AIRGEAD BANKING STANDARDS DOCUMENT

AIRGEAD BANKING | 2019

TABLE OF CONTENTS

TABLE OF CONTENTS 1
Coding Best Practices to be Followed 2
Safety 2
Style 2
Maintainability 3
Portability 3
References 4

Coding Best Practices to be Followed

Safety

- Returning values from functions by reference can have significant performance savings when the normal use of the returned value is for observation.
- Avoid raw memory access.

```
// Bad Idea
MyClass *myobj = new MyClass;

// ...
delete myobj;


// Good Idea
auto myobj = std::make_unique<MyClass>(constructor_param1, constructor_param2); // C++14
auto myobj = std::unique_ptr<MyClass>(new MyClass(constructor_param1, constructor_param2)); //
C++11
auto mybuffer = std::make_unique<char[]>(length); // C++14
auto mybuffer = std::unique_ptr<char[]>(new char[length]); // C++11

// or for reference counted objects
auto myobj = std::make_shared<MyClass>();

// ...
// myobj is automatically freed for you whenever it is no longer used.
```

- Use exceptions.
- Use C++-style cast instead of C-style cast.

Style

- Common C++ naming conventions:
  - Types start with upper case: MyClass.
  - Functions and variables start with lower case: myMethod.
  - Constants are all upper case: const double PI=3.14159265358979323;.
- The C++ Standard Library and other well-known C++ libraries like Boost use these guidelines:
  - Macro names use upper case with underscores: INT_MAX.
  - Template parameter names use camel case: InputIterator.
  - All other names use snake case: unordered_map.
- Variable names follow camel case notation.
- Distinguish private object data.
  - Name private data with an m_ prefix to distinguish it from public data. The m_ stands for "member" data.
- Distinguish function parameters.
  - Name function parameters with a t_ prefix. The t_ can be thought of as "the," but the meaning is arbitrary. The point is to distinguish function parameters from other variables in scope while using a consistent naming strategy.

```
Example
struct Size
{
  int width;
  int height;
  Size(int t_width, int t_height) : width(t_width), height(t_height) {}
};

// This version might make sense for thread safety or something,
// but more to the point, sometimes we need to hide data, sometimes we don't.
class PrivateSize
{
  public:
    int width() const { return m_width; }
    int height() const { return m_height; }
    PrivateSize(int t_width, int t_height) : m_width(t_width), m_height(t_height) {}

  private:
    int m_width;
    int m_height;
};
```

Maintainability

- Avoid using assert() in your code. This causes the program to fail when something unexpected happens. Consider using try/catch.
- Readable code is easy to understand. Write your code and comment on it as if a non-programmer were reading it.
- When needed, refactor code to improve its use.
- Properly utilize try/catch statements.
    - These keywords prevent your code from running at the client side (due to run-time errors) and can enable the program to "exit gracefully" with proper error messages as well as continue the program execution.

Portability

- Have minimal code in main(). Most of your code should be in classes and main is used simply as a driver to instantiate objects of these classes and call proper functions.
- Use header files. Your class prototype should exist in an H file while the body of that class should exist in a CPP file.

```
// tree.h
class tree
{
    public:
        double height;
        void SetHight(double h);
```

```
        double GetHeight();
};


// tree.cpp
#include “tree.h”
void Tree::SetHight(double h)
{
        height = h;
 }
double Tree::GetHeight()
{
    return height;
}
```

- The #define guard:
    - All header files should have #define guards to prevent multiple inclusion. The format of the symbol name should be <PROJECT>_<PATH>_<FILE>_H_.
    - To guarantee uniqueness, they should be based on the full path in a project's source tree. For example, the file foo/src/bar/baz.h in project foo should have the following guard:

```
#ifndef FOO_BAR_BAZ_H_
#define FOO_BAR_BAZ_H_

...

#endif  // FOO_BAR_BAZ_H_
```

References

Google C++ Style Guide. (n.d.). Retrieved from https://google.github.io/styleguide/cppguide.html

Turner, J. (2019, June 06). GitHub Collaborative Collection of C++ Best Practices. Retrieved from https://github.com/lefticus/cppbestpractices