# Contributing

\*\*\*Ask not what Notepad++ can do for you - ask what you can do for Notepad++\*\*\*

## Reporting Issues

Bug reports are appreciated. Following a few guidelines listed below will help speed up the process of getting them fixed.

1. Search the issue tracker to see if it has already been reported.

2. Disable your plugins to see if one of them is the problem. You can do this by renaming your `plugins` folder to something else.

3. Only report an issue with a plugin if it is one of the standard plugins included in the Notepad++ installation. Any other plugin issue should be reported to its respective issue tracker. The standard plugins include (for v6.8.3):

\* NppFTP

\* NppExport

\* Plugin Manager

\* Converter

\* mimeTools

4. Include additional information such as:

\* A detailed explanation

\* Operating System version

\* Notepad++ version

\* List of installed plugins (if it is related to a plugin)

\* Screen shots (if applicable)

\* ...and any other relevant information

## Pull Requests

Your pull requests are welcome; however, they may not be accepted for various reasons. If you want to make some GUI enhancement like renaming some graphic items or fixing typos, please create the issue instead of the pull requests. All Pull Requests, except for translations and user documentation, need to be attached to a issue on GitHub. For Pull Requests regarding enhancements and questions, the issue must first be approved by one of project's administrators before being merged into the project. An approved issue will have the label `Accepted`. For issues that have not been accepted, you may request to be assigned to that issue.

Opening a issue beforehand allows the administrators and the community to discuss bugs and enhancements before work begins, preventing wasted effort.

### Guidelines for pull requests

1. Respect Notepad++ coding style.

2. Create a new branch for each PR. \*\*Make sure your branch name wasn't used before\*\* - you can add date (for exemple `patch3\_20200528`) to ensure its unicity.

3. Single feature or bug-fix per PR.

4. Make single commit per PR.

5. Make your modification compact - don't reformat source code in your request. It makes code review more difficult.

6. PR of reformatting (changing of ws/TAB, line endings or coding style) of source code won't be accepted. Use issue trackers for your request instead.

7. Typo fixing and code refactoring won't be accepted - please create issues with title started with `TYPO` to request the changing.

In short: The easier the code review is, the better the chance your pull request will get accepted.

### Coding style

![stay clean](https://notepad-plus-plus.org/assets/images/good-bad-practice.jpg)

#### GENERAL

1. ##### Do not use Java-like braces.

\* ###### Good:

```cpp

if ()

{

// Do something

}

```

\* ###### Bad:

```cpp

if () {

// Do something

}

```

2. ##### Use tabs instead of white-spaces (we usually set our editors to 4 white-spaces for 1 tab, but the choice is up to you).

3. ##### Always leave one space before and after binary and ternary operators.

\* ###### Good:

```cpp

if (a == 10 && b == 42)

```

\* ###### Bad:

```cpp

if (a==10&&b==42)

```

4. ##### Only leave one space after semi-colons in "for" statements.

\* ###### Good:

```cpp

for (int i = 0; i != 10; ++i)

```

\* ###### Bad:

```cpp

for(int i=0;i<10;++i)

```

5. ##### Function names are not separated from the first parenthesis.

\* ###### Good:

```cpp

foo();

myObject.foo(24);

```

\* ###### Bad:

```cpp

foo ();

```

6. ##### Keywords are separated from the first parenthesis by one space.

\* ###### Good:

```cpp

if (true)

while (true)

```

\* ###### Bad:

```cpp

if(myCondition)

```

7. ##### Use the following indenting for "switch" statements:

```cpp

switch (test)

{

case 1:

{

// Do something

break;

}

default:

// Do something else

} // No semi-colon here

```

8. ##### Avoid magic numbers.

\* ###### Good:

```cpp

if (foo < I\_CAN\_PUSH\_ON\_THE\_RED\_BUTTON)

startThermoNuclearWar();

```

\* ###### Bad:

```cpp

while (lifeTheUniverseAndEverything != 42)

lifeTheUniverseAndEverything = buildMorePowerfulComputerForTheAnswer();

```

9. ##### Prefer enums for integer constants.

10. ##### Use initialization with curly braces.

\* ###### Good:

```cpp

MyClass instance{10.4};

```

\* ###### Bad:

```cpp

MyClass instance(10.4);

```

11. ##### Always use `empty()` for testing if a string is empty or not.

\* ###### Good:

```cpp

if (not string.empty())

...

```

\* ###### Bad:

```cpp

if (string != "")

...

```

12. ##### Always use `C++ conversion` instead of `C-Style cast`. Generally, all the conversion among types should be avoided. If you have no choice, use C++ conversion.

\* ###### Good:

```cpp

char aChar = static\_cast<char>(\_pEditView->execute(SCI\_GETCHARAT, j));

```

\* ###### Bad:

```cpp

char aChar = (char)\_pEditView->execute(SCI\_GETCHARAT, j);

```

#### NAMING CONVENTIONS

1. ##### Classes uses Pascal Case

\* ###### Good:

```cpp

class IAmAClass

{};

```

\* ###### Bad:

```cpp

class iAmAClass

{};

class I\_am\_a\_class

{};

```

2. ##### Methods & method parameters use camel Case

```cpp

void myMethod(uint myVeryLongParameter);

```

3. ##### Member variables

Any member variable name of class/struct should be preceded by an underscore.

```cpp

public:

int \_publicAttribute;

private:

int \_pPrivateAttribute;

float \_pAccount;

```

4. ##### Always prefer a variable name that describes what the variable is used for.

\* ###### Good:

```cpp

if (hours < 24 && minutes < 60 && seconds < 60)

```

\* ###### Bad:

```cpp

if (a < 24 && b < 60 && c < 60)

```

#### COMMENTS

1. ##### Use C++ comment line style than C comment style.

\* ###### Good:

```

// Two lines comment

// Use still C++ comment line style

```

\* ###### Bad:

```

/\*

Please don't piss me off with that

\*/

```

#### BEST PRACTICES

1. ##### Use C++11/14/17 whenever it is possible

2. ##### Use C++11 member initialization feature whenever it is possible

```cpp

class Foo

{

int value = 0;

};

```

3. ##### Prefer this form:

```cpp

++i

```

\*\*to:\*\*

```cpp

i++

```

(It does not change anything for built-in types but it would bring consistency)

4. ##### Avoid using pointers. Prefer references. You might need the variable to be assigned a NULL value: in this case the NULL value has semantics and must be checked. Wherever possible, use a SmartPtr instead of old-school pointers.

5. ##### Avoid using new if you can use automatic variable. However, avoid `shared\_ptr` as much as possible. Prefer `unique\_ptr` instead.

6. ##### Don't place any "using namespace" directives in headers.

7. ##### Compile time is without incidence. Increasing compile time to reduce execution time is encouraged.

8. ##### Code legibility and length is less important than easy and fast end-user experience.