How to Contribute to Cuberite

=============================

Thank you for your interest in Cuberite. Contributing to Cuberite is easy, just fork the project on GitHub, make your changes and submit a pull request to get your code merged. That's all there is to it.

Check out [GETTING-STARTED.md][1] for more information about setting up the development environment for Cuberite, finding issues to work on, etc...

If you are new to open source and/or GitHub, or just aren't sure about some details in the contribution process, here's a tutorial to get you started:

[How to Contribute to an Open Source Project on GitHub][2]

Code Conventions

----------------

When contributing, you must follow our code conventions. Otherwise, CI builds will automatically fail and your PR will not be merged until the non-conforming code is fixed. Due to this, we strongly advise you to run `src/CheckBasicStyle.lua` before committing, it will perform various code style checks and warn you if your code does not conform to our conventions. `CheckBasicStyle.lua` can be configured to run automatically before every commit via a pre-commit hook, \*\*this is highly recommended\*\*. There are instructions on how to achieve this at the bottom of this file.

Here are the conventions:

- We use C++17.

- All new public functions in all classes need documenting comments on what they do and what behavior they follow, use doxy-comments formatted as `/\*\* Description \*/`. Do not use asterisks on additional lines in multi-line comments.

- Use spaces after the comment markers: `// Comment` instead of `//Comment`. A comment must be prefixed with two spaces if it's on the same line with code:

- `SomeFunction()<Space><Space>//<Space>Note the two spaces prefixed to me and the space after the slashes.`

- All variable names and function names use CamelCase style, with the exception of single letter variables.

- `ThisIsAProperFunction()` `This\_is\_bad()` `this\_is\_bad()` `GoodVariableName` `badVariableName`.

- All member variables start with `m\_`, all function parameters start with `a\_`, all class names start with `c`.

- `class cMonster { int m\_Health; int DecreaseHealth(int a\_Amount); }`

- Function parameters that are coordinates should be passed using an appropriate storage container, and not as three separate arguments.

- e.g. for a block position, Vector3i. For an entity position, Vector3d. For a chunk coordinate, cChunkCoords.

- For a 3-dimensional box of blocks, use cCuboid. For an axis-aligned bounding box, use cBoundingBox.

- Parameters smaller than 4 elements (e.g. Vector3, cChunkCoords) should be passed by value. All other parameters should be passed by const reference, where applicable.

- `Foo(Vector3d a\_Param1, const cCuboid & a\_Param2)`

- See the discussion in issue #3853

- Put spaces after commas. `Vector3d(1, 2, 3)` instead of `Vector3d(1,2,3)`

- Put spaces before and after every operator, except unary operators.

- `a = b + c;`

- `if (a == b)`

- `++itr`

- Keep individual functions spaced out by 5 empty lines, this enhances readability and makes navigation in the source file easier.

- Add those extra parentheses to conditions, especially in C++:

- `if ((a == 1) && ((b == 2) || (c == 3)))` instead of ambiguous `if (a == 1 && b == 2 || c == 3)`

- This helps prevent mistakes such as `if (a & 1 == 0)`

- Use the provided wrappers for OS stuff:

- Threading is done by inheriting from `cIsThread`, thread synchronization through `cCriticalSection` and `cEvent`, file access and filesystem operations through the `cFile` class, high-precision timers through `cTimer`, high-precision sleep through `cSleep`

- No magic numbers, use named constants:

- `E\_ITEM\_XXX`, `E\_BLOCK\_XXX` and `E\_META\_XXX` for items and blocks

- `cEntity::etXXX` for entity types, `cMonster::mtXXX` for mob types

- `dimNether`, `dimOverworld` and `dimEnd` for world dimension

- `gmSurvival`, `gmCreative`, `gmAdventure` for game modes

- `wSunny`, `wRain`, `wThunderstorm` for weather

- `cChunkDef::Width`, `cChunkDef::Height` for chunk dimensions (C++)

- etc.

- Instead of checking for a specific value, use an `IsXXX` function, if available:

- `cPlayer:IsGameModeCreative()` instead of` (cPlayer:GetGameMode() == gmCreative)` (the player can also inherit the gamemode from the world, which the value-d condition doesn't catch)

- Please use \*\*tabs for indentation and spaces for alignment\*\*. This means that if it's at line start, it's a tab; if it's in the middle of a line, it's a space

- Alpha-sort stuff that makes sense alpha-sorting - long lists of similar items etc.

- White space is free, so use it freely

- "freely" as in "plentifully", not "arbitrarily"

- All `case` statements inside a `switch` need an extra indent

- Each and every control statement deserves its braces. This helps maintainability later on when the file is edited, lines added or removed - the control logic doesn't break so easily

- The only exception: a `switch` statement with all `case` statements being a single short statement is allowed to use the short brace-less form

- These two rules really mean that indent is governed by braces

- Add an empty last line in all source files (GCC and Git can complain otherwise)

Pre-commit Hook

---------------

When contributing, the code conventions above \*must\* be followed. Otherwise, the CI builds will automatically fail and your PR will not be merged until the non-conforming code is fixed. It is highly recommended to set up a pre-commit hook which will check your code style before every commit. Here is how to do that:

- Clone the repository as usual.

- Go to your `<clone location>/.git/hooks` folder, create a text file named "pre-commit" there with the following contents:

```

#!/bin/sh

src/CheckBasicStyle.lua 1>&2 -g

```

- If on Linux/Unix, you need to give the newly created file an execute permission: `chmod +x .git/hooks/pre-commit`

- Lua must be installed.

- You're done. Now, `src/CheckBasicStyle.lua` will check the changed files before every commit. If a problem is found, it will point you to that problem and will cancel the commit.

Note that the check script is not smart enough to catch everything, so not having any warnings does not necessarily imply that you followed the conventions fully. The other humans working on this will perform more checks before merging.

Github-related conventions

--------------------------

The following rules are not strictly enforced, but if you follow them, you will make it easier for us to review your changes.

- Use good short title for your PRs. `Store Health as float` is good, `Fix #4024` is not good.

- If a PR fixes an issue, mention it in the PR description, but \*\*not\*\* in the commits themselves. Typically, we add the line `Fixes #007` to the bottom of the PR description, this makes Github link the PR to the issue, auto-close the issue when the PR is merged, and also is included in the merge message.

- Focus on one thing only. Each PR should avoid making unrelated changes - those makes the history more difficult to understand later on, and they make the review a headache.

- Feel free to rebase, amend or otherwise manipulate the commits in the PR (of course, as long as you're the only one working on the PR). We mostly squash-merge PRs, so your internal commit structure will not be important in the end.

- If the PR is about one change, but it makes sense to keep the change broken into several commits, make sure each such commit is compilable, encapsulated, and possibly add a note to the PR description so that we consider this.

Copyright

---------

Your must either place your work in the public domain or licensed it under the Apache License 2.0, and if so you must add yourself to the contributors file to show that you accept the publication of your work under the license.

\*\*PLUGINS ONLY\*\*: If your plugin is not licensed under the Apache License 2.0, then it must be compatible and marked as such. This is only valid for the plugins included within the Cuberite source; plugins developed on separate repositories can use whatever license they want.

[1]: https://github.com/cuberite/cuberite/blob/master/GETTING-STARTED.md

[2]: https://egghead.io/courses/how-to-contribute-to-an-open-source-project-on-github