C, D, \emptyset$, where $\emptyset$ denotes the absence of an element (or beginning/end of a sequence). Starting with A, we can see that the A element is followed only by B (twice), and D (once) in S. The total number of elements ever following an A is thus three. The probabilities following an element A is thus

$$P_A = \begin{cases} B & : \text{twice} \\ D & : \text{once} \end{cases} \implies P_A = \begin{cases} B & : 66.\overline{6}\% \\ D & : 33.\overline{3}\% \end{cases} = \{B : 0.\overline{6}, D : 0.\overline{3}\} \tag{4.1}$$

Following this same process for the other elements gives

$$P_B = \{A : 0.\overline{3}, C : 0.\overline{3}, D : 0.\overline{3}\} \tag{4.2}$$

$$P_C = P_D = \{\emptyset : 1.0\} \tag{4.3}$$

$$P_\emptyset = \{A : 0.\overline{6}, B : 0.\overline{3}\}. \tag{4.4}$$

The total probability matrix for this set of sequences would then be

$$P(S) = \{A : P_A, B : P_B, C : P_C, D : P_D, \emptyset : P_\emptyset\} = \begin{cases} A : \{B : 0.\overline{6}, D : 0.\overline{3}\} \\ B : \{A : 0.\overline{3}, C : 0.\overline{3}, D : 0.\overline{3}\} \\ C : \{\emptyset : 1.0\} \\ D : \{\emptyset : 1.0\} \\ \emptyset : \{A : 0.\overline{6}, B : 0.\overline{3}\}. \end{cases} \tag{4.5}$$

The $\emptyset$ is a special case in that it represents the first character of a sequence (there is never a character after the last). This format may not look like a matrix at all, but it can be re-written to matrix format. First, note that there are a total of 5 elements (A, B, C, D, $\emptyset$) which will give a 5×5 matrix for all possible combinations. The matrix is configured such that both the rows and columns span from $A \to \emptyset$, covering all the elements of the set. The matrix value of a, b then represents the probability that element a will be proceeded by element b.

$$P(S) = \begin{bmatrix} 0 & 0.\overline{3} & 0 & 0 & 0.\overline{6} \\ 0.\overline{6} & 0 & 0 & 0 & 0.\overline{3} \\ 0 & 0.\overline{3} & 0 & 0 & 0 \\ 0.\overline{3} & 0.\overline{3} & 0 & 0 & 0 \\ 0 & 0 & 1.0 & 1.0 & 0 \end{bmatrix} \tag{4.6}$$

This probability matrix thus represents the probability of an element proceeding another in one of the given sequences. It can be used to generate new sequences which adhere to similar patterns of the input sequences. With larger data sets, more possibilities of sequences typically arise as probable outputs.

One important feature of these models is that under low-entropy (the model is derived from a deterministic source), a uniquely resolvable input set (You can reconstruct exactly one input set) and with enough metadata (initial state, model size, model order, etc), the model can be used to reconstruct the original data.

4.1.3 Functionality

This module provides the following key features:

- **Probability Matrix Construction:** Generates a normalized transition matrix from a collection of input sequences, where transitions between adjacent elements are counted and then normalized into probabilities.
- **State Querying:** Provides utility functions to:
  - Retrieve successor probabilities for a given state.
  - Check whether a given state exists in the matrix.
  - Extract the list of all states in the model.
- **Matrix Utilities:** Includes functions to:

  - Clear the matrix.
  - Print the matrix to an output stream.

- **Sampling Support (Planned):** A placeholder exists for a future function that will enable sampling the next state from a given current state based on its transition distribution.

4.1.4 Design Notes

The transition matrix is represented using nested `std::unordered_map` containers. Specifically, the outer map keys are current states, and the values are maps from successor states to transition probabilities. The code leverages C++17 features, including structured bindings and template aliasing, and is contained within the `markov_models` namespace for modularity and clarity.

4.2 D0sag3 Command (D3C) Integration

These features are all part of the `Encryption_UTIL`.

4.2.1 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.

4.2.2 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.

4.2.3 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.

4.2.4 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.

4.3 Terminal Color Codes

The terminal color features are part of the `System_UTIL` utility. It provides a set of inline functions that insert ANSI escape sequences into output streams to enable colored terminal text. To use it in an application or library, the utility needs included in Cmake, similar to the following example:

```
target_link_libraries(MIATemplate PRIVATE Framework_CORE System_UTIL )
```

Usage

Each color function takes a reference to an `std::ostream` and returns the stream with the appropriate ANSI code inserted. The `TerminalColors.hpp` file needs included to use these features. For example, to print red text:

```
#include "TerminalColors.hpp"

std::cout << red << "This text is red" << reset << std::endl;
```

Available Colors

- `red`
- `bgred` (background color)
- `green`
- `bggreen` (background color)
- `yellow`
- `blue`
- `bgblue` (background color)
- `magenta`
- `cyan`
- `white`
- `def` (default color)
- `bgdef` (default background color)
- `reset` (resets all formatting)

These functions leverage ANSI escape codes and are compatible with Unix-like systems and modern Windows terminals supporting ANSI sequences.

4.4 Virtual Key Strokes Utility

These features are all part of the `System_UTIL`.

The `VirtualKeyStrokes` class provides a cross-platform interface for simulating keyboard and mouse input on Windows and Linux systems. It supports character and string input emulation, basic automation, and platform-specific input features. This utility is designed for automation and testing

scenarios requiring simulated user input. It abstracts platform-specific details, providing a unified API for key and mouse event generation.

Note: The current implementation is very old code, which could definitely use improvements and may require further work for full feature completeness and robustness.

4.4.1 Features

- Simulate key presses for individual characters and strings.
- Support for mouse clicks: left, right, and middle click.
- Specialized key presses on Windows (numbers, letters, special keys).
- Functions to move the mouse cursor and retrieve pixel color at or near the mouse.
- Automation support with sequential key typing and custom delays.
- Cross-platform support via Windows API and X11/`xdo` library on Linux.

4.4.2 Usage Example

To simulate typing a string and a mouse left click:

```
virtual_keys::VirtualKeyStrokes vk;
vk.type("Hello, World!");
vk.leftclick();
```

4.4.3 API Highlights

`void press(const char& character, int holdTime = 0, bool verboseMode = false)` Simulates pressing a single key with optional hold time and verbose output.

`void type(const std::string& word)` Simulates typing a sequence of characters.

`void mouseClick(ClickType clickType, bool verboseMode = false)` Simulates mouse clicks of different types (`LEFT_CLICK`, `RIGHT_CLICK`, `MIDDLE_CLICK`).

`void moveMouseTo(int x, int y)` Moves the mouse pointer to specified screen coordinates.

`void getPixelColorAtMouse()` Retrieves and prints the RGB color of the pixel under the mouse cursor (Windows only).

4.4.4 Platform Details

- **Windows:** Uses the Windows `SendInput` API for keyboard and mouse simulation.
- **Linux:** Utilizes the `xdo` library along with X11 functions to emulate input events.

Chapter 5

Libraries

This chapter provides documentation for domain-specific or general-purpose libraries developed or integrated into the project. All library modules are defined in CMake using the naming convention *Name*_LIB. These modules can be linked into your targets via CMake's `target_link_libraries` command. This allows modular inclusion of specific library functionalities based on your build requirements. To use a library, simply add the corresponding library target to your CMake target, for example:

```
target_link_libraries(MyTarget PRIVATE Database_LIB)
```

Chapter 6

Automated Features and CI/CD Pipeline

This chapter documents the automation and continuous integration mechanisms used in the project. The goal is to ensure consistent, reproducible builds and automated testing during development.

6.1 GitHub Actions: Build and Test Pipeline

The project uses GitHub Actions for continuous integration. The primary workflow is defined in the file `.github/workflows/tests.yml`. This workflow is configured to automatically build the project and run tests when changes are pushed to the `main` branch or when a pull request is opened, but only if the `bin/` directory is modified.

Trigger Conditions

The workflow is triggered on:

- **Push** events to the `main` branch.
- **Pull requests** targeting `main`.

However, it only runs if files inside the `bin/` directory have been modified. This is specified using the `paths` filter in the workflow configuration.

Workflow Steps

The workflow consists of the following steps:

1. **Checkout the repository.** Uses the `actions/checkout@v3` action to retrieve the code.
2. **Install dependencies.** Runs the `scripts/setup.sh` script, which installs required packages including CMake, compilers, and Google Test.
3. **Configure and build.** Executes the `build.sh -v` script to configure and compile the project using CMake.
4. **Run tests.** Runs `ctest` on the `build/` directory and prints test output on failure.

This automation ensures that every relevant code change is verified through build and test steps without manual intervention, improving code reliability and reducing regressions.

Chapter 7

Old MIA Commands and Syntax

7.1 Valid Syntax

MIA was originally designed to be used similar to a terminal or command prompt. One entered a command and then uses the 'Enter' key to perform the commands. Since the migration away from this format and to that of an app format, the terminal is no longer the primary feature of MIA. However, in order to preserve much of the original functionality, an app named `MIAOriginal` was created which implements a similar design to the original MIA using the new app framework. When ran, this app will behave as the old MIA did, with a subset of the original commands. Many of the commands were removed as they have been moved to apps or just don't make much sense with the new structure.

When using the `MIAOriginal`, the following features are available. MIA commands are NOT case sensitive. By default, all commands are changed to lower case before executing through the MIA program. If a command is not typed exactly how it is intended (including spaces and newline characters) it may not execute. Many of these features have not been tested (at the time of writing this, none of the Windows specific features have been tested) since moving to the new app structure and may not work.

7.2 Complete List of Valid Commands for `MIAOriginal`

7.2.1 Static Commands

```
help
```

Displays a valid lists of commands and a brief description to go along with each.

```
add
```

Adds two positive integers of any length. This adds two strings together using a similar algorithm one would when adding large numbers by hand. It is possible to get results by entering non-number entries but will serve no significance due to the way MIA internally converts strings to integers by shifting the ASCII values.

```
button spam
```

Spams a specified button (key press). This function asks for a key input as well as asks for a number of times the user would like the button spammed. Not all keys are programmed in and the time between button spamming is currently fixed (this will be updated at a later time). This function currently only works on Windows OS.

```
button spam -t
```

Does the same as the "button spam" command but also simulates the tab key in between each key press.

```
collatz
```

Produces a collatz sequence based on a specified starting integer. This method uses the login data type which means if a number of the sequence extends the storage of a long, the results will become untrustworthy.

```
crypt -d0s1
```

Encrypts a string using the d0s1 algorithm. This is explained more in chapter 4.2.

```
crypt -d0s2
```

Encrypts a string using the d0s2 algorithm. This is explained more in chapter 4.2.

```
decrypt -d0s1
```

De-crypts a string using the d0s1 algorithm. This is explained more in chapter 4.2.

```
decrypt -d0s2
```

De-crypts a string using the d0s2 algorithm. This is explained more in chapter 4.2.

```
digitsum
```

Returns the sum of the digits within an integer of any size. Similar to the add command, this converts a string to an array of integers using ASCII shifting and then sums the values together. Due to this, you can also find values for entering non-numerical strings.

```
error info
```

Returns information regarding an error code.

```
error info -a
```

Returns information regarding all error codes.

```
eyedropper
```

Returns the RGB value of the pixel located at the cursor.

```
exit
```

Quits MIA.

```
factors
```

Returns the number of factors within an integer. The integer must be smaller than C++'s internal storage for the long data type.

```
find mouse
```

This function will locate the position of the usere mouse pointer after 5 seconds and print the coordinates it is located at.

`lattice`

Returns total lattice paths to the bottom right corner of an n x m grid. This function is only valid for situations in which the answer will not exceed the internal storage of a long data type.

`multiply`

Multiplies two integers of any length. Similar to add, this multiplies two strings together using a similar algorithm one would when multiplying large numbers by hand. It is possible to get results by entering non-number entries but will serve no significance due to the way MIA internally converts strings to integers by shifting the ASCII values.

`palindrome`

Determines if a positive integer is palindrome. The integer must be smaller than C++'s internal storage for the long data type.

`prime`

Determines if a positive integer is prime or not. The integer must be smaller than C++'s internal storage for the long data type.

`prime -help`

Displays help defaults for prime functions.

`prime -f`

Determines all of the prime factors of a positive integer. The integer must be smaller than C++'s internal storage for the long data type.

`prime -n`

Calculates the n'th prime number up to a maximum number of 2147483647.

`prime -n -p`

Creates a file of all prime numbers up to a maximum number of 2147483647.

`prime -n -c`

Clears the file created by 'prime -n -p'.

`quadratic form`

Calculates the solution to an equation of the form $ax^2 + bx + c = 0$. This function accounts for imaginary answers.

`subtract`

Finds the difference between two integers of any length. Similar to add, this subtracts two strings together using a similar algorithm one would when subtracting large numbers by hand. It is possible to get results by entering non-number entries but will serve no significance due to the way MIA internally converts strings to integers by shifting the ASCII values.

`randomFromFile`

Prints a number of random lines from a text file. This will read in each line from a text file and print a user specified number of the lines by choosing them randomly.

`triangle`

Determines if a number is a triangle number or not. The integer must be smaller than C++'s internal storage for the long data type.

7.2.2 Fluid (Volatile) Commands for `MIAOriginal`

The fluid commands are commands that do not have a fixed input. They are generally formatted commands that can be entered with user specific input.

```
XXdYY    //Where XX and YY are integers.
1d20     //Example of rolling a 20 sided dice.
3d6      //Example of rolling three 6 sided dice.
```

Rolls a dice. The format of this command is XXdYY, where XX and YY are both integers. The value of XX determines the number of dice to roll and the value of YY determines the value of each dice.

Chapter 8

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.

8.1 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
# LISTEN=1
# PRESS=SPACE
#----------------------------------------
# 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.
   -d, --debug             Enable debug output at a specified level.
   -h, --help              Show this help message
   -l, --logfile           Set a custom logfile.

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.
   -L, --loop              Loop over the activated sequence
      ↪ indefinitely.
   -P, --list              Print a list of all valid sequences when ran
      ↪ .
```

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.

8.2 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
LISTEN=1
TYPE=abcdefg
PRESS=SPACE
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. `LISTEN=1` This will open a background thread which listens for the key corresponding to a key code of 10 (the ’1’ key). When detected, the sequence will toggle on and off. When toggled, the sequence will stop or restart.
4. `TYPE=abcdefg` This command will type the text entered. In this case “abcdefg” will be typed.
5. `PRESS=SPACE` This command will press the space key.
6. `SLEEP=500` This defines the time to sleep when the hover keyword is used.
7. `MOVEMOUSE=145,887` This is a command signaling to move the mouse to the coordinates (145,887).
8. `CLICK=RIGHTCLICK` This is a command signaling to perform a right click.
9. `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.
LISTEN	char	The key used for stopping and restarting the sequence.
TYPE	String	Any string of characters to be typed as keystrokes.
PRESS	Enum (SpecialButton)	See section 8.2.1
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.

8.2.1 SpecialButton Enumeration

The SpecialButton enumeration defines a set of special, non-alphanumeric button types. These typically correspond to keys that cannot be represented as single characters and are used in scenarios where such special key input needs to be identified or handled explicitly.

- **UNKNOWN**
  Represents an unrecognized or unspecified button type.
- **ENTER**
  Represents the Enter (Return) key, commonly used to submit input or commands.
- **TAB**
  Represents the Tab key, used to move focus or navigate between input fields.
- **SPACE**
  Represents the Spacebar key, used to insert a space character or activate UI elements.
- **NUM_LOCK**
  Represents the Num Lock key, which toggles the numeric keypad input mode.

Additional special button types may be added to this enumeration as needed to support more complex input scenarios.

8.3 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.

Chapter 9

Workout Generation

9.1 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 9.4 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.

9.2 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.

9.3 Input File and Defining Workouts

9.3.1 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.

9.3.2 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 9.4 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.

9.3.3 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
```

9.4 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.

9.4.1 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

$$S_{max}(s_{min}, s_{max}, d) \equiv S_{max} = \frac{s_{max} - s_{min}}{10^{4/3}} d^{2/3} + s_{min} \tag{9.1}$$

$$S_{min}(s_{min}, s_{max}, d) \equiv S_{min} = \frac{s_{max} - 1.9 \times s_{min}}{1.9 \times 10^{4/3}} d^{2/3} + s_{min}. \tag{9.2}$$

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

$$S_{ave}(s_{min}, s_{max}, d) \equiv S_{ave} = \frac{S_{max} + S_{min}}{2} \tag{9.3}$$

$$= \frac{(2.9s_{max} - 3.8s_{min})}{3.8 \times 10^{4/3}} d^{2/3} + s_{min} \tag{9.4}$$

$$S(s_{min}, s_{max}, d) = rand[S_{min}(s_{min}, s_{max}, d), S_{max}(s_{min}, s_{max}, d)]. \tag{9.5}$$

These are shown in Figure 9.1. The algorithm used to determine the sets per workout is identical to that of determining the number of exercises per set.

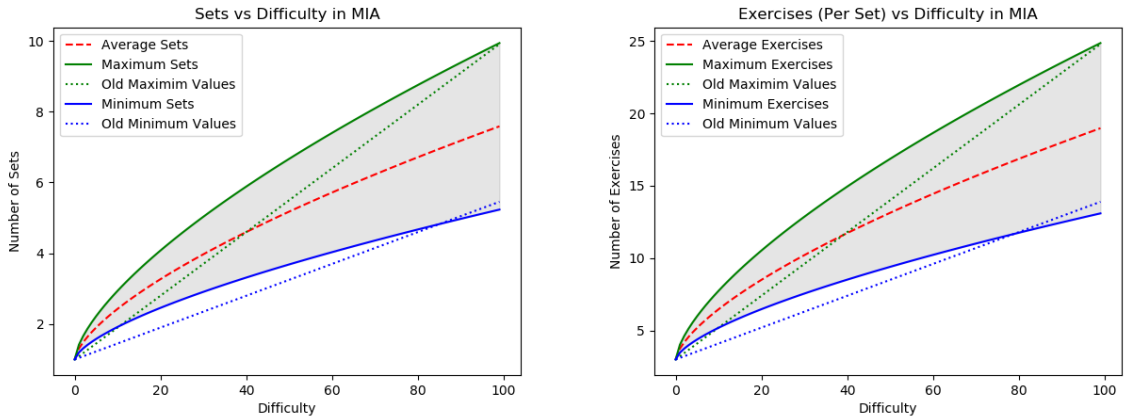

Figure 9.1: Number of sets per workout (left) and number of exercises pet set (right) based on the user input difficulty. The small dotted lines represent the original algorithm which was a simple linear increase in difficulty for each parameter. For the above two figures, values of $s_{min} = 1.0$, $s_{max} = 10.0$, $e_{min} = 3.0$ and $e_{max} = 25.0$ were used. The possible values for S and E are shown via the gray shaded areas.

9.4.2 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

$$E_{max}(e_{min}, e_{max}, d) \equiv E_{max} = \frac{e_{max} - e_{min}}{10^{4/3}} d^{2/3} + e_{min} \tag{9.6}$$

$$E_{min}(e_{min}, e_{max}, d) \equiv E_{min} = \frac{e_{max} - 1.9 \times e_{min}}{1.9 \times 10^{4/3}} d^{2/3} + e_{min}. \tag{9.7}$$

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

$$E_{ave}(e_{min}, e_{max}, d) \equiv E_{ave} = \frac{E_{max} + E_{min}}{2} \tag{9.8}$$

$$= \frac{(2.9e_{max} - 3.8e_{min})}{3.8 \times 10^{4/3}} d^{2/3} + e_{min} \tag{9.9}$$

$$E(e_{min}, e_{max}, d) = rand[E_{min}(e_{min}, e_{max}, d), E_{max}(e_{min}, e_{max}, d)]. \tag{9.10}$$

These are shown in Figure 9.1. 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

$$E_{i,max}(e_{min}, e_{max}, d) \equiv E_{max} = \frac{e_{max} - e_{min}}{10^{4/3}} d^{2/3} + e_{min} \tag{9.11}$$

$$E_{i,min}(e_{min}, e_{max}, d) \equiv E_{min} = \frac{e_{max} - 1.9 \times e_{min}}{1.9 \times 10^{4/3}} d^{2/3} + e_{min} \tag{9.12}$$

$$E_{i,ave}(e_{min}, e_{max}, d) \equiv E_{ave} = \frac{E_{max} + E_{min}}{2} \tag{9.13}$$

$$= \frac{(2.9e_{max} - 3.8e_{min})}{3.8 \times 10^{4/3}} d^{2/3} + e_{min} \tag{9.14}$$

$$E_i(e_{min}, e_{max}, d) = rand[E_{i,min}(e_{min}, e_{max}, d), E_{i,max}(e_{min}, e_{max}, d)]. \tag{9.15}$$

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

$$E_W = \frac{1}{S} \sum_{i=1}^{S} E_i. \tag{9.16}$$

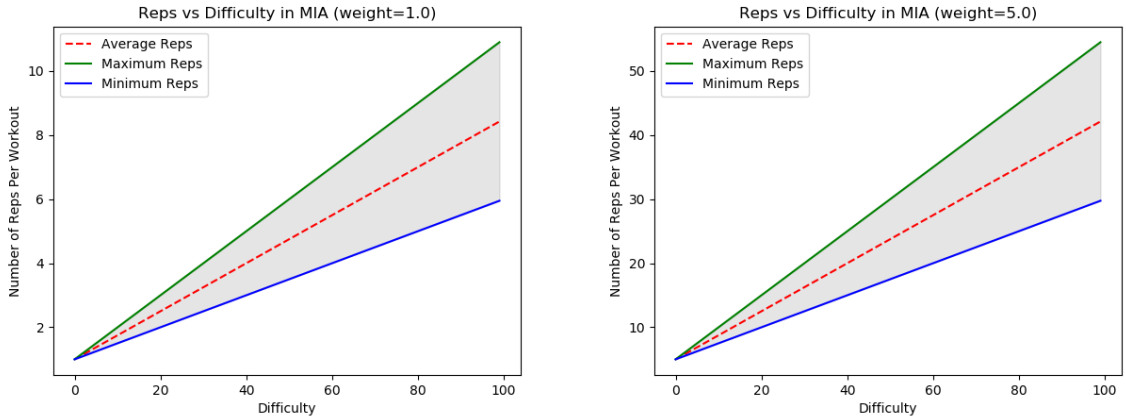

Figure 9.2: Number of reps per exercise. On the right, we have $R(0.1, d, 1)$ and on the left, $R(0.1, d, 5)$. The possible values for R are shown via the gray shaded area.

9.4.3 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

$$R_{max}(t, d, w) = tdw + w \tag{9.17}$$

$$R_{min}(t, d, w) = \frac{tdw}{2} + w \tag{9.18}$$

$$R_{ave}(t, d, w) = \frac{R_{max} + R_{min}}{2} = \frac{3tdw}{4} + w \tag{9.19}$$

$$R(t, d, w) = rand[Rmin(t, d, w), R_{max}(t, d, w)]. \tag{9.20}$$

These are shown in Figure 9.2. 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

$$R_{j,max}(t, d, w_j) = tdw_j + w_j \tag{9.21}$$

$$R_{j,min}(t, d, w_j) = \frac{tdw_j}{2} + w_j \tag{9.22}$$

$$R_{j,ave}(t, d, w_j) = \frac{R_{max} + R_{min}}{2} = \frac{3tdw_j}{4} + w_j \tag{9.23}$$

$$R_j(t, d, w_j) = rand[R_{j,min}(t, d, w_j), R_{j,max}(t, d, w_j)]. \tag{9.24}$$

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

$$R_{i,ave} = \frac{1}{E_i} \sum_{j=1}^{E_i} \frac{R_j}{w_j}. \tag{9.25}$$

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

$$R_W = \frac{1}{S} \sum_{i=1}^{S} \frac{1}{E_i} \sum_{j=1}^{E_i} \frac{R_j}{w_j} \tag{9.26}$$

9.5 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

$$D \equiv SE_W R_W = S \left(\frac{1}{S} \sum_{i=1}^{S} E_i \right) \frac{1}{S} \sum_{i=1}^{S} \frac{1}{E_i} \sum_{j=1}^{E_i} \frac{R_j}{w_j}. \tag{9.27}$$

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

$$D_{max} = S_{max} \left(\frac{1}{S_{max}} \sum_{i=1}^{S_{max}} E_{i,max} \right) \frac{1}{S_{max}} \sum_{i=1}^{S_{max}} \frac{1}{E_{i,max}} \sum_{j=1}^{E_{i,max}} \frac{R_{j,max}}{w_j}. \tag{9.28}$$

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

$$D_{max} = S_{max} \left(\frac{1}{S_{max}} S_{max} E_{max} \right) \frac{1}{S_{max}} S_{max} \frac{1}{E_{max}} E_{max} \frac{R_{max}}{w} \tag{9.29}$$

$$= S_{max} E_{max} \frac{R_{j,max}}{w_j}. \tag{9.30}$$

Similarly, the minimum value is

$$D_{min} = S_{min} E_{min} \frac{R_{j,min}}{w_j}, \tag{9.31}$$

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 9.3.

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.

9.6 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.

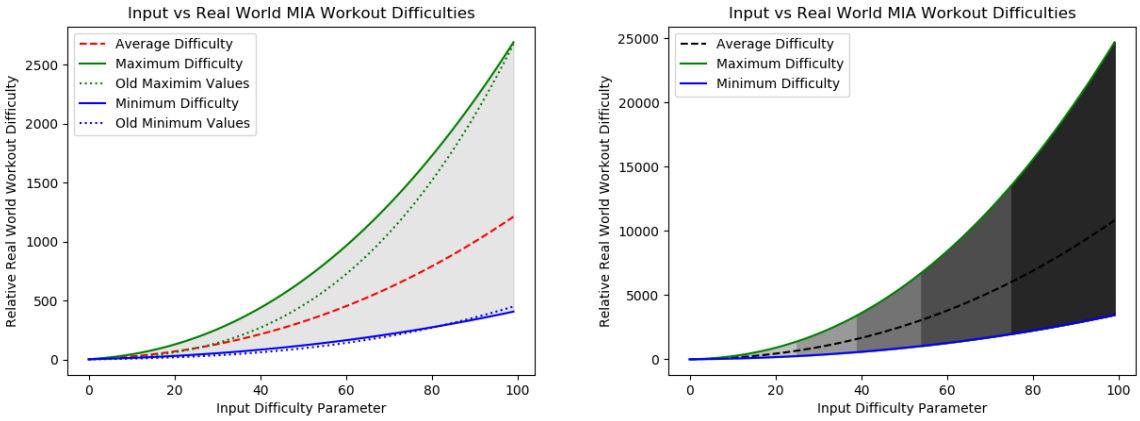

Figure 9.3: The real world difficulties D output by the MIA workout generation program versus the input difficulty parameter (LEFT). The possible values for D are shown via the gray shaded area. The old min/max values represent D based on the old S and E values (see figure 9.1). On the right is the same plot only depicting the difficulty ranges. The ranges run from very easy (light gray), to very hard (dark gray).

Chapter 10

World of Warcraft (WoW) Features in MIA

10.1 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.

10.1.1 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 10.1). Be sure to click apply before closing out of the settings.

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

Figure 10.1: 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 10.2). 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 10.2: 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 10.3: 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 10.3. The second two parameters, `WoWFishBotEndX, WoWFishBotEndY` define the coordinates of the point B in figure 10.3. 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 10.3, 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.

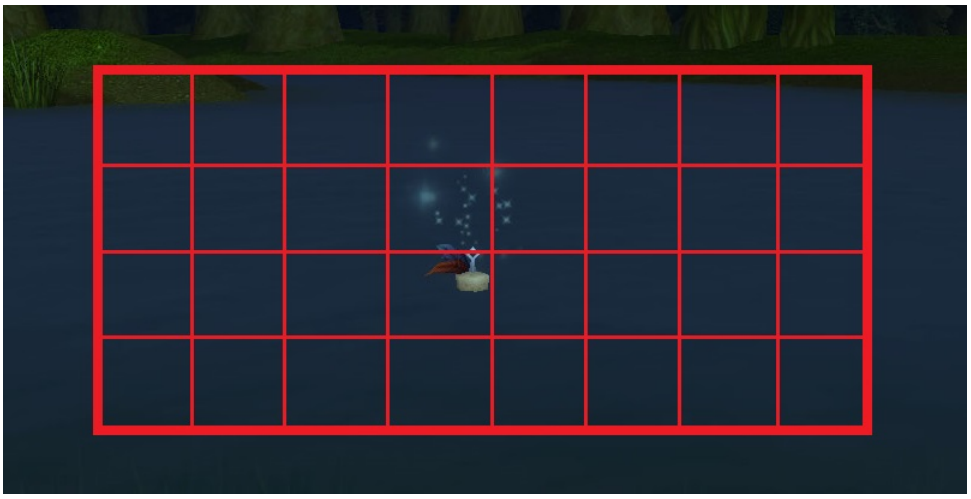

Figure 10.4: The red square similar to that shown in figure 10.3 partitioned into squares of size defined by the `WoWFishBotIncrement` value.

```
fishbot

...CAUTION! This fishbot was made for educational purposes.
...WARNING! Use the fishbot at your own risk!
...DANGER! Using this fishbot may have negative consequences.
...ALERT! This fishbot may get you banned.
...                                                          ...
...In order for the fishbot to work, please enter in game settings and DISABLE
... hardware cursor.
...To use default values (3 for cast and 8 for lure) leave the following options
... blank.
...Press CTRL-C to stop the fishbot early once started.
...Press ENTER to continue.

...                                                          ...
...Please enter which button you have set to cast:

...If you are not using a lure please enter 'NONE'
...Please enter which button you have set to apply a lure:

...Loading Fishbot Modules.
```

Figure 10.5: 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 10.5)

10.2 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 10.6). 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 10.7). MIA is designed to automate this entire process.

10.2.1 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 10.6. 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 10.6: 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."

10.2.2 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.

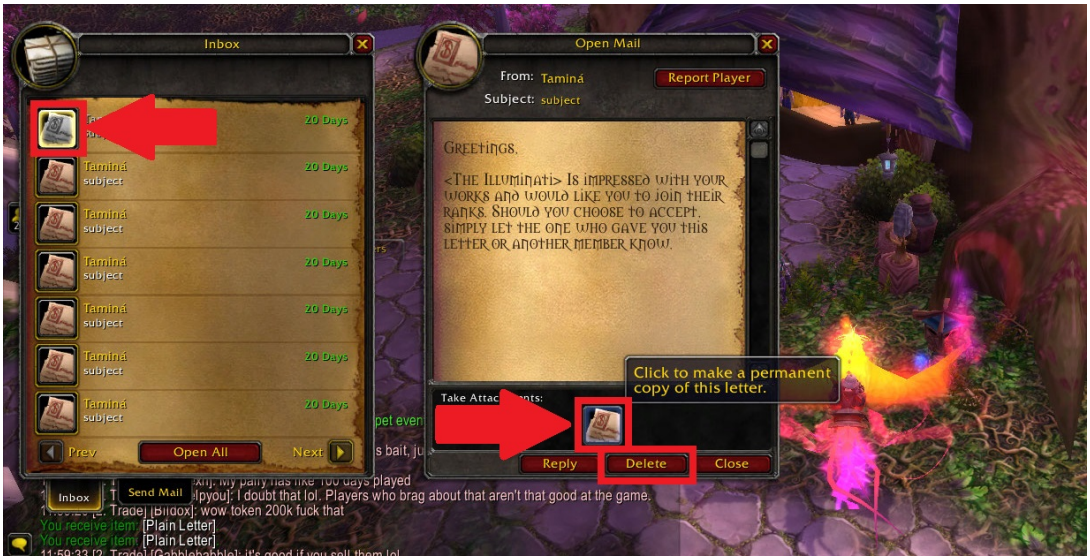

Figure 10.7: A screen shot of the WoW in game incoming mailbox menu. The mail management within MIA utilizes the fields indicated by red squares in this menu. The left arrow indicates the `WoWMailboxFirstLetterLocation` position. The right arrow indicates the `WoWMailboxLootLetterLocation` position. 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 10.7). The second, `WoWMailboxLootLetterLocation` corresponds to the location of the letter to loot from the user inbox (see figure 10.7). The last, `WoWMailboxDeleteLetterLocation` represents the locations of the delete button on a letter in the WoW inbox (see figure 10.7). 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 11

Coding Standards

IMPORTANT: These coding standards are still being developed and not finished or finalized. They can safely be followed, but some parts may change, or additions will be made in the future.

11.1 Overview

As of MIA version 2[1], a set of coding standards has been adopted by this project. To maintain consistency, readability, and maintainability across the project, a unified coding standard for all C++, CMake, and Bash code is used. This section outlines the conventions and best practices to be followed by all current and future contributors. Adhering to these guidelines helps reduce bugs, ease code reviews, and improve collaboration if MIA ever became a multi-developer environment.

Since this project was (and still is at the time of writing this) solely for personal edification, learning, and experimentation, its primary purpose has been to explore new tools, techniques, and patterns across C++, CMake, and Bash. However, it also serves as a practical demonstration of my coding style and development practices - much like a portfolio, though firmly within the realm of hobby code. As such, while the codebase may not meet production-level rigor in every detail, this coding standards documentation exists to highlight the conventions I choose to follow, the rationale behind them, and how they reflect what I believe to be a synthesis of the best practices across the diverse landscape of existing standards. This includes adapting principles from well-known style guides while also incorporating personal preferences that I've found to improve clarity, maintainability, and efficiency through hands-on experience.

11.2 C++ Coding Standards

11.2.1 Formatting and Style

- **Indentation**: Use `spaces` with a width of `4` spaces.
- **Line length**: Limit lines to `120` to `140` characters. Traditionally, less characters are used. However, with newer computers and displays, this rarely makes sense. With a `120` to `140` width, most modern-sized computer displays can display 2-3 columns/files of code at a time wth an additional file-browser window. The range in values comes from a standput of developer discretion. Often times, a hard cap at a set line length can produce some fairly unappealing code and less readable, where the, range allows the developer to use discretion based on the code itself.
- **Brace placement**: `Allman`. Although the original MIA project was in K&R format, I've chosen to move to `Allman` as of version 2. The benefits of Allman include the following:
  - Aligned Blocks: With Allman, readability is often enhanced because you do not have to visually search the end of variable-length lines for brackets or rely on tabs. Allman can help reduce cognitive load when scanning code. All brackets are at the start of the line and aligned with the ending bracket.

[1] Note: Although the coding standards are adopted in version 2, not all of the code yet conforms to this standard. Much of the old code which was migrated over still needs improvements and fixes in order to fit these standards.

  - Clearly Separated Blocks: This style makes each block of code more clearly separated.
  - Commenting: The initial bracket of a code block can serve as a perfect spot for a brief comment related to the block itself.

- **Naming conventions**:
  - Classes and structs: `PascalCase`
  - Enums and typedefs: `PascalCase`
  - Functions and methods: `camelCase`
  - Variables: `camelCase`
  - Constants: `ALL_CAPS`

11.2.2 Code Organization and Practices

- Header files should include include guards or `#pragma once`.
- Prefer smart pointers over raw pointers where ownership is involved. Smart pointers such as `std::unique_ptr` and `std::shared_ptr` manage memory automatically and help prevent common issues like memory leaks, dangling pointers, and double deletions. They make ownership semantics explicit and integrate with RAII (Resource Acquisition Is Initialization), ensuring that resources are properly released when no longer needed. Raw pointers can still be used for non-owning references, but ownership should be clearly represented through smart pointers to improve code safety and maintainability.
- Avoid using `using namespace std;` in headers. Placing `using namespace` directives in headers pollutes the global namespace for any file that includes that header, potentially causing name collisions and ambiguous references. This can lead to subtle and hard-to-debug errors, especially in large codebases or when integrating with third-party libraries. Instead, fully qualify names (i.e., `std::vector`) in headers to keep dependencies explicit and avoid unintended side effects.
- Separate declaration and implementation: use headers (`.hpp`) for interfaces and source files (`.cpp`) for implementation.
- Prefer forward declarations over includes in headers when only a pointer or reference to a type is needed. Only include full headers when access to the complete type is required.
- Where applicable, default fallback values (i.e., 0, "", false) may be returned in lieu of exceptions for non-critical failures, provided this behavior is explicitly documented. This includes all get*() functions for configuration access.
- Exceptions should be used only for unrecoverable errors or contract violations, and custom exceptions (i.e., MIAException) should encapsulate error context where possible.
- Use Resource acquisition is initialization (RAII) for resource management.
- Use const correctness to express immutability.
- Document all public interfaces with Doxygen-style comments.

11.2.3 Virtual Classes, Abstractions, and PIMPL

Object-oriented abstractions such as virtual classes and the PIMPL (Pointer to Implementation) idiom are powerful tools but must be used judiciously to avoid unnecessary complexity and indirection. Code should prioritize readability and simplicity over overengineering. Clear, straightforward implementations are easier to maintain, debug, and extend than overly abstract or generalized solutions. Avoid premature optimization or architectural complexity unless justified by concrete requirements. The goal is to make the intent and behavior of the code immediately understandable to other developers. When in doubt, favor code that communicates clearly over code that attempts to be overly clever or flexible without a proven need.

Virtual Classes and Abstract Interfaces

Abstract classes (i.e., classes with pure virtual functions) should be used to define clean, minimal interfaces that enable polymorphism and decouple high-level logic from specific implementations. These interfaces are particularly appropriate in the following cases:

- Multiple interchangeable implementations are required.
- Code must be testable via mocks or stubs.
- A plugin or extensibility mechanism is being designed.

Avoid over-engineering by introducing interfaces prematurely. When there is no clear need for polymorphism, prefer concrete classes for simplicity and clarity. Overall, free-functions are preferred where applicable.

PIMPL Idiom

The PIMPL idiom may be used to encapsulate implementation details and reduce compilation dependencies, particularly for public interfaces that are expected to remain stable while their internals may change. Its use must be justified by the need to isolate dependencies, improve build performance, or enforce ABI stability. The PIMPL (Pointer to Implementation) idiom is used to:

- Hide private data members and internal implementation details.
- Reduce build dependencies and improve compilation times.
- Achieve binary interface stability across library versions.

PIMPL should only be used when these benefits are specifically needed. It introduces runtime indirection and heap allocation, which may be unnecessary in simpler or performance-sensitive components.

```
/**
 * @brief A class demonstrating the PIMPL idiom for encapsulation.
 */
class MyClass
{
public:
    /**
     * @brief Constructs a MyClass instance.
     */
    MyClass();

    /**
     * @brief Destroys the MyClass instance, freeing resources.
     */
    ~MyClass();
private:
    class Data; ///< Forward declaration of implementation details.
    std::unique_ptr<Data> data; ///< Pointer to implementation.
```

Usage Guidelines

- Use virtual functions only where dynamic behavior is explicitly required.
- Keep interfaces small, focused, and consistent.
- Use PIMPL for library boundaries or complex internals where ABI stability or encapsulation justifies the added indirection.
- Avoid using PIMPL as a default or substitute for proper modular design.

By adhering to these principles, the code remains both flexible and maintainable without incurring the overhead of unnecessary abstraction.

11.2.4 Attributes, Specifiers, and Member Annotation Standards

C++ Attribute Specifiers

C++ standard attributes such as `[[deprecated]]`, `[[maybe_unused]]`, and others should be used appropriately to enhance clarity, compiler diagnostics, and maintainability.

- `[[deprecated]]`
  Use to mark deprecated interfaces explicitly. Always include an explanatory message:

```
[[deprecated("Use newMethod() instead")]]
void oldMethod();
```

- `[[maybe_unused]]`
  Apply to functions or variables that may be unused in some contexts (i.e., for debugging or optional features):

```
[[maybe_unused]]
void dumpConfigMap(std::ostream& os = std::cout) const;
```

- `[[nodiscard]]`
  Use for return values where the result must be used by the caller (i.e., for error codes or ownership types).

Use attribute specifiers only when their intended behavior is part of the API contract or serves a specific, useful purpose. Avoid excessive or speculative use.

`override` and `final` Usage

All overridden virtual methods must be explicitly marked with `override`. Do **not** use the `virtual` keyword on the overriding declaration.

- Correct:

```
void onStart() override;
```

- Incorrect:

```
virtual void onStart() override;
```

Use `final` on classes or methods where further subclassing or overriding is explicitly disallowed:

```
class Logger final { ... };
void onShutdown() final;
```

11.2.5 Comments and Documentation

All code elements including classes, namespaces, methods, typedefs, variables, and files must have Doxygen comments. These comments should primarily be placed in header files to serve as the main documentation source. For methods not declared in headers but implemented in `.cpp` files, corresponding Doxygen comments should be added in the implementation files. In-code comments must be sufficiently informative to allow any developer to understand the purpose, usage, and modification requirements of the code without needing to consult external documentation. This includes describing the intent of classes and methods, explaining parameters and return values, and clarifying design decisions where applicable.

In `.cpp` implementation files, comments should be added at the developer's discretion but are ideally included:

- Before blocks of code that perform distinct tasks,
- When specific literal values or "magic numbers" are used for particular reasons,
- Where behavior might not be immediately clear from the code itself,
- To highlight important side-effects or assumptions.

For ease of navigation and maintenance, closing braces of namespaces, classes, and long code blocks should be followed by simple trailing comments indicating their scope, for example:

```
} // namespace myNamespace
```

This convention helps match braces quickly without excessive scrolling.

Additional guidelines

- Avoid redundant comments that restate obvious code.
- Use complete sentences, present tense, punctuation, and consistent phrasing for clarity.
- Use consistent terminology and formatting in all comments.
- Prefer clarity and brevity over verbosity.
- Update comments promptly alongside code changes to keep them accurate.
- All functions that may throw exceptions must document them using @throws.

Example Doxygen comments

For multi-line comments, use the `/** .. */` comment block.

```
/**
 * @brief Represents a configurable widget.
 * This class manages widget configuration and state.
 * @param config[Config] - Configuration settings for the widget.
 * @return Widget instance initialized with given config.
 */
class Widget
{
public:
    /**
     * @brief Initializes the widget with given settings.
     * @param settings[Config] - The configuration to apply.
     */
    void initialize(const Config& settings);
};
```

For single-line comments that document member variables, use `///` directly above the definition. These comments should be clear, concise, and written in full sentences when appropriate.

```
/// The name of the configuration file for this object.
std::string configFileName;
```

Guidelines:

- Use `///` rather than `//` for documentation intended to be parsed by tools like Doxygen.
- Comments must describe the purpose or semantics of the variable, not its type or structure (which should be clear from the code).
- Maintain grammatical consistency and use proper punctuation.

Each enumeration should be preceded by a Doxygen-style comment block explaining its purpose. Individual enum values must use `///<` inline comments to document their meaning. This ensures that documentation tools capture the purpose of each constant clearly, and that readers can understand the intent without leaving the context of the declaration.

```
/**
 * These are the various actions which the sequencer supports.
 */
enum SequenceActionType
{
    UNKNOWN,      ///< Unknown action - do nothing.
    TYPE,         ///< This will type a sequence of characters.
    SLEEP,        ///< This will wait/pause some time.
    DELAY,        ///< This is the time to wait between each action.
    MOVEMOUSE,    ///< This will move the mouse to a specific coordinate.
    CLICK         ///< This will perform a click with the mouse.
};
```

Guideline: Always use `///<` on the same line as the enum value, not on a new line above. Comments should be concise, use present tense, and describe the runtime behavior.

Line Comments in Implementation Files

Use // for brief, inline, or explanatory comments within function implementations. These should clarify logic, document assumptions, or highlight non-obvious decisions.

```
// Apply default configuration values if not explicitly set
if (!settings.hasTimeout())
{
    settings.setTimeout(DEFAULT_TIMEOUT); // DEFAULT_TIMEOUT defined in config.
        h
}
```

```
// Bad: Sets timeout to default
settings.setTimeout(DEFAULT_TIMEOUT);

// Good: Use default timeout if none specified in config
settings.setTimeout(DEFAULT_TIMEOUT);
```

Guidelines:

- Place standalone comments on their own line above the code they describe.
- Use inline comments sparingly and only when the code is not self-explanatory.
- Do not restate what the code does—explain why it does it, if not obvious.
- Maintain clarity and brevity; comments should add value, not clutter.

11.2.6 README Files and Higher-Level Documentation

Each code directory must contain a `README` file that provides a concise overview of the directory's purpose. The README should include:

- A brief description of the contents of the directory (key files, subfolders).
- The role and relationship of these files or components within the larger project.
- Any special instructions or notes relevant to developers navigating or modifying the directory.

This ensures quick orientation for developers and aids maintainability across the codebase.

At the project level, comprehensive documentation is maintained in the main MIA manual. This manual offers a high-level overview of:

- The complete set of features and functionality.
- Descriptions of libraries and core components.
- Available tools and utilities.
- Architectural design decisions and workflows.

The MIA manual serves as the primary reference for understanding the system holistically and should be kept up-to-date as the project evolves.

Together, directory-level README files and the MIA manual provide layered documentation: granular, file-level guidance paired with broad, conceptual understanding, facilitating both detailed development work and strategic planning.

Example README excerpt

```
# apps Directory Overview

This directory contains the standalone apps for the MIA project. Below is a
    brief description of each subfolder:

- **template/**
  Contains a fully functional app featuring most of the basic application
      framework techniques used in developing MIA apps. This is used to both
      test new features and provide a common template for storing correct usage
       of the various features.
```

11.3 CMake Standards

11.3.1 Structure and Conventions

A clean CMake structure improves build reproducibility and portability. Modern practices enable more maintainable and scalable build definitions. Clear separation of targets and dependencies reduces cross-target pollution.

- Use lowercase CMake commands (i.e., `add_executable`, `target_link_libraries`).
- Group CMake files logically and modularly using subdirectories and `add_subdirectory()`.
- Use modern CMake features (i.e., target_include_directories, target_compile_features).
- External dependencies must be managed using `FetchContent` or `find_package`.
- Add comments to any blocks of cmake code which function together to perform a particular task. For example:

```
# 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 )
```

11.4 Bash Scripting Standards

11.4.1 Style and Safety

- Use `#!/bin/bash` (or `#!/usr/bin/env bash`) at the top of scripts.
- Use functions to encapsulate logic (when appropriate) instead of writing top-level scripts.

11.4.2 Documentation and Naming

- Start each script with a brief comment block describing its purpose and usage.
- Use descriptive function and variable names.
- Use comments to clarify non-obvious logic, especially when using subshells or traps.

11.5 Enforcement and Exceptions

- Continuous integration pipelines are still in development for this project which will enforce code quality.
- Code reviews are expected to evaluate adherence to these standards.

If a contributor proposes a deviation from these standards for good reason (i.e., performance, cross-platform needs), it must be documented and justified in the merge request. In all other cases, adherence is expected.