Style-Guidelines

Purpose

The purpose of this document is to define the stylistic choices for coding in this repository.

This is likely to be an incomplete document that will evolve as the style evolves. Any information that is incorrect is purely by mistake and will be corrected as soon as I am aware.

Banners

I define a **banner** to refer to a structure formed by a multi-line comment.

e.g.

```
//--------------------------------------------------------------------
//
//--------------------------------------------------------------------
```

The above is an empty, hyphenated banner (because hyphens are used for form the lines).

The base standard banner has an 80 character line width. The internal comment may have text on it.

Key portions of code should be **wrapped** between a banner and its **end** banner.

e.g.

```
//--------------------------------------------------------------------
// Some function
//--------------------------------------------------------------------
void some_function()
{
    return;
}
//--------------------------------------------------------------------
// End some function
//--------------------------------------------------------------------
```

This helps with visual clarity on quickly finding then end of a definition (it is easy to lose track of the **{}** characters, including when tabulation is not entirely consistent, see for example the section on namespaces).

When layering wrapped-banners, internal layers can be reduced by 4 to 5 characters, depending on context.

e.g.

```
//--------------------------------------------------------------------
// Some function
//--------------------------------------------------------------------
void some_function()
{
    //----------------------------------------------------------------
    // Another wrapper
    //----------------------------------------------------------------
    int some_int{};
    //----------------------------------------------------------------
    // This is an internal wrapper
    //----------------------------------------------------------------
    double some_double{};
    //----------------------------------------------------------------
    // End This is an internal wrapper
    //----------------------------------------------------------------
    float some_float{};
    //----------------------------------------------------------------
    // End Another wrapper
    //----------------------------------------------------------------
    return;
}
//--------------------------------------------------------------------
// End some function
//--------------------------------------------------------------------
```

As you can see above, the first set of internal wrapper banners have shurnk by 4 characters, do to the initial indentation from being place inside of the function, but they still extend out to character 80.

The second set of internal wrapped banners, only go out to character 75 (truncated by 5 characters) to help maintain clarity.

For extra-emphasis, you may use a heavy banner, otherwise called an equal banner (because it uses equal signs).

e.g.

```
//====================================================================
// Extra emphasis
```

```
//==========================================================================
```

Natural code structures should be surrounded by appropriate wrapped banners. The point is to provide quick, easy, visual guidance on otherwise easy to miss boundaries.

e.g. (Bad)

```
namespace somename
{
Class some_class
{
    int some_int{};
    void some_function()
    {
        return;
    }
};
}
```

The above is a simple example where it isn't too complicated to see the stack of closing curly-brackets at the end and figure thingso out. But when a source-code file is 1000+ lines long, with multiple levels of internal structure that extend longer than a vertical page, it can be very difficult to quickly distinguish where exactly you are in the code. Banners, while increasing the overall length of the file, provide a powerful tool to quickly and visually separate chunks.

e.g. (Good)

```
//--------------------------------------------------------------------------
// My namespace somename
//--------------------------------------------------------------------------
namespace somename
{
//--------------------------------------------------------------------------
// My class some_class
//--------------------------------------------------------------------------
Class some_class
{
    int some_ints{};
    //----------------------------------------------------------------------
    // some_function, takes no parameters, returns nothing
    //----------------------------------------------------------------------
    void some_function() { return; }
    //----------------------------------------------------------------------
    // End some_function, takes no parameters, returns nothing
    //----------------------------------------------------------------------
};
//--------------------------------------------------------------------
// End My class some_class
//--------------------------------------------------------------------
}
//--------------------------------------------------------------------
// End My namespace somename
//--------------------------------------------------------------------
```

This will make files longer, but the visual structure will help when others read your code, or when you return to it at some future date and need to quickly recall what is going on. This has no impact on the size of the binary, so use these banner structures where they seem most useful/natural.

Comments

Comments should be quick an intuitive, and should always use single-line '//' (even if multi-line)

e.g. (Good)

```
// This is a comment that is designed to take multiple lines.
// I'm doing this purely for show, I like to keep lines down to a sing
// If setences are long, break them across multiple lines with an extr
//  so that we can see quickly/easily that this line carries over from
//  line before it.
// Then a new sentence has a unique line with no initial tab.
```

e.g. (Bad)

```
/*
This is a comment that is designed to take multiple lines.
I'm doing this purely for show, I like to keep lines down to a single
If setences are long, break them across multiple lines with an extra t
    so that we can see quickly/easily that this line carries over from
    line before it.
Then a new sentence has a unique line with no initial tab.
*/
```

Why is the `/**/` multi-line comment not used?

I chose to reserve it for debugging purposes (to quickly comment out a section of code as needed). It isn't that hard to type `//` on a new-line and many modern IDEs/editor will automatically add it for you. I also just think that it looks better.

Namespaces

When defining functions/classes/etc inside a namespace, do not give an initial indentation to the definitions.

e.g. (Correct)

```
namespace pssp
{
void some_function();
void some_other_function();
}
```

And not (Incorrect)

```
namespace pssp
{
    void some_function();
    void some_other_function();
}
```

This reduces line-shrinkage. If we want to maintain relatively short lines of code, to help with readibility, it is silly to immediately lose 4 characters of line-space, just because we're inside of a namespace. **Especially** because we will **ALWAYS** be inside a namespace, for the sake of safety.

Interfaces and Implementations

I follow the practice of separating interfaces from implementations, that is having separate header files (.*hpp files) and their corresponding*.cpp files.

I have chosen the naming convention of .hpp for a C++ header file because my editor kept mistaking *.h files for C, even though they can also be for C++. No deeper reason than that.

Inside header files, you should declare structs, classes, and constants. You should also forward delcare you functions (that is, give the return type, name, and information on all the input variables). Do not implement a function inside an interface unless you must (specifically, in pssp_threadpool.hpp I had to define the enqueue function due to the template meta-programming on the Functions and Arguments, there was no way to explicitly forward declare every possible combination of input Function and variable Arguments, so the implementation in that case had to be in the interface file). You'll note that is distinctly different from the template programs in the sac-format library, where I was able to forward declare the explicit versions of the template functions because they were both 1) known a priori and 2) few in number. As much as possible, try to keep implementations out of interfaces.

This is useful because the interface allows us to hide the internal functionality from those who want to use it. This is valuable because it reduces **clutter** in the interface. That allows us to look at the interface file and figure out the higher-level logic/organization more easily. The implementation isn't actually hidden, as it is easily accessible in its appropriate interface file in its full glory. It is a small bit of additional book-keeping, for a huge improvement in overall clarity/readibility at different levels.

This is also useful because it speeds up compilation. When compiling, the compiler sees the interface and the promised functionality and puts a marker there for the linker to handle linking later. I know, linker errors can be ugly, but having both sets is rather useful. If there is a disconnect between the interface and the implementation, your language server will detect this mismatch (clangd for instance), before you ever reach compilation to begin with.

Because at compilation the compiler must scan the entire header file, this reduces the length of each header file and results in faster compilations.

For now, PsSp is small and this isn't a huge concern, but I see no reason not to try to instill good-practices for the future while I'm aware of them.

Interfaces go in the header file and get automatically included in the build. Implementations are automatically included in the build as well.

Always user header-guards for your interfaces and keep all include statements in the interface (the only include statement inside of an implementation should be for its respective interface).

e.g. (interface.hpp)

```
#ifndef PSSP_INTERFACE_HPP_20230610
#define PSSP_INTERFACE_HPP_20230610
#include <iostream>
#include <vector>
#endif
```

e.g. (interface.cpp)

```
#include "interface.hpp"
```

Header-Guards

Header-guard names are in full-caps, prefaced by the namespace (or program), then the actual header followed by the extension (HPP) and then the date the header guard was added/modified in YYYYMMDD format. These subsections of the header-guard are separate by and underscore character (_). Previously the date addition was not included, it has been tacked on to the format for extra internal safety.

e.g. (Good)

```
#ifndef PSSP_INTERFACE_HPP_20230610
#define PSSP_INTERFACE_HPP_20230610
#endif

#ifndef SAC_SOMEINTERFACE_HPP_20230610
#define SAC_SOMEINTERFACE_HPP_20230610
#endif
```

e.g. (Bad)

```
#ifndef pssp
#define pssp
#endif

#ifndef myinterface_h
#define myinterface_h
#endif
```

Why? For organization and safety. By prefacing with the namespace or program-name, you are less likely to run into naming conflicts down the line. Especially as it seems more common for people to use *_H* to end their header-guard instead of *_HPP*. The point is to try to be easy, safe, and intuitive.

Why not use `pragma once`? Because it is not a standard component of the C++ language, while many compilers do support it, it is compiler specific. Header-guards, while a controversial solution, are a commonly accepted one.

Multi-line statements

In general, if a statement is short, it should be a single-line statement and if it is long it should be a multi-line statement.

This comes up very commonly for `if` statements.

e.g. (Good)

```
if (statment) { return; }

if (statment) { return true; } else { return false; }

if (really_long_statement_that_results_in_taking_most_of_the_line)
{ return true; } else {return false; }

if (statement)
{
    do_something();
    do_something_else();
    for (int i{0}; i < 10; ++i) { do_even_more_stuff(i); }
}
else { do_something_else(); }
```

e.g. (Bad)

```
if (statment)
    return;

if (statment)
{
    return true;
}
else
{
    return false;
}

if (really_long_statement_that_results_in_taking_most_of_the_line) {
    return true;
}
else {
    return false;
}

if (statement)
{
    do_something();
    do_something_else();
    for (int i{0}; i < 10; ++i)
    {
        do_even_more_stuff(i);
    }
}
else
{
    do_something_else();
}
```

Why? For a couple of reasons:

1. Excluding the curly-brackets ({}) risks weird bugs later on if the statement gets changed. All you need to do is forget to add the curly-brackets before/during/after the process of making the modification and enjoy the ensueing weirdness.
2. I don't want an if statement that needs to only take 1 line to end up taking 4-lines. It just unnecessarily inflates the length of your code (which we're already inflating with the infinitely more-valuable comments and wrapped banners!).

Include Statements

Inclusion order is as follows:

1. Our interfaces
2. External, non-standard libraries
3. Standard library stuff

Within those, it would be ideal for everything to be in alphabetical order.

The style of inclusions is as follows

```
#include "our_interface.hpp"
#include <external_interface.h>
#include <standard_library_component>
```

Note our internal interfaces are enclosed in double-quotes (and always end in .hpp). External libraries end in whatever extension they choose to use, but we include them in-between angular brackets.

Stuff from the standard library is also included inside angular brackets, but we do not provide the file-extension. This is because the standard library has internal header-guards such that single-components of the library can be include (instead of the entire file). This is unique to the standard library.

It is also ideal to add a note above the include statement as to why it is being included (what does it do, what does it provide?).