# Terser

[. Check out [PATRONS.md](https://github.com/terser/terser/blob/master/PATRONS.md) for our first-tier patrons. Terser recommends you use RollupJS to bundle your modules, as that produces smaller code overall. \*Beautification\* has been undocumented and is \*being removed\* from terser, we recommend you use [prettier](https://npmjs.com/package/prettier). Find the changelog in [CHANGELOG.md](https://github.com/terser/terser/blob/master/CHANGELOG.md) [npm-image]: https://img.shields.io/npm/v/terser.svg [npm-url]: https://npmjs.org/package/terser [downloads-image]: https://img.shields.io/npm/dm/terser.svg [downloads-url]: https://npmjs.org/package/terser [travis-image]: https://img.shields.io/travis/terser/terser/master.svg [travis-url]: https://travis-ci.org/terser/terser [opencollective-contributors]: https://opencollective.com/terser/tiers/badge.svg [opencollective-url]: https://opencollective.com/terser Why choose terser? ------------------ `uglify-es` is [no longer maintained](https://github.com/mishoo/UglifyJS2/issues/3156#issuecomment-392943058) and `uglify-js` does not support ES6+. \*\*`terser`\*\* is a fork of `uglify-es` that mostly retains API and CLI compatibility with `uglify-es` and `uglify-js@3`. Install ------- First make sure you have installed the latest version of [node.js](http://nodejs.org/) (You may need to restart your computer after this step). From NPM for use as a command line app: npm install terser -g From NPM for programmatic use: npm install terser # Command line usage terser [input files] [options] Terser can take multiple input files. It's recommended that you pass the input files first, then pass the options. Terser will parse input files in sequence and apply any compression options. The files are parsed in the same global scope, that is, a reference from a file to some variable/function declared in another file will be matched properly. If no input file is specified, Terser will read from STDIN. If you wish to pass your options before the input files, separate the two with a double dash to prevent input files being used as option arguments: terser --compress --mangle -- input.js ### Command line options ``` -h, --help Print usage information. `--help options` for details on available options. -V, --version Print version number. -p, --parse Specify parser options: `acorn` Use Acorn for parsing. `bare\_returns` Allow return outside of functions. Useful when minifying CommonJS modules and Userscripts that may be anonymous function wrapped (IIFE) by the .user.js engine `caller`. `expression` Parse a single expression, rather than a program (for parsing JSON). `spidermonkey` Assume input files are SpiderMonkey AST format (as JSON). -c, --compress [options] Enable compressor/specify compressor options: `pure\_funcs` List of functions that can be safely removed when their return values are not used. -m, --mangle [options] Mangle names/specify mangler options: `reserved` List of names that should not be mangled. --mangle-props [options] Mangle properties/specify mangler options: `builtins` Mangle property names that overlaps with standard JavaScript globals and DOM API props. `debug` Add debug prefix and suffix. `keep\_quoted` Only mangle unquoted properties, quoted properties are automatically reserved. `strict` disables quoted properties being automatically reserved. `regex` Only mangle matched property names. `reserved` List of names that should not be mangled. -b, --beautify [options] Specify output options: `preamble` Preamble to prepend to the output. You can use this to insert a comment, for example for licensing information. This will not be parsed, but the source map will adjust for its presence. `quote\_style` Quote style: 0 - auto 1 - single 2 - double 3 - original `wrap\_iife` Wrap IIFEs in parenthesis. Note: you may want to disable `negate\_iife` under compressor options. `wrap\_func\_args` Wrap function arguments in parenthesis. -o, --output Output file path (default STDOUT). Specify `ast` or `spidermonkey` to write Terser or SpiderMonkey AST as JSON to STDOUT respectively. --comments [filter] Preserve copyright comments in the output. By default this works like Google Closure, keeping JSDoc-style comments that contain "@license" or "@preserve". You can optionally pass one of the following arguments to this flag: - "all" to keep all comments - `false` to omit comments in the output - a valid JS RegExp like `/foo/` or `/^!/` to keep only matching comments. Note that currently not \*all\* comments can be kept when compression is on, because of dead code removal or cascading statements into sequences. --config-file Read `minify()` options from JSON file. -d, --define [=value] Global definitions. --ecma Specify ECMAScript release: 5, 2015, 2016, etc. -e, --enclose [arg[:value]] Embed output in a big function with configurable arguments and values. --ie8 Support non-standard Internet Explorer 8. Equivalent to setting `ie8: true` in `minify()` for `compress`, `mangle` and `output` options. By default Terser will not try to be IE-proof. --keep-classnames Do not mangle/drop class names. --keep-fnames Do not mangle/drop function names. Useful for code relying on Function.prototype.name. --module Input is an ES6 module. If `compress` or `mangle` is enabled then the `toplevel` option will be enabled. --name-cache File to hold mangled name mappings. --safari10 Support non-standard Safari 10/11. Equivalent to setting `safari10: true` in `minify()` for `mangle` and `output` options. By default `terser` will not work around Safari 10/11 bugs. --source-map [options] Enable source map/specify source map options: `base` Path to compute relative paths from input files. `content` Input source map, useful if you're compressing JS that was generated from some other original code. Specify "inline" if the source map is included within the sources. `filename` Name and/or location of the output source. `includeSources` Pass this flag if you want to include the content of source files in the source map as sourcesContent property. `root` Path to the original source to be included in the source map. `url` If specified, path to the source map to append in `//# sourceMappingURL`. --timings Display operations run time on STDERR. --toplevel Compress and/or mangle variables in top level scope. --verbose Print diagnostic messages. --warn Print warning messages. --wrap Embed everything in a big function, making the “exports” and “global” variables available. You need to pass an argument to this option to specify the name that your module will take when included in, say, a browser. ``` Specify `--output` (`-o`) to declare the output file. Otherwise the output goes to STDOUT. ## CLI source map options Terser can generate a source map file, which is highly useful for debugging your compressed JavaScript. To get a source map, pass `--source-map --output output.js` (source map will be written out to `output.js.map`). Additional options: - `--source-map "filename=''"` to specify the name of the source map. - `--source-map "root=''"` to pass the URL where the original files can be found. - `--source-map "url=''"` to specify the URL where the source map can be found. Otherwise Terser assumes HTTP `X-SourceMap` is being used and will omit the `//# sourceMappingURL=` directive. For example: terser js/file1.js js/file2.js \ -o foo.min.js -c -m \ --source-map "root='http://foo.com/src',url='foo.min.js.map'" The above will compress and mangle `file1.js` and `file2.js`, will drop the output in `foo.min.js` and the source map in `foo.min.js.map`. The source mapping will refer to `http://foo.com/src/js/file1.js` and `http://foo.com/src/js/file2.js` (in fact it will list `http://foo.com/src` as the source map root, and the original files as `js/file1.js` and `js/file2.js`). ### Composed source map When you're compressing JS code that was output by a compiler such as CoffeeScript, mapping to the JS code won't be too helpful. Instead, you'd like to map back to the original code (i.e. CoffeeScript). Terser has an option to take an input source map. Assuming you have a mapping from CoffeeScript → compiled JS, Terser can generate a map from CoffeeScript → compressed JS by mapping every token in the compiled JS to its original location. To use this feature pass `--source-map "content='/path/to/input/source.map'"` or `--source-map "content=inline"` if the source map is included inline with the sources. ## CLI compress options You need to pass `--compress` (`-c`) to enable the compressor. Optionally you can pass a comma-separated list of [compress options](#compress-options). Options are in the form `foo=bar`, or just `foo` (the latter implies a boolean option that you want to set `true`; it's effectively a shortcut for `foo=true`). Example: terser file.js -c toplevel,sequences=false ## CLI mangle options To enable the mangler you need to pass `--mangle` (`-m`). The following (comma-separated) options are supported: - `toplevel` (default `false`) -- mangle names declared in the top level scope. - `eval` (default `false`) -- mangle names visible in scopes where `eval` or `with` are used. When mangling is enabled but you want to prevent certain names from being mangled, you can declare those names with `--mangle reserved` — pass a comma-separated list of names. For example: terser ... -m reserved=['$','require','exports'] to prevent the `require`, `exports` and `$` names from being changed. ### CLI mangling property names (`--mangle-props`) \*\*Note:\*\* THIS \*\*WILL\*\* BREAK YOUR CODE. A good rule of thumb is not to use this unless you know exactly what you're doing and how this works and read this section until the end. Mangling property names is a separate step, different from variable name mangling. Pass `--mangle-props` to enable it. The least dangerous way to use this is to use the `regex` option like so: ``` terser example.js -c -m --mangle-props regex=/\_$/ ``` This will mangle all properties that end with an underscore. So you can use it to mangle internal methods. By default, it will mangle all properties in the input code with the exception of built in DOM properties and properties in core JavaScript classes, which is what will break your code if you don't: 1. Control all the code you're mangling 2. Avoid using a module bundler, as they usually will call Terser on each file individually, making it impossible to pass mangled objects between modules. 3. Avoid calling functions like `defineProperty` or `hasOwnProperty`, because they refer to object properties using strings and will break your code if you don't know what you are doing. An example: ```javascript // example.js var x = { baz\_: 0, foo\_: 1, calc: function() { return this.foo\_ + this.baz\_; } }; x.bar\_ = 2; x["baz\_"] = 3; console.log(x.calc()); ``` Mangle all properties (except for JavaScript `builtins`) (\*\*very\*\* unsafe): ```bash $ terser example.js -c passes=2 -m --mangle-props ``` ```javascript var x={o:3,t:1,i:function(){return this.t+this.o},s:2};console.log(x.i()); ``` Mangle all properties except for `reserved` properties (still very unsafe): ```bash $ terser example.js -c passes=2 -m --mangle-props reserved=[foo\_,bar\_] ``` ```javascript var x={o:3,foo\_:1,t:function(){return this.foo\_+this.o},bar\_:2};console.log(x.t()); ``` Mangle all properties matching a `regex` (not as unsafe but still unsafe): ```bash $ terser example.js -c passes=2 -m --mangle-props regex=/\_$/ ``` ```javascript var x={o:3,t:1,calc:function(){return this.t+this.o},i:2};console.log(x.calc()); ``` Combining mangle properties options: ```bash $ terser example.js -c passes=2 -m --mangle-props regex=/\_$/,reserved=[bar\_] ``` ```javascript var x={o:3,t:1,calc:function(){return this.t+this.o},bar\_:2};console.log(x.calc()); ``` In order for this to be of any use, we avoid mangling standard JS names and DOM API properties by default (`--mangle-props builtins` to override). A regular expression can be used to define which property names should be mangled. For example, `--mangle-props regex=/^\_/` will only mangle property names that start with an underscore. When you compress multiple files using this option, in order for them to work together in the end we need to ensure somehow that one property gets mangled to the same name in all of them. For this, pass `--name-cache filename.json` and Terser will maintain these mappings in a file which can then be reused. It should be initially empty. Example: ```bash $ rm -f /tmp/cache.json # start fresh $ terser file1.js file2.js --mangle-props --name-cache /tmp/cache.json -o part1.js $ terser file3.js file4.js --mangle-props --name-cache /tmp/cache.json -o part2.js ``` Now, `part1.js` and `part2.js` will be consistent with each other in terms of mangled property names. Using the name cache is not necessary if you compress all your files in a single call to Terser. ### Mangling unquoted names (`--mangle-props keep\_quoted`) Using quoted property name (`o["foo"]`) reserves the property name (`foo`) so that it is not mangled throughout the entire script even when used in an unquoted style (`o.foo`). Example: ```javascript // stuff.js var o = { "foo": 1, bar: 3 }; o.foo += o.bar; console.log(o.foo); ``` ```bash $ terser stuff.js --mangle-props keep\_quoted -c -m ``` ```javascript var o={foo:1,o:3};o.foo+=o.o,console.log(o.foo); ``` ### Debugging property name mangling You can also pass `--mangle-props debug` in order to mangle property names without completely obscuring them. For example the property `o.foo` would mangle to `o.\_$foo$\_` with this option. This allows property mangling of a large codebase while still being able to debug the code and identify where mangling is breaking things. ```bash $ terser stuff.js --mangle-props debug -c -m ``` ```javascript var o={\_$foo$\_:1,\_$bar$\_:3};o.\_$foo$\_+=o.\_$bar$\_,console.log(o.\_$foo$\_); ``` You can also pass a custom suffix using `--mangle-props debug=XYZ`. This would then mangle `o.foo` to `o.\_$foo$XYZ\_`. You can change this each time you compile a script to identify how a property got mangled. One technique is to pass a random number on every compile to simulate mangling changing with different inputs (e.g. as you update the input script with new properties), and to help identify mistakes like writing mangled keys to storage. # API Reference Assuming installation via NPM, you can load Terser in your application like this: ```javascript var Terser = require("terser"); ``` Browser loading is also supported: ```html ``` There is a single high level function, \*\*`minify(code, options)`\*\*, which will perform all minification [phases](#minify-options) in a configurable manner. By default `minify()` will enable the options [`compress`](#compress-options) and [`mangle`](#mangle-options). Example: ```javascript var code = "function add(first, second) { return first + second; }"; var result = Terser.minify(code); console.log(result.error); // runtime error, or `undefined` if no error console.log(result.code); // minified output: function add(n,d){return n+d} ``` You can `minify` more than one JavaScript file at a time by using an object for the first argument where the keys are file names and the values are source code: ```javascript var code = { "file1.js": "function add(first, second) { return first + second; }", "file2.js": "console.log(add(1 + 2, 3 + 4));" }; var result = Terser.minify(code); console.log(result.code); // function add(d,n){return d+n}console.log(add(3,7)); ``` The `toplevel` option: ```javascript var code = { "file1.js": "function add(first, second) { return first + second; }", "file2.js": "console.log(add(1 + 2, 3 + 4));" }; var options = { toplevel: true }; var result = Terser.minify(code, options); console.log(result.code); // console.log(3+7); ``` The `nameCache` option: ```javascript var options = { mangle: { toplevel: true, }, nameCache: {} }; var result1 = Terser.minify({ "file1.js": "function add(first, second) { return first + second; }" }, options); var result2 = Terser.minify({ "file2.js": "console.log(add(1 + 2, 3 + 4));" }, options); console.log(result1.code); // function n(n,r){return n+r} console.log(result2.code); // console.log(n(3,7)); ``` You may persist the name cache to the file system in the following way: ```javascript var cacheFileName = "/tmp/cache.json"; var options = { mangle: { properties: true, }, nameCache: JSON.parse(fs.readFileSync(cacheFileName, "utf8")) }; fs.writeFileSync("part1.js", Terser.minify({ "file1.js": fs.readFileSync("file1.js", "utf8"), "file2.js": fs.readFileSync("file2.js", "utf8") }, options).code, "utf8"); fs.writeFileSync("part2.js", Terser.minify({ "file3.js": fs.readFileSync("file3.js", "utf8"), "file4.js": fs.readFileSync("file4.js", "utf8") }, options).code, "utf8"); fs.writeFileSync(cacheFileName, JSON.stringify(options.nameCache), "utf8"); ``` An example of a combination of `minify()` options: ```javascript var code = { "file1.js": "function add(first, second) { return first + second; }", "file2.js": "console.log(add(1 + 2, 3 + 4));" }; var options = { toplevel: true, compress: { global\_defs: { "@console.log": "alert" }, passes: 2 }, output: { beautify: false, preamble: "/\* minified \*/" } }; var result = Terser.minify(code, options); console.log(result.code); // /\* minified \*/ // alert(10);" ``` To produce warnings: ```javascript var code = "function f(){ var u; return 2 + 3; }"; var options = { warnings: true }; var result = Terser.minify(code, options); console.log(result.error); // runtime error, `undefined` in this case console.log(result.warnings); // [ 'Dropping unused variable u [0:1,18]' ] console.log(result.code); // function f(){return 5} ``` An error example: ```javascript var result = Terser.minify({"foo.js" : "if (0) else console.log(1);"}); console.log(JSON.stringify(result.error)); // {"message":"Unexpected token: keyword (else)","filename":"foo.js","line":1,"col":7,"pos":7} ``` Note: unlike `uglify-js@2.x`, the Terser API does not throw errors. To achieve a similar effect one could do the following: ```javascript var result = Terser.minify(code, options); if (result.error) throw result.error; ``` ## Minify options - `ecma` (default `undefined`) - pass `5`, `2015`, `2016`, etc to override `parse`, `compress` and `output`'s `ecma` options. - `warnings` (default `false`) — pass `true` to return compressor warnings in `result.warnings`. Use the value `"verbose"` for more detailed warnings. - `parse` (default `{}`) — pass an object if you wish to specify some additional [parse options](#parse-options). - `compress` (default `{}`) — pass `false` to skip compressing entirely. Pass an object to specify custom [compress options](#compress-options). - `mangle` (default `true`) — pass `false` to skip mangling names, or pass an object to specify [mangle options](#mangle-options) (see below). - `mangle.properties` (default `false`) — a subcategory of the mangle option. Pass an object to specify custom [mangle property options](#mangle-properties-options). - `module` (default `false`) — Use when minifying an ES6 module. "use strict" is implied and names can be mangled on the top scope. If `compress` or `mangle` is enabled then the `toplevel` option will be enabled. - `output` (default `null`) — pass an object if you wish to specify additional [output options](#output-options). The defaults are optimized for best compression. - `sourceMap` (default `false`) - pass an object if you wish to specify [source map options](#source-map-options). - `toplevel` (default `false`) - set to `true` if you wish to enable top level variable and function name mangling and to drop unused variables and functions. - `nameCache` (default `null`) - pass an empty object `{}` or a previously used `nameCache` object if you wish to cache mangled variable and property names across multiple invocations of `minify()`. Note: this is a read/write property. `minify()` will read the name cache state of this object and update it during minification so that it may be reused or externally persisted by the user. - `ie8` (default `false`) - set to `true` to support IE8. - `keep\_classnames` (default: `undefined`) - pass `true` to prevent discarding or mangling of class names. Pass a regular expression to only keep class names matching that regex. - `keep\_fnames` (default: `false`) - pass `true` to prevent discarding or mangling of function names. Pass a regular expression to only keep class names matching that regex. Useful for code relying on `Function.prototype.name`. If the top level minify option `keep\_classnames` is `undefined` it will be overridden with the value of the top level minify option `keep\_fnames`. - `safari10` (default: `false`) - pass `true` to work around Safari 10/11 bugs in loop scoping and `await`. See `safari10` options in [`mangle`](#mangle-options) and [`output`](#output-options) for details. ## Minify options structure ```javascript { parse: { // parse options }, compress: { // compress options }, mangle: { // mangle options properties: { // mangle property options } }, output: { // output options }, sourceMap: { // source map options }, ecma: 5, // specify one of: 5, 2015, 2016, etc. keep\_classnames: false, keep\_fnames: false, ie8: false, module: false, nameCache: null, // or specify a name cache object safari10: false, toplevel: false, warnings: false, } ``` ### Source map options To generate a source map: ```javascript var result = Terser.minify({"file1.js": "var a = function() {};"}, { sourceMap: { filename: "out.js", url: "out.js.map" } }); console.log(result.code); // minified output console.log(result.map); // source map ``` Note that the source map is not saved in a file, it's just returned in `result.map`. The value passed for `sourceMap.url` is only used to set `//# sourceMappingURL=out.js.map` in `result.code`. The value of `filename` is only used to set `file` attribute (see [the spec][sm-spec]) in source map file. You can set option `sourceMap.url` to be `"inline"` and source map will be appended to code. You can also specify sourceRoot property to be included in source map: ```javascript var result = Terser.minify({"file1.js": "var a = function() {};"}, { sourceMap: { root: "http://example.com/src", url: "out.js.map" } }); ``` If you're compressing compiled JavaScript and have a source map for it, you can use `sourceMap.content`: ```javascript var result = Terser.minify({"compiled.js": "compiled code"}, { sourceMap: { content: "content from compiled.js.map", url: "minified.js.map" } }); // same as before, it returns `code` and `map` ``` If you're using the `X-SourceMap` header instead, you can just omit `sourceMap.url`. If you happen to need the source map as a raw object, set `sourceMap.asObject` to `true`. ## Parse options - `bare\_returns` (default `false`) -- support top level `return` statements - `ecma` (default: `2017`) -- specify one of `5`, `2015`, `2016` or `2017`. Note: this setting is not presently enforced except for ES8 optional trailing commas in function parameter lists and calls with `ecma` `2017`. - `html5\_comments` (default `true`) - `shebang` (default `true`) -- support `#!command` as the first line ## Compress options - `defaults` (default: `true`) -- Pass `false` to disable most default enabled `compress` transforms. Useful when you only want to enable a few `compress` options while disabling the rest. - `arrows` (default: `true`) -- Class and object literal methods are converted will also be converted to arrow expressions if the resultant code is shorter: `m(){return x}` becomes `m:()=>x`. To do this to regular ES5 functions which don't use `this` or `arguments`, see `unsafe\_arrows`. - `arguments` (default: `false`) -- replace `arguments[index]` with function parameter name whenever possible. - `booleans` (default: `true`) -- various optimizations for boolean context, for example `!!a ? b : c → a ? b : c` - `booleans\_as\_integers` (default: `false`) -- Turn booleans into 0 and 1, also makes comparisons with booleans use `==` and `!=` instead of `===` and `!==`. - `collapse\_vars` (default: `true`) -- Collapse single-use non-constant variables, side effects permitting. - `comparisons` (default: `true`) -- apply certain optimizations to binary nodes, e.g. `!(a <= b) → a > b` (only when `unsafe\_comps`), attempts to negate binary nodes, e.g. `a = !b && !c && !d && !e → a=!(b||c||d||e)` etc. - `computed\_props` (default: `true`) -- Transforms constant computed properties into regular ones: `{["computed"]: 1}` is converted to `{computed: 1}`. - `conditionals` (default: `true`) -- apply optimizations for `if`-s and conditional expressions - `dead\_code` (default: `true`) -- remove unreachable code - `directives` (default: `true`) -- remove redundant or non-standard directives - `drop\_console` (default: `false`) -- Pass `true` to discard calls to `console.\*` functions. If you wish to drop a specific function call such as `console.info` and/or retain side effects from function arguments after dropping the function call then use `pure\_funcs` instead. - `drop\_debugger` (default: `true`) -- remove `debugger;` statements - `ecma` (default: `5`) -- Pass `2015` or greater to enable `compress` options that will transform ES5 code into smaller ES6+ equivalent forms. - `evaluate` (default: `true`) -- attempt to evaluate constant expressions - `expression` (default: `false`) -- Pass `true` to preserve completion values from terminal statements without `return`, e.g. in bookmarklets. - `global\_defs` (default: `{}`) -- see [conditional compilation](#conditional-compilation) - `hoist\_funs` (default: `false`) -- hoist function declarations - `hoist\_props` (default: `true`) -- hoist properties from constant object and array literals into regular variables subject to a set of constraints. For example: `var o={p:1, q:2}; f(o.p, o.q);` is converted to `f(1, 2);`. Note: `hoist\_props` works best with `mangle` enabled, the `compress` option `passes` set to `2` or higher, and the `compress` option `toplevel` enabled. - `hoist\_vars` (default: `false`) -- hoist `var` declarations (this is `false` by default because it seems to increase the size of the output in general) - `if\_return` (default: `true`) -- optimizations for if/return and if/continue - `inline` (default: `true`) -- inline calls to function with simple/`return` statement: - `false` -- same as `0` - `0` -- disabled inlining - `1` -- inline simple functions - `2` -- inline functions with arguments - `3` -- inline functions with arguments and variables - `true` -- same as `3` - `join\_vars` (default: `true`) -- join consecutive `var` statements - `keep\_classnames` (default: `false`) -- Pass `true` to prevent the compressor from discarding class names. Pass a regular expression to only keep class names matching that regex. See also: the `keep\_classnames` [mangle option](#mangle). - `keep\_fargs` (default: `true`) -- Prevents the compressor from discarding unused function arguments. You need this for code which relies on `Function.length`. - `keep\_fnames` (default: `false`) -- Pass `true` to prevent the compressor from discarding function names. Pass a regular expression to only keep function names matching that regex. Useful for code relying on `Function.prototype.name`. See also: the `keep\_fnames` [mangle option](#mangle). - `keep\_infinity` (default: `false`) -- Pass `true` to prevent `Infinity` from being compressed into `1/0`, which may cause performance issues on Chrome. - `loops` (default: `true`) -- optimizations for `do`, `while` and `for` loops when we can statically determine the condition. - `module` (default `false`) -- Pass `true` when compressing an ES6 module. Strict mode is implied and the `toplevel` option as well. - `negate\_iife` (default: `true`) -- negate "Immediately-Called Function Expressions" where the return value is discarded, to avoid the parens that the code generator would insert. - `passes` (default: `1`) -- The maximum number of times to run compress. In some cases more than one pass leads to further compressed code. Keep in mind more passes will take more time. - `properties` (default: `true`) -- rewrite property access using the dot notation, for example `foo["bar"] → foo.bar` - `pure\_funcs` (default: `null`) -- You can pass an array of names and Terser will assume that those functions do not produce side effects. DANGER: will not check if the name is redefined in scope. An example case here, for instance `var q = Math.floor(a/b)`. If variable `q` is not used elsewhere, Terser will drop it, but will still keep the `Math.floor(a/b)`, not knowing what it does. You can pass `pure\_funcs: [ 'Math.floor' ]` to let it know that this function won't produce any side effect, in which case the whole statement would get discarded. The current implementation adds some overhead (compression will be slower). - `pure\_getters` (default: `"strict"`) -- If you pass `true` for this, Terser will assume that object property access (e.g. `foo.bar` or `foo["bar"]`) doesn't have any side effects. Specify `"strict"` to treat `foo.bar` as side-effect-free only when `foo` is certain to not throw, i.e. not `null` or `undefined`. - `reduce\_funcs` (legacy option, safely ignored for backwards compatibility). - `reduce\_vars` (default: `true`) -- Improve optimization on variables assigned with and used as constant values. - `sequences` (default: `true`) -- join consecutive simple statements using the comma operator. May be set to a positive integer to specify the maximum number of consecutive comma sequences that will be generated. If this option is set to `true` then the default `sequences` limit is `200`. Set option to `false` or `0` to disable. The smallest `sequences` length is `2`. A `sequences` value of `1` is grandfathered to be equivalent to `true` and as such means `200`. On rare occasions the default sequences limit leads to very slow compress times in which case a value of `20` or less is recommended. - `side\_effects` (default: `true`) -- Pass `false` to disable potentially dropping function calls marked as "pure". A function call is marked as "pure" if a comment annotation `/\*@\_\_PURE\_\_\*/` or `/\*#\_\_PURE\_\_\*/` immediately precedes the call. For example: `/\*@\_\_PURE\_\_\*/foo();` - `switches` (default: `true`) -- de-duplicate and remove unreachable `switch` branches - `toplevel` (default: `false`) -- drop unreferenced functions (`"funcs"`) and/or variables (`"vars"`) in the top level scope (`false` by default, `true` to drop both unreferenced functions and variables) - `top\_retain` (default: `null`) -- prevent specific toplevel functions and variables from `unused` removal (can be array, comma-separated, RegExp or function. Implies `toplevel`) - `typeofs` (default: `true`) -- Transforms `typeof foo == "undefined"` into `foo === void 0`. Note: recommend to set this value to `false` for IE10 and earlier versions due to known issues. - `unsafe` (default: `false`) -- apply "unsafe" transformations ([details](#the-unsafe-compress-option)). - `unsafe\_arrows` (default: `false`) -- Convert ES5 style anonymous function expressions to arrow functions if the function body does not reference `this`. Note: it is not always safe to perform this conversion if code relies on the the function having a `prototype`, which arrow functions lack. This transform requires that the `ecma` compress option is set to `2015` or greater. - `unsafe\_comps` (default: `false`) -- Reverse `<` and `<=` to `>` and `>=` to allow improved compression. This might be unsafe when an at least one of two operands is an object with computed values due the use of methods like `get`, or `valueOf`. This could cause change in execution order after operands in the comparison are switching. Compression only works if both `comparisons` and `unsafe\_comps` are both set to true. - `unsafe\_Function` (default: `false`) -- compress and mangle `Function(args, code)` when both `args` and `code` are string literals. - `unsafe\_math` (default: `false`) -- optimize numerical expressions like `2 \* x \* 3` into `6 \* x`, which may give imprecise floating point results. - `unsafe\_symbols` (default: `false`) -- removes keys from native Symbol declarations, e.g `Symbol("kDog")` becomes `Symbol()`. - `unsafe\_methods` (default: false) -- Converts `{ m: function(){} }` to `{ m(){} }`. `ecma` must be set to `6` or greater to enable this transform. If `unsafe\_methods` is a RegExp then key/value pairs with keys matching the RegExp will be converted to concise methods. Note: if enabled there is a risk of getting a "`` is not a constructor" TypeError should any code try to `new` the former function. - `unsafe\_proto` (default: `false`) -- optimize expressions like `Array.prototype.slice.call(a)` into `[].slice.call(a)` - `unsafe\_regexp` (default: `false`) -- enable substitutions of variables with `RegExp` values the same way as if they are constants. - `unsafe\_undefined` (default: `false`) -- substitute `void 0` if there is a variable named `undefined` in scope (variable name will be mangled, typically reduced to a single character) - `unused` (default: `true`) -- drop unreferenced functions and variables (simple direct variable assignments do not count as references unless set to `"keep\_assign"`) - `warnings` (default: `false`) -- display warnings when dropping unreachable code or unused declarations etc. ## Mangle options - `eval` (default `false`) -- Pass `true` to mangle names visible in scopes where `eval` or `with` are used. - `keep\_classnames` (default `false`) -- Pass `true` to not mangle class names. Pass a regular expression to only keep class names matching that regex. See also: the `keep\_classnames` [compress option](#compress-options). - `keep\_fnames` (default `false`) -- Pass `true` to not mangle function names. Pass a regular expression to only keep class names matching that regex. Useful for code relying on `Function.prototype.name`. See also: the `keep\_fnames` [compress option](#compress-options). - `module` (default `false`) -- Pass `true` an ES6 modules, where the toplevel scope is not the global scope. Implies `toplevel`. - `reserved` (default `[]`) -- Pass an array of identifiers that should be excluded from mangling. Example: `["foo", "bar"]`. - `toplevel` (default `false`) -- Pass `true` to mangle names declared in the top level scope. - `safari10` (default `false`) -- Pass `true` to work around the Safari 10 loop iterator [bug](https://bugs.webkit.org/show\_bug.cgi?id=171041) "Cannot declare a let variable twice". See also: the `safari10` [output option](#output-options). Examples: ```javascript // test.js var globalVar; function funcName(firstLongName, anotherLongName) { var myVariable = firstLongName + anotherLongName; } ``` ```javascript var code = fs.readFileSync("test.js", "utf8"); Terser.minify(code).code; // 'function funcName(a,n){}var globalVar;' Terser.minify(code, { mangle: { reserved: ['firstLongName'] } }).code; // 'function funcName(firstLongName,a){}var globalVar;' Terser.minify(code, { mangle: { toplevel: true } }).code; // 'function n(n,a){}var a;' ``` ### Mangle properties options - `builtins` (default: `false`) — Use `true` to allow the mangling of builtin DOM properties. Not recommended to override this setting. - `debug` (default: `false`) — Mangle names with the original name still present. Pass an empty string `""` to enable, or a non-empty string to set the debug suffix. - `keep\_quoted` (default: `false`) — Only mangle unquoted property names. - `true` -- Quoted property names are automatically reserved and any unquoted property names will not be mangled. - `"strict"` -- Advanced, all unquoted property names are mangled unless explicitly reserved. - `regex` (default: `null`) — Pass a [RegExp literal or pattern string](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/RegExp) to only mangle property matching the regular expression. - `reserved` (default: `[]`) — Do not mangle property names listed in the `reserved` array. - `undeclared` (default: `false`) - Mangle those names when they are accessed as properties of known top level variables but their declarations are never found in input code. May be useful when only minifying parts of a project. See [#397](https://github.com/terser/terser/issues/397) for more details. ## Output options The code generator tries to output shortest code possible by default. In case you want beautified output, pass `--beautify` (`-b`). Optionally you can pass additional arguments that control the code output: - `ascii\_only` (default `false`) -- escape Unicode characters in strings and regexps (affects directives with non-ascii characters becoming invalid) - `beautify` (default `true`) -- whether to actually beautify the output. Passing `-b` will set this to true, but you might need to pass `-b` even when you want to generate minified code, in order to specify additional arguments, so you can use `-b beautify=false` to override it. - `braces` (default `false`) -- always insert braces in `if`, `for`, `do`, `while` or `with` statements, even if their body is a single statement. - `comments` (default `"some"`) -- by default it keeps JSDoc-style comments that contain "@license" or "@preserve", pass `true` or `"all"` to preserve all comments, `false` to omit comments in the output, a regular expression string (e.g. `/^!/`) or a function. - `ecma` (default `5`) -- set output printing mode. Set `ecma` to `2015` or greater to emit shorthand object properties - i.e.: `{a}` instead of `{a: a}`. The `ecma` option will only change the output in direct control of the beautifier. Non-compatible features in the abstract syntax tree will still be output as is. For example: an `ecma` setting of `5` will \*\*not\*\* convert ES6+ code to ES5. - `indent\_level` (default `4`) - `indent\_start` (default `0`) -- prefix all lines by that many spaces - `inline\_script` (default `true`) -- escape HTML comments and the slash in occurrences of `` in strings - `keep\_numbers` (default `false`) -- keep number literals as it was in original code (disables optimizations like converting `1000000` into `1e6`) - `keep\_quoted\_props` (default `false`) -- when turned on, prevents stripping quotes from property names in object literals. - `max\_line\_len` (default `false`) -- maximum line length (for minified code) - `preamble` (default `null`) -- when passed it must be a string and it will be prepended to the output literally. The source map will adjust for this text. Can be used to insert a comment containing licensing information, for example. - `quote\_keys` (default `false`) -- pass `true` to quote all keys in literal objects - `quote\_style` (default `0`) -- preferred quote style for strings (affects quoted property names and directives as well): - `0` -- prefers double quotes, switches to single quotes when there are more double quotes in the string itself. `0` is best for gzip size. - `1` -- always use single quotes - `2` -- always use double quotes - `3` -- always use the original quotes - `preserve\_annotations` -- (default `false`) -- Preserve [Terser annotations](#annotations) in the output. - `safari10` (default `false`) -- set this option to `true` to work around the [Safari 10/11 await bug](https://bugs.webkit.org/show\_bug.cgi?id=176685). See also: the `safari10` [mangle option](#mangle-options). - `semicolons` (default `true`) -- separate statements with semicolons. If you pass `false` then whenever possible we will use a newline instead of a semicolon, leading to more readable output of minified code (size before gzip could be smaller; size after gzip insignificantly larger). - `shebang` (default `true`) -- preserve shebang `#!` in preamble (bash scripts) - `webkit` (default `false`) -- enable workarounds for WebKit bugs. PhantomJS users should set this option to `true`. - `wrap\_iife` (default `false`) -- pass `true` to wrap immediately invoked function expressions. See [#640](https://github.com/mishoo/UglifyJS2/issues/640) for more details. - `wrap\_func\_args` (default `true`) -- pass `false` if you do not want to wrap function expressions that are passed as arguments, in parenthesis. See [OptimizeJS](https://github.com/nolanlawson/optimize-js) for more details. # Miscellaneous ### Keeping copyright notices or other comments You can pass `--comments` to retain certain comments in the output. By default it will keep JSDoc-style comments that contain "@preserve", "@license" or "@cc\_on" (conditional compilation for IE). You can pass `--comments all` to keep all the comments, or a valid JavaScript regexp to keep only comments that match this regexp. For example `--comments /^!/` will keep comments like `/\*! Copyright Notice \*/`. Note, however, that there might be situations where comments are lost. For example: ```javascript function f() { /\*\* @preserve Foo Bar \*/ function g() { // this function is never called } return something(); } ``` Even though it has "@preserve", the comment will be lost because the inner function `g` (which is the AST node to which the comment is attached to) is discarded by the compressor as not referenced. The safest comments where to place copyright information (or other info that needs to be kept in the output) are comments attached to toplevel nodes. ### The `unsafe` `compress` option It enables some transformations that \*might\* break code logic in certain contrived cases, but should be fine for most code. It assumes that standard built-in ECMAScript functions and classes have not been altered or replaced. You might want to try it on your own code; it should reduce the minified size. Some examples of the optimizations made when this option is enabled: - `new Array(1, 2, 3)` or `Array(1, 2, 3)` → `[ 1, 2, 3 ]` - `new Object()` → `{}` - `String(exp)` or `exp.toString()` → `"" + exp` - `new Object/RegExp/Function/Error/Array (...)` → we discard the `new` - `"foo bar".substr(4)` → `"bar"` ### Conditional compilation You can use the `--define` (`-d`) switch in order to declare global variables that Terser will assume to be constants (unless defined in scope). For example if you pass `--define DEBUG=false` then, coupled with dead code removal Terser will discard the following from the output: ```javascript if (DEBUG) { console.log("debug stuff"); } ``` You can specify nested constants in the form of `--define env.DEBUG=false`. Terser will warn about the condition being always false and about dropping unreachable code; for now there is no option to turn off only this specific warning, you can pass `warnings=false` to turn off \*all\* warnings. Another way of doing that is to declare your globals as constants in a separate file and include it into the build. For example you can have a `build/defines.js` file with the following: ```javascript var DEBUG = false; var PRODUCTION = true; // etc. ``` and build your code like this: terser build/defines.js js/foo.js js/bar.js... -c Terser will notice the constants and, since they cannot be altered, it will evaluate references to them to the value itself and drop unreachable code as usual. The build will contain the `const` declarations if you use them. If you are targeting < ES6 environments which does not support `const`, using `var` with `reduce\_vars` (enabled by default) should suffice. ### Conditional compilation API You can also use conditional compilation via the programmatic API. With the difference that the property name is `global\_defs` and is a compressor property: ```javascript var result = Terser.minify(fs.readFileSync("input.js", "utf8"), { compress: { dead\_code: true, global\_defs: { DEBUG: false } } }); ``` To replace an identifier with an arbitrary non-constant expression it is necessary to prefix the `global\_defs` key with `"@"` to instruct Terser to parse the value as an expression: ```javascript Terser.minify("alert('hello');", { compress: { global\_defs: { "@alert": "console.log" } } }).code; // returns: 'console.log("hello");' ``` Otherwise it would be replaced as string literal: ```javascript Terser.minify("alert('hello');", { compress: { global\_defs: { "alert": "console.log" } } }).code; // returns: '"console.log"("hello");' ``` ### Using native Terser AST with `minify()` ```javascript // example: parse only, produce native Terser AST var result = Terser.minify(code, { parse: {}, compress: false, mangle: false, output: { ast: true, code: false // optional - faster if false } }); // result.ast contains native Terser AST ``` ```javascript // example: accept native Terser AST input and then compress and mangle // to produce both code and native AST. var result = Terser.minify(ast, { compress: {}, mangle: {}, output: { ast: true, code: true // optional - faster if false } }); // result.ast contains native Terser AST // result.code contains the minified code in string form. ``` ### Annotations Annotations in Terser are a way to tell it to treat a certain function call differently. The following annotations are available: \* `/\*@\_\_INLINE\_\_\*/` - forces a function to be inlined somewhere. \* `/\*@\_\_NOINLINE\_\_\*/` - Makes sure the called function is not inlined into the call site. \* `/\*@\_\_PURE\_\_\*/` - Marks a function call as pure. That means, it can safely be dropped. You can use either a `@` sign at the start, or a `#`. Here are some examples on how to use them: ```javascript /\*@\_\_INLINE\_\_\*/ function\_always\_inlined\_here() /\*#\_\_NOINLINE\_\_\*/ function\_cant\_be\_inlined\_into\_here() const x = /\*#\_\_PURE\_\_\*/i\_am\_dropped\_if\_x\_is\_not\_used() ``` ### Working with Terser AST Traversal and transformation of the native AST can be performed through [`TreeWalker`](https://github.com/fabiosantoscode/terser/blob/master/lib/ast.js) and [`TreeTransformer`](https://github.com/fabiosantoscode/terser/blob/master/lib/transform.js) respectively. Largely compatible native AST examples can be found in the original UglifyJS documentation. See: [tree walker](http://lisperator.net/uglifyjs/walk) and [tree transform](http://lisperator.net/uglifyjs/transform). ### ESTree / SpiderMonkey AST Terser has its own abstract syntax tree format; for [practical reasons](http://lisperator.net/blog/uglifyjs-why-not-switching-to-spidermonkey-ast/) we can't easily change to using the SpiderMonkey AST internally. However, Terser now has a converter which can import a SpiderMonkey AST. For example [Acorn][acorn] is a super-fast parser that produces a SpiderMonkey AST. It has a small CLI utility that parses one file and dumps the AST in JSON on the standard output. To use Terser to mangle and compress that: acorn file.js | terser -p spidermonkey -m -c The `-p spidermonkey` option tells Terser that all input files are not JavaScript, but JS code described in SpiderMonkey AST in JSON. Therefore we don't use our own parser in this case, but just transform that AST into our internal AST. ### Use Acorn for parsing More for fun, I added the `-p acorn` option which will use Acorn to do all the parsing. If you pass this option, Terser will `require("acorn")`. Acorn is really fast (e.g. 250ms instead of 380ms on some 650K code), but converting the SpiderMonkey tree that Acorn produces takes another 150ms so in total it's a bit more than just using Terser's own parser. [acorn]: https://github.com/ternjs/acorn [sm-spec]: https://docs.google.com/document/d/1U1RGAehQwRypUTovF1KRlpiOFze0b-\_2gc6fAH0KY0k ### Terser Fast Minify Mode It's not well known, but whitespace removal and symbol mangling accounts for 95% of the size reduction in minified code for most JavaScript - not elaborate code transforms. One can simply disable `compress` to speed up Terser builds by 3 to 4 times. | d3.js | size | gzip size | time (s) | | --- | ---: | ---: | ---: | | original | 451,131 | 108,733 | - | | terser@3.7.5 mangle=false, compress=false | 316,600 | 85,245 | 0.82 | | terser@3.7.5 mangle=true, compress=false | 220,216 | 72,730 | 1.45 | | terser@3.7.5 mangle=true, compress=true | 212,046 | 70,954 | 5.87 | | babili@0.1.4 | 210,713 | 72,140 | 12.64 | | babel-minify@0.4.3 | 210,321 | 72,242 | 48.67 | | babel-minify@0.5.0-alpha.01eac1c3 | 210,421 | 72,238 | 14.17 | To enable fast minify mode from the CLI use: ``` terser file.js -m ``` To enable fast minify mode with the API use: ```js Terser.minify(code, { compress: false, mangle: true }); ``` #### Source maps and debugging Various `compress` transforms that simplify, rearrange, inline and remove code are known to have an adverse effect on debugging with source maps. This is expected as code is optimized and mappings are often simply not possible as some code no longer exists. For highest fidelity in source map debugging disable the `compress` option and just use `mangle`. ### Compiler assumptions To allow for better optimizations, the compiler makes various assumptions: - `.toString()` and `.valueOf()` don't have side effects, and for built-in objects they have not been overridden. - `undefined`, `NaN` and `Infinity` have not been externally redefined. - `arguments.callee`, `arguments.caller` and `Function.prototype.caller` are not used. - The code doesn't expect the contents of `Function.prototype.toString()` or `Error.prototype.stack` to be anything in particular. - Getting and setting properties on a plain object does not cause other side effects (using `.watch()` or `Proxy`). - Object properties can be added, removed and modified (not prevented with `Object.defineProperty()`, `Object.defineProperties()`, `Object.freeze()`, `Object.preventExtensions()` or `Object.seal()`). - `document.all` is not `== null` - Assigning properties to a class doesn't have side effects and does not throw. ### Build Tools and Adaptors using Terser https://www.npmjs.com/browse/depended/terser ### Replacing `uglify-es` with `terser` in a project using `yarn` A number of JS bundlers and uglify wrappers are still using buggy versions of `uglify-es` and have not yet upgraded to `terser`. If you are using `yarn` you can add the following alias to your project's `package.json` file: ```js "resolutions": { "uglify-es": "npm:terser" } ``` to use `terser` instead of `uglify-es` in all deeply nested dependencies without changing any code. Note: for this change to take effect you must run the following commands to remove the existing `yarn` lock file and reinstall all packages: ``` $ rm -rf node\_modules yarn.lock $ yarn ``` # Reporting issues In the terser CLI we use [source-map-support](https://npmjs.com/source-map-support) to produce good error stacks. In your own app, you're expected to enable source-map-support (read their docs) to have nice stack traces that will make good issues. # README.md Patrons: \*note\*: You can support this project on patreon: . Check out [PATRONS.md](https://github.com/terser/terser/blob/master/PATRONS.md) for our first-tier patrons. These are the second-tier patrons. Great thanks for your support! \* CKEditor ![](https://c10.patreonusercontent.com/3/eyJoIjoxMDAsInciOjEwMH0%3D/patreon-media/p/user/15452278/f8548dcf48d740619071e8d614459280/1?token-time=2145916800&token-hash=SIQ54PhIPHv3M7CVz9LxS8\_8v4sOw4H304HaXsXj8MM%3D) \* 38elements ![](https://c10.patreonusercontent.com/3/eyJ3IjoyMDB9/patreon-media/p/user/12501844/88e7fc5dd62d45c6a5626533bbd48cfb/1?token-time=2145916800&token-hash=c3AsQ5T0IQWic0zKxFHu-bGGQJkXQFvafvJ4bPerFR4%3D) ## Contributors ### Code Contributors This project exists thanks to all the people who contribute. [[Contribute](CONTRIBUTING.md)].  ### Financial Contributors Become a financial contributor and help us sustain our community. [[Contribute](https://opencollective.com/terser/contribute)] #### Individuals  #### Organizations Support this project with your organization. Your logo will show up here with a link to your website. [[Contribute](https://opencollective.com/terser/contribute)]          