# Contributing

We welcome contibutions in the form of pull requests, issues, or code reviews. Here are some quick guidelines to help ensure we can make use of your effort.

### Reporting Issues

If you stumble upon a bug or something seems odd, please file an issue in the [issue tracker][issues].

### Submitting Pull Requests

Fork, then clone the repo:

```

git clone --recursive git@github.com:your-username/cinder/Cinder.git

```

Push to your fork and [submit a pull request][pr]. When describing your pull request, it is best to explain the problem you are solving or feature you are adding, some reasoning behind it, and how the code can be verified. If your modifications affect multiple platforms, please make a note of which platforms and OS versions you have tested on and which ones still need to be tested. If your contribution has already been discussed in an issue or forum post, linking to that may be a sufficient explanation.

Some things to facilitate acceptance of your pull request:

\* Provide a way to test the changes. Examples can be within a sample, the [test](test) folder, a gist, or by explaining how a reviewer can make some edits locally to test.

\* Keep the changes as concise as possible. It is best if a pull request addresses a single issue.

\* Follow our style guide as described below.

\* Write clear and concise commit messages.

\* For sample / test code, try to keep binary data to a minimum. For test assets, the folder [samples/data](samples/data) contains some assets you can possibly make use of, or adding new assets there means they can be used by future samples. App icons should also be referenced from this folder, reducing our overall binary weight and also making it easy to update in the future.

In general, we review pull requests first that best adhere to these suggestions.

### Style Guide

Please make sure your code conforms to the following general guidelines. If something is unclear or undocumented, the rule of thumb is to try to make the source code easy to read, concise, and match the code around it.

#### Naming

\* Use camelCase; classes begin with upper case and variables with lower case.

\* Member variables begin with an 'm' prefix (`mVariable`).

\* Static variables begin with an 's' prefix (`sInstance`).

\* enum types are all upper case with underscore separators

\* ex. `enum Types { TYPE\_1, TYPE\_2, TYPE\_UNKNOWN }`.

#### Spacing and Line Breaks

\* Use tabs for indentation, with a tab stop value of 4.

\* Anything inside a namespace starts with no indentation.

\* Place spaces between braces, braces and arguments, etc.

\* Place a space between `!` and variables to negate them

\* ex. `if( ! mEnabled ) { ...`

\* Brackets for class methods and functions begin on a new line. Brackets for everything else (class declarations, if / for / while loops, etc. ) begin on the same line following a space.

\* The contents of an `if`, `for`, or `while` statement should always be on a new line. This not only makes it easier to read but also prevents some ambiguities that come up in some debugging situations, where you can't tell if you've jumped into the body of the statement or not.

\* `else` statements should be placed on a new line for vertical readability.

#### Types

\* Adhere to const correctness wherever possible. See [this][const\_correctness\_1] and [this][const\_correctness\_2] for explanation.

\* For small, built-in types (ex. int, float, double, size\_t) pass by value

\* For larger types (ex. STL containers, really anything other than built-in types or similar), pass in by const &.

\* If a parameter is expected to be modified by a method, it should be a pointer (not reference) so it is clear at the call site. An exception is when you are using private utility methods.

#### Class inheritance

\* When an object doesn't have clear copy semantics (ex. a system resource or `gl::Texture`, inherit from `ci::Noncopyable` so it is obvious that the object cannot be copied.

\* When overriding a virtual method in a subclass, do not use the `virtual` keyword a second time, instead use `override`, which has the added benefit of the compiler checking that the override took place.

\* Unless it is templated or performance critical code, place method implementations in the .cpp to try to keep the header more concise. The exception to this rule is simple getters

\* ex. `getWidth() const { return mWidth; }`.

#### Comments

\* Place comments in the header files (in the form of doxygen for public or protected interfaces) and leave the implementations as concise as possible.

\* If there is something you think is not obvious in an implementation, then comments can help clarify, but in general it is nice to let the code do the explaining.

\* We generate reference documentation using doxygen, so you are encouraged to give descriptions to as much of the public interface as possible in order to fill out the reference documentation.

\* To make it easier to distinguish the end of a namespace, place a comment after the namespace's closing bracket

\* ex. `} // namespace ci::gl`

#### Example: Class Declaration

```cpp

#pragma once

#inclue "cinder/includes/first.h"

#include <system/includes/next>

namespace cinder {

//! \brief Description of class SomeClass

//!

//! More detailed information about SomeClass (optional)

class SomeClass {

public: // two leading spaces for access specifiers

//! description of what you are constructing and any arguments passed in

SomeClass( int var1 = 0, int var2 = 0 );

//! Place spaces in between braces and arguments.

void someMethod( int argA, const Rectf &bounds );

//! Inline simple methods by keeping the implementation on the same line as the declaration.

void setVar1( int var ) { mVar1 = var; }

//! Try to align the interface so it is easier to read.

void setVar2( int var ) { mVar2 = var; }

//! Getters are usually declared as const.

int getVar1() const { return mVar1; }

private:

void doSomething( int arg );

int mVar1, mVar2;

};

} // namespace cinder

```

#### Example: Class Implementation

```cpp

#include "cinder/SomeClass.h"

using namespace std;

namespace cinder {

SomeClass::SomeClass( int var1, int var2 )

: mVar1( var1 ), mVar2( var2 )

{

if( var1 == var2 ) {

someMethod( var1, Rectf::zero() );

}

else {

...

...

}

}

void SomeClass::someMethod( int argA, const Rectf &bounds )

{

for( int i = 0; i < argA; i++ ) {

doSomething( argA + 2 );

}

}

} // namespace cinder

```

#### Notes on shared\_ptr usage

We use `shared\_ptr`s a lot. Usually they are in the form of a typedef'ed `ObjectRef` to save the user a bit of typing (the `Ref` suffix is short for 'reference counted object'):

```cpp

typedef shared\_ptr<class Object> ObjectRef;

```

Here are a couple rules of thumb regarding how to pass around these `shared\_ptr` objects:

Pass in by const &:

```cpp

void setObject( const ObjectRef &object );

```

However, return by value:

```cpp

ObjectRef getObject() const;

```

This is to avoid some nasty bugs that can be caused when the actual object returned is a subclass of `Object` and the method unwittingly returns a reference to an already destroyed temporary.

#### Error Handling

Avoid silent failures. In general, we use exceptions for cases that an application can recover from, such as when image decoding or glsl compilation fails. For situations that a user cannot possibly recover from, it is better to use assertions and the handy `CI\_ASSERT` macro. However, as is the case many times, there are no rules set in stone for choosing one technique over the other, or choosing a different way to handle an error altogether (such as logging). When in doubt, post your code for peer review.

Exceptions in core cinder code should all inherit from `ci::Exception`.

[pr]: https://github.com/cinder/Cinder/pulls

[issues]: https://github.com/cinder/Cinder/issues

[const\_correctness\_1]:http://www.cprogramming.com/tutorial/const\_correctness.html

[const\_correctness\_2]:http://www.parashift.com/c++-faq/const-correctness.html