

Inexpensive protection of your source code

Snob

Simple Name Obfuscator

Tutorial

Version 1.0



MacroExpressions
<http://www.macroexpressions.com>

Table of Contents

0.	STRUCTURE OF SNOB TUTORIAL	2
1.	INSTALLING AND UNINSTALLING SNOB.....	2
2.	WHAT EXACTLY SNOB DOES	2
3.	THE TEST PROJECT	3
4.	CONFIGURING SNOB FOR THE BARK PROJECT.....	5
4.1	PROTECTING FILES FROM OBFUSCATION. INTRODUCING APIFILES.SNOB	5
4.2	INFORMING SNOB OF A PROGRAMMING LANGUAGE: DOTEXT.SNOB	5
4.2.1	<i>Comments in dotext.snob.....</i>	6
4.2.2	<i>Reusing a configuration: include= statement.....</i>	6
4.2.3	<i>Using pre-packaged configurations: use= statement.....</i>	7
5.	RUNNING SNOB AND INSPECTING RESULTS.....	8
5.1	HOW TO RUN SNOB	8
5.2	SNOB OBFUSCATION MAP: PROJMAP.SNOB.....	9
6.	SYNTAX OF A LANGUAGE DEFINITION FILE.....	10
6.1	REGULAR EXPRESSIONS IN DOTEXT.SNOB.....	10
6.2	STATEMENTS OF LANGUAGE-SPECIFIC CONFIGURATION FILES	11
6.3	TELLING SNOB WHAT TO OBFUSCATE: NAME= STATEMENT	12
6.4	TELLING SNOB WHAT TO REMOVE: COMMENT= STATEMENT	12
6.5	TELLING SNOB WHAT NOT TO CONFUSE WITH NAMES: KEYWORD= STATEMENT 12	
6.6	TELLING SNOB WHERE NOT TO LOOK FOR NAMES: IGNORE= STATEMENT	13
6.6.1	<i>Example: Adding pragma handling to dotc.snob.....</i>	13
6.7	TELLING SNOB WHERE NOT TO LOOK FOR NAMES IN ANY LANGUAGE: RESERVED= STATEMENTS	16
6.8	INTRODUCING: STRING= STATEMENTS	16
7.	ADDING RESERVED WORDS TO SNOB CONFIGURATION.....	17
7.1	A METHOD OF PRESERVING THIRD-PARTY NAMES AUTOMATICALLY	18
7.2	PRESERVING LITERAL WORDS IN ALL LANGUAGES: CONFIGURATION FILE RESERVED.SNOB	19
8.	SUMMARY.....	21
8.1.1	<i>Files that make it to the target directory tree</i>	<i>21</i>
8.1.2	<i>Files that Snob will not attempt to obfuscate.....</i>	<i>21</i>
8.1.3	<i>Names that Snob will not attempt to obfuscate.....</i>	<i>21</i>
8.1.4	<i>Snob configuration files.....</i>	<i>21</i>
9.	CONCLUSION.....	22

0. Structure of Snob tutorial

This tutorial introduces the use and configuration of Snob, or Simple Name OBFuscator. Its objective is to replace meaningful names in your project with meaningless ones in an irreversible way and to remove comments. We introduce a toy project in C and demonstrate how to work with and configure Snob.

Snob by itself is independent of the programming language(s) used in your project. Instead, it relies on language-specific configuration files.

Correspondingly, we will show Snob usage in case the language configuration files are already available, at least in the most basic version. Then we'll explore the insides of the language configuration files

1. Installing and uninstalling Snob

Snob is a standalone executable, and a rather small one by today's standards. Just copy the Snob executable file, `snob.exe` to some directory, such as `C:\Snob` (as we will assume from now on), and that completes the installation. If you have basic configuration files, copy them to the same directory. If you feel like that, you may want to add `C:\Snob` to your PATH environment variable.

To uninstall Snob, simply remove the Snob directory `C:\Snob` or whatever the name you gave to it.

2. What exactly Snob does

Snob, we said, obfuscates names and removes comments. Snob, we said, is a tool independent of your project's programming language(s). Therefore, to do its job, Snob must be told:

- what a name is, and
- what a comment is

Given only those definitions, Snob is ready to do its job, but you will almost certainly not like the results. The reason is that Snob will replace not only your own identifiers, but also anything that looks like a name to it. However, your project's programming language may have elements lexically indistinguishable from names (such as keywords) or other language constructs (like C `pragmas`) that should not be touched at all. Therefore, to do its job properly, Snob must be told:

- where *not* to look for a name
- where to look for a name differently – and how
- what name-like looking text is not a name (such as `break` in C)

These configuration items taken together define a programming language to Snob.

Turning attention to your project, we observe that there may be

- the whole files that should not be obfuscated
- other names that should not be obfuscated

Files that you don't want obfuscated are, for instance, your API (application programmer's interface) files that you deliver to your customer, or any accompanying application examples.

Other names not to be obfuscated include, for instance, the API names of a third-party library you are using in your project, or any language extensions provided by your compiler.

To summarize, Snob configuration for your project consists of

- Language-specific configuration, largely independent of your particular project, and
- Project-specific configuration, largely independent of the languages used.

In this tutorial, we introduce a toy project, Bark, in the C programming language and follow the steps needed to obfuscate it using Snob.

3. The test project

Consider a toy project in the C language that we want to obfuscate because we deliver it in the source code format.

It contains a single function as its public interface, `bark ()`, which takes a pointer to a character string and prints it to the standard output, but prefixed with “BARK: ” and ending with three exclamation points. E.g., given “Hello, world!” it would print “BARK: Hello, world!!!\n”.

We implemented this function in two `.c` files and two headers, one (`bark.h`) for the public interface and one (`barkpriv.h`) as internal public header.

Here is our implementation:

bark.c:

```
/* Here is an implementation of Bark */

#include <stdio.h>
#include "bark.h"
#include "barkpriv.h"

void bark(const char *str)
{
    printf(BARK_PREFIX); //print prefix
    bark_internal(str);  //print the rest
}
```

barkpriv.c

```
/* Here is an implementation of Bark's internals */

#include <stdio.h>
#include "barkpriv.h"

unsigned int interval = 17u; //just for kicks

void bark_internal(const char *str)
{
    printf("%s", str); //need "%s" in case str contains formatting
    printf(BARK_SUFFIX); //print suffix
}
```

barkpriv.h

```
/* This is the private header of the terrific Bark package */

#define BARK_PREFIX "BARK: "
#define BARK_SUFFIX "!!!\n"

extern void bark_internal(const char *);
```

And, finally,

bark.h

```
/* This is a public API of the terrific Bark package */

extern void bark(const char *);
```

In addition we want to provide a self-test which serves also as an illustration to an application note:

barktest.c

```
/* This is a self-test and an example of the bark code */
/* Bark will output "BARK: ", then your string
   and then three exclamation points and a newline.
*/
#include <stdio.h>
#include <string.h>
#include "bark.h"

int main()
{
    char buf[200];
    printf(">"); //prompt
    while(NULL!=fgets(buf, sizeof(buf), stdin)) {
        size_t len = strlen(buf);
        if(len > 0) {
            if(buf[len-1] != '\n') {
                //We didn't get the whole string; try again
                printf(
                    "          (ERROR) String too long. Try again\n\n");
            }
            else {
                buf[--len] = 0; //truncate the newline
            }
        }
        if(len == 0) {
            printf("Bye\n");
            break;
        }
        bark(buf);
        fflush(stdin); //start clean;
        printf(">"); //print prompt
    }
}
```

```
    }  
    return 0;  
}
```

This is the project that is supplied in [Bark\Code](#) directory in the distribution. You can actually build it and play with barking output.

4. Configuring Snob for the Bark project

If you have not done so already, unzip all [.snob](#) files in the distribution to [C:\Snob](#) where [snob.exe](#) is as well. The [.snob](#) files are basic-level configuration files for the C language. And yes, for now we'll assume that the most basic configuration for your programming language exists.

4.1 Protecting files from obfuscation. Introducing [APIfiles.snob](#)

First of all, the files [bark.h](#) and [barktest.c](#) represent our API. We don't want to obfuscate them at all. The way to inform Snob about it is to list them in your project's [APIfiles.snob](#) file. So, let's create [APIfiles.snob](#) in [Bark\Code](#):

[APIfiles.snob](#)

[bark.h](#)
[barktest.c](#)

Any subdirectory in the project directory tree may have a file named [APIfiles.snob](#). Its syntax is as follows: each non-empty line is a file specification of file(s) considered your API. The filespec can contain wildcards ('*' and '?'); in this case all matching files are considered API.

Snob treats the API filespecs as follows:

- If the filespec does not contain any directory information, not even '.', this is treated as real, real API file in the same directory as the [APIfiles.snob](#) itself.
 - If the filespec contains wildcards, Snob would register any match as API but would not complain if it didn't find any.
 - If the filespec does not contain wildcards, i.e., it is just a filename, Snob would exit with an error information if the file didn't exist and Snob knows configuration for its extension.
 - If Snob doesn't know the extension configuration, it would ignore the filespec match.
- If the filespec does contain any directory information whatsoever, Snob would learn names from any filespec match with known extension configuration and mark them as preserved (i.e., not subject to obfuscation). The net effect of this is that the filespec matches with known extensions and found in the project directory tree would be stripped off any comments but all names in them would be preserved.

4.2 Informing Snob of a programming language: [dotext.snob](#)

Now we need to produce the extension-specific configuration files.

To decide how to obfuscate a file [filename.ext](#), Snob searches for a configuration file with a fixed name [dotext.snob](#) (so, for [.cpp](#) files Snob will search for [dotcpp.snob](#)).

Snob's rule of search is as follows: first look in the directory where the file, `filename.ext`, is located and then go up the directory tree all the way to the root of the project directory (such as `Bark\Code` above). The first configuration file found takes effect. (It may be said that subdirectories that do not have the configuration inherit it from their parent directory, and those which do have it override the inherited configuration, if any.) If none is found, Snob finally looks for it in its own directory (`C:\Snob`).

If the configuration is not found, the file, `filename.ext`, will not be processed into the target directory tree. For instance, we do not provide `dotc.snob`, so Snob configuration files, which have the extension `.snob`, are skipped.

First of all, since the project is in the C programming language, we need to create configuration for it. C files have, by convention, extension `.c`, so, by Snob convention, the name of the configuration file for it is `dotc.snob`.

C has something of an oddity in that it has “header files” which, being perfectly good C files, have, by convention, a different extension, `.h`. So, to create a configuration for C, we need a second configuration file, `doth.snob`.

4.2.1 Comments in `dotext.snob`

4.2.2 Reusing a configuration: `include=` statement.

Good news is that however we decide to configure `dotc.snob`, we should configure `doth.snob` the same way, simply because the header files have the same syntax. (Things get more complicated if we throw C++ into the language mix: `.h` files may have C or C++ syntax – or both. This matter is discussed in the manual and we skip it in the tutorial.) Instead of copying `dotc.snob` to `doth.snob` and thus creating ourselves a maintenance headache, we create the following

`doth.snob`

```
Same configuration as in dotc.snob
include=dotc.snob
```

The first line is a comment; any line starting with non-keyword is a comment and it is safe to start a comment with a blank, as we just did.

The second line contains `include=` keyword; it instructs Snob to read configuration from another file as if it were textually included. The filename to include is `dotc.snob`, the one that we are going to create next.

Snob has rules (covered in the manual) on where to search for the file to include:

- If no path information is specified, it searches first in the directory where the current file (`doth.snob`) itself is located and all the way up to the top level of the project directory. If still not found, the file is searched in the directory where the Snob executable, `snob.exe`, is located.
- If the path part is present and resolves to a relative path, it is considered relative to the directory where the current file is located
- If the path is not relative, Snob looks for the file exactly where specified
- If Snob cannot find the file, it reports an error and exits.

In our case, since no path information is specified, it searches first in the directory where `doth.snob` itself is located and all the way up to the top level of the project directory.

Since our project directory tree is quite simple – it contains just the project root directory `Code`, we put our `doth.snob` right there. If we had a few subdirectories, each would inherit the configuration from the parent directory, if present. However, any directory may have its own `doth.snob` which would override the inherited configuration.

We are done with `doth.snob`; let's concentrate on the `.c` configuration file, `dotc.snob`. We'll put it in the same directory `Code`.

4.2.3 Using pre-packaged configurations: `use=` statement.

We want to make use of a basic language configuration; we start with this:

```
dotc.snob  
use=C99base.snob
```

The `use=` statement is similar to the `include=` statement we've seen in `doth.snob`; the only difference is that Snob looks for the specified file only in its own directory (`C:\Snob`).

So, we included the base C configuration pre-packaged in `C99base.snob`. In the Snob directory, there is another similarly named file, `C90base.snob`, which is also a C configuration file but corresponding to the previous revision of the C standard. That revision didn't allow the `//`-comments we use in the Bark project, so we need the new standard. (Your compiler may be C90 and allow `//`-comments as a language extension. Snob knows none of this.)

The Bark project uses a few C standard library calls and macros; their names must not be obfuscated. (The same, by the way, applies to the standard `typedefs`, `struct`, `union` and `enum` tags and members.) As to the definitions of them, the Snob directory contains two files to choose from: `Crsvnormal.snob` and `CParanoia.snob`. The first file contains commonly used reserved words; the second one reserves anything claimed in the standard however mildly; for instance, it reserves names beginning with an underscore (`_`). Usually, you'll be OK with the first file, which we add now to our `dotc.snob`:

```
use=Crsvnormal.snob
```

We are ready to obfuscate our Bark project.

5. Running Snob and inspecting results

5.1 How to run Snob

Snob is a command-line utility that takes two arguments: your project directory and the name of the directory which will contain the obfuscated version of your project, e.g.,

```
snob MyProject MyObfuscatedProject
```

The target directory must not exist yet and its location must be writeable.

Snob will create the target directory and clone the directory tree of the project directory (with no files in the target tree yet). Then it will look at each file in the project tree, such as and examine its extension, `.ext` in this case.

The configuration files (`doth.snob`, `dotc.snob` and `APIfiles.snob`) are supplied in `Bark\Step1` directory. If you didn't work along, simply copy those files to `Bark\Code`. Now change to `Bark` directory and issue the following command:

```
C:\Snob\snob Code Obf1
```

Here is the Snob output in all its glory:

```
Looking for configuration files under "Code"
Entering directory "Code"
Searching configuration for the extension .snob
  No configuration file dotsnob.snob found
Searching configuration for the extension .c
  Found configuration dotc.snob in Code
Searching configuration for the extension .h
  Found configuration doth.snob in Code
Leaving directory "Code"
Done looking for configuration files
Looking for API specs under "Code"
Looking for configuration files under "Code"
Entering directory "Code"
  Marking bark.h copy-only
  Marking barktest.c copy-only
Leaving directory "Code"
Done looking for API specs
Processing project "Code" to "Obf1"
Entering directory "Code"
  APIfiles.snob -- skipping
  bark.c --> Obf1\bark.c (process)
  bark.h --> Obf1\bark.h (copy)
  barkpriv.c --> Obf1\barkpriv.c (process)
  barkpriv.h --> Obf1\barkpriv.h (process)
  barktest.c --> Obf1\barktest.c (copy)
  dotc.snob -- skipping
  doth.snob -- skipping
Leaving directory "Code"
Writing obfuscation map to Obf1\projmap.snob
End processing the project
```

Finished

We can see that Snob created the directory structure under `Obf1` identical to that of `Code`. Let's take a look at non-API files it created:

bark.c

```
#include <stdio.h>
#include "bark.h"
#include "barkpriv.h"

void bark(const char *C0000000B)
{
    printf(C0000000C);
    C0000000D(C0000000B);
}
```

barkpriv.c

```
#include <stdio.h>
#include "barkpriv.h"

unsigned int C0000000E = 17u;

void C0000000D(const char *C0000000B)
{
    printf("%s", C0000000B);
    printf(C0000000F);
}
```

barkpriv.h

```
#define C0000000C "BARK: "
#define C0000000F "!!!\n"

extern void C0000000D(const char *);
```

It seems quite clear that it is not for human eyes.

5.2 Snob obfuscation map: `projmap.snob`

It may be interesting to look at the obfuscation map which Snob saves in the file `projmap.snob` in the target directory:

projmap.snob

```
#Snob Substitution Table for the project "Code"
C0000000B : str
C0000000C : BARK_PREFIX
```

```
C0000000D : bark_internal
C0000000E : interval
C0000000F : BARK_SUFFIX
```

The first line is the title; following it is a list of pairs – a name that Snob invented vs. a name that was replaced by the invented name.

6. Syntax of a language definition file

To this point, we carefully avoided a question of the format of the files `Crsvnormal.snob` and `C99base.snob` that we used. Since those files were only good for inclusion in `dotc.snob`, we are actually going to talk about the syntax of `dotext.snob` files in general.

6.1 Regular expressions in `dotext.snob`

Let's begin with a motivational example. In C (and C++) there is a rather interesting feature, although rarely used: the stringize operator. Consider a macro definition

```
#define MYSTRING(x) #x
```

If you write then

```
char *p = MYSTRING(Because I can);
```

the macro expands to a double-quoted string after pre-processing, but Snob has no way of knowing that. So, Snob will treat the words “Because,” “I,” “can” as names and replace them with invented names, which is not what we want.

Clearly, Snob must be told to treat text in parentheses preceded by the word `MYSTRING` as a string, so as to not look for names there.

Unfortunately, this cannot be done in the common pre-packaged language configuration file like `C99base.snob`. That's because the name of the macro was invented within your project, and cannot be known in advance. So, we need to do this ourselves.

To re-iterate, we want to designate as a string the following definition: “text in parentheses preceded by the word `MYSTRING`.” This is a rather complex idea and Snob needs a rather expressive means to express such ideas.

In language-specific configuration files, Snob uses *regular expressions* to express complex configuration rules.

Regular expressions are, in a way, text search patterns. By telling Snob that a string is such and such regular expression, we mean (and Snob understands) that a segment of text matching a search by the regular expression is considered a string.

(There are several flavors of regular expressions; Snob uses PCRE, Perl-compatible regular expressions. The PCRE library, which is open source software, is written by Philip Hazel, and copyright the University of Cambridge, England. See

<ftp://ftp.csx.cam.ac.uk/pub/software/programming/pcre/>

A reference of Snob regular expressions, which is an adaptation of PCRE reference, is provided in the distribution and also online on the Snob pages of MacroExpressions website.)

A suitable regular expression for our verbal definition of a string is

```
\bMYSTRING\( . * ? \)
```

(In regular expressions, parentheses are used for grouping, just like in math expressions. To give them their literal meaning, they must be “escaped” with a backslash, as we have in the expression above. The dot means “match any character” and the “*” means repeated any number of times “ungreedy” – so that the first, rather than the very last right parenthesis will end the search. The starting \b requires that the match begins on the word boundary.)

It’s worth noting that the word `MYSTRING` itself will not be obfuscated because it is a part of the search pattern. If we want to obfuscate `MYSTRING`, we need to exclude it from the search pattern. To do so, we can say that a string is any parenthesized text if it follows the word `MYSTRING`. The following regular expression corresponds to this improved definition:

```
( ? <= \bMYSTRING ) \( . * ? \)
```

So, we have a perfectly good regular expression. Now, we need to tell Snob, in our `dotc.snob` file, that it defines a string. This is done via `string=` statement which will be covered in its turn.

6.2 Statements of language-specific configuration files

It is time to discuss general syntax of language-specific configuration files. Recall that for extension `.ext` the name of the configuration file is `dotext.snob` and that the configuration file acts on the directory it is located in and down the directory tree until overridden in some subdirectory by a file with the same name.

A `dotext.snob` file consists of statements and comments. A comment is anything that is not a statement. Since all statements start from the first position, it is safe to start comment lines with a blank space, as we were doing all along in the example.

The following statements are recognized by Snob:

- `name=<regex>`
- `keyword=<regex>`
- `ignore=<regex>`
- `reserved=<regex>`
- `string=<regex>`
- `string=<regex> name=<regex>`
- `comment=<regex>`
- `include=<filespec>`
- `use=<filespec>`

Here `<regex>` is a Perl-style (or, more precisely, PCRE-style) regular expression and `<filespec>` is a filename with optional path component. Notice that there are no spaces around the `=` sign.

We already encountered `include=` and `use=` statements. Let's look at the other statements from the list above.

6.3 Telling Snob what to obfuscate: `name=` statement

Since Snob is a name obfuscator, it needs to know what the names are. Since the notion of a name is language-dependent, Snob thinks a name is any match to a regular expression. For instance, if we inspect `C90base.snob use='ed` indirectly in our `dotc.snob`, we find the following text:

```
Names are made of underscores, letters and digits and
start with non-digit, starting on a word boundary.
name=\b[_A-Za-z][_0-9A-Za-z]*
```

That is, a name starts on a word boundary with a `_` or a letter (upper- or lowercase) followed by a sequence of underscores, letters and digits. (So, in `20042005UL`, `UL` is not a name since it doesn't start on a word boundary.) This “word boundary” stuff is not mentioned in C textbooks because C extracts a C language token and decides whether it is a name. Snob has no idea of C tokenization and therefore needs a self-contained definition.

Generally speaking, there may be more than one `name=` statement; different definitions would act as alternatives. This is also true for all other statements with regular expression arguments. In case of `name=`, though, you'll hardly ever need more than one definition.

6.4 Telling Snob what to remove: `comment=` statement

Since Snob promises to remove comments, it needs to know what the comments are. Even if Snob doesn't remove a comment (as in an API file), it should not confuse any text in a comment with a name. Comments are defined by a regular expression supplied via a `comment=` statement. Again, from `C90base.snob`:

```
Multi-line comment
comment=(?s)(/\*.*?\*/)
```

which means that a comment is any text between `/*` and the next `*/`, where the `(?s)` option allows the dot to match the end-of-line character. Note that single-line comments are not in `C90base.snob`: they made their official entry to the C world with C99. That's why we included `C99base.snob` in our example.

6.5 Telling Snob what not to confuse with names: `keyword=` statement

In many languages, C included, keywords are lexically similar to names: we just have to remember that, say, `switch` is a keyword. The `keyword=` statements tell Snob what words are keywords and therefore *not* names. Here is another quote from `C90base.snob`:

```
Language keywords (start and end on the word boundary)
keyword=\b(auto|break|case|char|continue)|default|double)\b
keyword=\b(float|for|goto|if|int)|long|register|turn)\b
keyword=\b(short|signed|sizeof|static|struct|switch)\b
keyword=\b(typedef|union|signed)|void|volatile|while)\b
keyword=\b(define|do|if|_n|_def|_elif|_include|_endif)\b
```

```
keyword=\b(e|lse|num|extern)|include|undef|pragma)\b
```

The right-hand sides of the `keyword=` statements are regular expressions covering some of the C language keywords. The representation syntax we chose here is a bit twisted (as the syntax of the regular expressions themselves); the last statement is as good as more readable

```
keyword=\b(else|enum|extern|include|undef|pragma)\b
```

but the original format is arguably slightly faster.

Taken together, our `keyword=` statements cover all C keywords. Of course, we could pool all keywords in a single statement (and that would make Snob run faster), but for presentation purposes we avoid excessively long lines.

Note that if a name cannot be confused with the language keyword (as in Perl, for instance), we do not need to spell out the actual keywords.

6.6 Telling Snob where not to look for names: `ignore=` statement

There may be constructs in the language other than the keywords that must be left intact. It would be entirely correct to use this regular expression in a `keyword=` statement. However, for purely aesthetic reasons, Snob has a different statement, `ignore=`, which just looks better with tricky regular expressions. The `ignore=` syntax is functionally equivalent to `keyword=`, the only difference being a different connotation: we tend to think of `keyword=` as of a finite collection of distinct words whereas `ignore=` is thought of as an arbitrarily complex regular expression. An example comes again from C90base.snob:

```
Ignore <>-includes
ignore=(?m)^[ \t]*\#[ \t]*include[ \t]*<.*?>
```

6.6.1 Example: Adding pragma handling to dotc.snob

Imagine that you need to use a compiler-specific `#pragma` statement in `bark.c`, like this (also in subdirectory `Step2`):

```
bark.c
/* Here is an implementation of Bark */

#include <stdio.h>
#include "bark.h"
#include "barkpriv.h"

#pragma optimize("atp", on)

void bark(const char *str)
{
    printf(BARK_PREFIX); //print prefix
    bark_internal(str);  //print the rest
```

```
}
```

If we run Snob:

C:\Snob\snob Code Obf2

and inspect the obfuscated version, we are up to frustration:

bark.c (obfuscated)

```
#include <stdio.h>
#include "bark.h"
#include "barkpriv.h"

#pragma C0000000B("atp", C0000000C)

void bark(const char *C0000000D)
{
    printf(C0000000E);
    C0000000F(C0000000D);
}
```

Snob messed up the `pragma` statement and it is no longer usable! It means that there is a bug in the configuration files that we used, and it needs to be corrected. We will correct it in `dotc.snob`.

6.6.1.1 Making our own `ignore=` statement

We want to let Snob know that `pragma` statements must be ignored, i.e., left untouched.

Let's invent a regular expression describing a `pragma` statement in C. Since the syntax of a `pragma` statement is entirely compiler-dependent, it looks like Snob needs to ignore everything from the `#pragma` statement to the end of the line:

```
#pragma.*$
```

That's a good start, but we need to make sure that:

- `pragma` starts at the beginning of a line (^ marker)
- therefore the subject text must be considered multi-line – (?m) option
- there are optional blank spaces before and after # - [\t] *
- and there is at least one blank after `pragma` – [\t]

Here is the candidate:

```
(?m)^[ \t]*#[ \t]*pragma[ \t].*$
```

This expression meets our verbal definition, but a closer look at the definition itself reveals a problem: If a comment starts after the `pragma` on the same line and continues on to the next line, this regular expression will cause Snob to skip beyond the beginning of the comment and thus to misunderstand – and partially obfuscate – the comment instead of removing it. Most likely, thus obfuscated code won't even compile.

A remedy is to include an assertion that there must be `/*`, `//` or end of line after the `pragma` statement ((?=/(* | /) | \$)). In addition, the dot matches the first, rather than the last occurrence (. * ? instead of . *). Here is our final version:

```
ignore=(?m)^[ \t]*#[ \t]*pragma[ \t].*?((?=/(\*|/))|$)
```

which we put in our `dotc.snob`.

You can now run Snob again and see that the `pragma` is now handled correctly, even if we add a wicked comment, like this:

```
bark.c
/* Here is an implementation of Bark */

#include <stdio.h>
#include "bark.h"
#include "barkpriv.h"

#pragma optimize("atp", on) /* Take
    this */
void bark(const char *str)
{
    printf(BARK_PREFIX); //print prefix
    bark_internal(str);  //print the rest
}
```

Here is the obfuscated version:

```
bark.c

#include <stdio.h>
#include "bark.h"
#include "barkpriv.h"

#pragma optimize("atp", on)
void bark(const char *C0000000B)
{
    printf(C0000000C);
    C0000000D(C0000000B);
}
```

Our fix worked! We can even put it in `C99base.snob`!

An inquisitive reader may observe, however, that our implementation is still flawed: A `#pragma` syntax may allow a string in a `pragma` statement (as it is in our example) and this string may conceivably contain a marker, say, `/*`, which is not a start of a comment because it is inside the string and therefore should not count. Not that it cannot be handled correctly, but it is certainly rather involved and beyond the scope of this tutorial and, quite frankly, not worth it.

Another problem is generic and has to do with line continuation marker in C (and equally C++) – a backslash followed by a newline. Line continuation can cut through a language token, like this:

```
#inc\
lude <st\
dio.h>
```


Snob has no idea of this line continuation syntax or properties, so this crazy stuff has to be captured in regular expressions. While this is theoretically possible, it would make the regular expressions monstrous and Snob very slow. Besides, line continuations cutting through names will almost certainly confuse Snob. However, since no-one codes in such fashion, the best solution is to be aware of the potential problem and ignore it.

6.7 Telling Snob where not to look for names in any language: `reserved=` statements

A variation on the theme of preserving certain names is the `reserved=` statement.

Its meaning is similar to the meaning of the `keyword=` statement but `reserved=` collects names cumulatively regardless of the subdirectory where the configuration file with `reserved=` statement is located, as opposed to overriding action of `ignore=` in subdirectories.

More importantly, for projects written in several languages, `ignore=` limits its action to files with the extension linked to the configuration file, but `reserved=` acts on files with any extension as long as a file with the linked extension exists. It's a bit fuzzy, so let's consider an example of C and, say, a FOO language. If your project is written in FOO, it may have a name, say, `strcat`. You can use it as you please. Now you add C files to your project, and there `strcat` has a special meaning of a standard library function. At this point, Snob has no idea whether `strcat` in a FOO file is FOO's own or it is a reference to C's `strcat`. On the grounds that it's better to be safe than sorry, Snob should assume the worst and preserve `strcat` if C files are present in the project. That's the mission of the `reserved=` command.

Here is a shortened example of a `reserved=` statement from `CParanoia.snob` from the `C:\Snob` directory:

```
Paranoia: reserved for the future and anything claimed by the
standard
reserved=\b(LC_|SIG)[A-Z]|(is|mem|str|to|wcs)[a-z]|_|[_0-9A-Za-z]*
.....
```

According to this definition, words starting with `LC_`, or with `SIG` followed by an uppercase letter, or with `is`, `mem`, `str`, `to` or `wcs` followed by a lowercase letter, or with an underscore, are reserved. If `CParanoia.snob` is included in `dotc.snob`, Snob won't obfuscate these names. This may appear too draconian a restriction, so we included in our `dotc.snob` another file, `Crsvnormal.snob`, which is compliant with the C90 ISO/ANSI standard.

6.8 Introducing: `string=` statements

One more place where Snob may confuse names with non-names is inside literal strings.

In some languages (like C or C++) a string is a string is a string: everything inside a string is taken literally. A definition of a string can be then added via the `ignore=` statement. For reasons which can also be called aesthetic, there is a synonym, `string=`. Here is a C example from `C90base.snob`:

String goes from one unescaped quote to the next
`string=(?<!\\\)\".*?(?<!\\\)\"`

While we are at it: we don't want Snob to touch literal characters either. For Snob, it doesn't matter that there may be only a single character in a C `'`-string:

A character is a string for our purposes (with single quotes)
`string=(?<!\\\)\'.*?(?<!\\\)\'`

Now we can also form a statement protecting C/C++ strings defined by the stringize operator as we discussed earlier:

`string=(?<=\bMYSTRING)\(. *?\)`

Since the name of the macro is defined within the project, this definition cannot go to a pre-packaged configuration file; its place is in `dotc.snob`.

In other languages, like UNIX shell script languages or Perl, a name is recognized within a string “literal” in certain circumstances (such as in a double-quoted string if preceded by an unescaped `$`). For those cases Snob recognizes an extended syntax of a `string=` statement: a definition is followed by a single space followed by a `name=` statement. The latter takes a regular expression defining what a name within a string might be. See a discussion on this in the manual. This form of the `string=` statement allows Snob to look for names in literal strings using a different (usually, a more restrictive) definition of a name. For instance, a statement

`string=(?<!\\\)\".*?(?<!\\\)\\" name=(?=\$)[_A-Za-z][_A-Za-z0-9]*`

recognizes “usual” names in a double-quoted string if they follow a dollar sign.

7. Adding reserved words to Snob configuration

If you are comfortable with regular expressions (and if not, consult the reference included), you are almost ready to configure Snob for a different programming language: you “simply” need to define the language elements via regular expressions as described earlier and fill out your `APIfiles.snob` file, if any.

The only missing part is reserved words. These come from two main sources:

- Non-standard language extensions provided by your compiler (such as `asm`, `interrupt` or `locate` in C)
- Third-party code (such as, for C++, Microsoft Foundation Classes, which are a nice addition to C++ standard compiler but are not a part of the language standard).

The first source has just several words; you can fish for them in your compiler documentation and add manually to `dotc.snob`. You can do so only once and at any time during development.

The second source is usually plentiful; adding the reserved words by hand may be an intimidating task. What's even worse, if you change your vendor of some widget implementation, you are likely to need to redo the third-party name collection.

7.1 A method of preserving third-party names automatically

Is there a way to automate this? In many cases, yes, there is. We'll explore the options on C, with our toy project in mind.

For the sake of example, let's treat standard names as third-party and remove the line

```
use= Crsvnormal.snob
```

from our `dotc.snob` file. Also, we need to remove `APIfiles.snob` temporarily, because `snobtest.c` has too many names preserved.

If we run Snob now, it would take the now “third-party” names and obfuscate it, as the following obfuscation map shows:

```
Obf4\projmap.snob
#Snob Substitution Table for the project "Code"
C00000000 : bark
C00000001 : str
C00000002 : printf
C00000003 : BARK_PREFIX
C00000004 : bark_internal
C00000005 : interval
C00000006 : BARK_SUFFIX
C00000007 : main
C00000008 : buf
C00000009 : NULL
C0000000A : fgets
C0000000B : stdin
C0000000C : size_t
C0000000D : len
C0000000E : strlen
C0000000F : fflush
```

Snob's excesses are shown in bold.

The first insight is that all those names appear in a header of the “third-party” code. What we can do immediately is to tell Snob to preserve *all* third-party names. The vehicle to do this is the `APIfiles.snob` file. It accepts wildcards in the filenames, so let's add the following lines to the now-empty `APIfiles.snob`:

```
C:\Program Files\CCompiler\INCLUDE\*.h
C:\Program Files\CCompiler\INCLUDE\SYS\*.h
```

(On your machine the path may be different.)

Since the files specified are located outside the project directory, they will not be copied to the target directory (and we don't want them to), but any names Snob finds in them will be learned and preserved (as we want them to).

If we run Snob now, we get a correct obfuscation map:

```
Obf5\projmap.snob
#Snob Substitution Table for the project "Code"
C0000000B : str
```

```

C0000000C : BARK_PREFIX
C0000000D : bark_internal
C0000000E : interval
C0000000F : BARK_SUFFIX

```

However, Snob runs very slowly while grinding all the files. It may be acceptable if your real third-party headers are small and few, but we still want to find a faster way.

7.2 Preserving literal words in all languages: configuration file `reserved.snob`

Snob recognizes a special filename, `reserved.snob`, as a configuration file. It has a simple syntax: each non-empty line is considered a word that must not be obfuscated. You can have a `reserved.snob` file in any subdirectory of the project directory tree; they all act cumulatively. Note that this configuration acts on files with any extension known to Snob: if there is a file `filename.ext` for which `dotext.snob` was found as described above, and a name in it is found in one of the `reserved.snob` files, it will not be obfuscated.

We are going to create `reserved.snob` which captures all the third-party names.

Here is a plan by the example. In the previous run of Snob, we made it inspect all third-party names by including

```

C:\Program Files\CCompiler\INCLUDE\*.h
C:\Program Files\CCompiler\INCLUDE\SYS\*.h
in APIfiles.snob.

```

We can as well make Snob inspect *only* those files. Indeed, let's imagine that the directory

`C:\Program Files\CCompiler\INCLUDE` is a project directory. We create a Snob project just by copying our bare configuration files there, `doth.snob` and `dotc.snob`. Having done that, let's run Snob on it:

```
C:\Snob\snob "C:\Program Files\CCompiler\INCLUDE" Obf6
```

The names Snob learned to preserve as API in the previous run are the names it obfuscated now. They are, in other words, in the latest obfuscation map, which turns out to be a large file:

Obf6\projmap.snob

```
#Snob Substitution Table for the project "C:\Program
Files\CCompiler\Include"
```

```

C00000000 : __ACCESS_CONTROL__
C00000001 : __midl
C00000002 : __cplusplus
.....
C00011A0E : get_lowsrc
C00011A0F : put_vrml
C00011A10 : get_vrml
C00011A11 : put_dynsrc
C00011A12 : get_dynsrc
.....
C00025EC3 : utimbuf
C00025EC4 : _ftime
C00025EC5 : utime
.....

```

All the names in this map are the third-party names that we want preserved. Let's extract them by removing the first line and from all other lines, everything before the name (there are many ways to do so). Let's save the resulting file in our Code directory as `reserved.snob`:

reserved.snob

```
__ACCESS_CONTROL__
__midl
__cplusplus
.....
get_lowsrc
put_vrml
get_vrml
put_dynsrc
get_dynsrc
.....
utimbuf
_futime
utime
.....
```

Recall that `reserved.snob` contains a list of names which Snob does not obfuscate, regardless of the filename extension of a file being processed and, unlike the `reserved=` statement, regardless of the presence of any particular language in the project.

Let's run Snob on our Bark project again,

C:\Snob\snob Code Obf6

Snob runs faster and we get a different but correct obfuscation map:

Obf6\projmap.snob

```
#Snob Substitution Table for the project "Code"
C000265F8 : bark
C000265F9 : BARK_PREFIX
C000265FA : bark_internal
C000265FB : interval
C000265FC : BARK_SUFFIX
```

(Your numbers may be different, depending on the compiler.) The numbers that appear in the obfuscated names are different from the previous Snob runs because they depend on the order in which names are encountered, and the addition of `reserved.snob` changed the order.

This technique of creating `reserved.snob` from a fake project containing the header files can be somewhat improved: In many cases, you don't use all the headers you have in a single project. If you can estimate what headers you might use, copy them to a separate directory and make a fake Snob project there. This will significantly reduce the size of the derived `reserved.snob` and make Snob run faster on your real project.

8. Summary

8.1.1 Files that make it to the target directory tree

A file with a name `filename.ext` in a project directory tree will have a version in the obfuscated directory tree if:

- the file `dotext.snob` exists in the same directory, or
- the file `dotext.snob` exists somewhere up from that directory in the project directory tree, or
- the file `dotext.snob` exists in the Snob home directory

Definitions:

The first `dotext.snob` found this way is called active configuration for `filename.ext`.

A file `dotext.snob` is called active if it is the active configuration for some file in the project directory tree.

8.1.2 Files that Snob will not attempt to obfuscate

Snob will not attempt to obfuscate a file `filename.ext` if the file `APIfiles.snob` exists in the same directory and mentions `filename.ext` either directly or as a wildcard match, provided that the mention contains no directory information.

8.1.3 Names that Snob will not attempt to obfuscate

A name in the file `filename.ext` will *not* be obfuscated if:

- It is listed in a `reserved.snob` file anywhere in the project directory tree, or
- It is found inside a match to the regular expression of a `reserved=` statement in some active `dotfoo.snob`, or
- It is inside a match to the regular expression of a `keyword=` or `ignore=` statement in the active configuration file for `filename.ext`, or
- It is inside a comment as defined by the active configuration file for `filename.ext`, or
- It is inside a match to the regular expression of a `string=` statement in the active configuration file for `filename.ext`, but does not match the `name=` regular expression in the `string=` statement.

8.1.4 Snob configuration files

Filename	Language-specific	Syntax	Keywords
<code>APIfiles.snob</code>	No	One filespec per line; wildcards allowed	-
<code>reserved.snob</code>	No	One word per line	-
<code>dotext.snob</code>	Yes	<code><keyword>=<regex></code>	<code>use, include, string, name, ignore, keyword, comment, reserved</code>

9. Conclusion

If you reached this point, you got a fair exposure to Snob, a simple and efficient name obfuscator. It is very flexible and independent of your project's programming languages.

As we have seen, configuring Snob is quite simple if basic configuration for your programming languages already exists. For those languages that do not have Snob configuration yet, we have shown how to configure Snob from scratch using the syntax of language-specific configuration files.

If you need to configure Snob for a new language, you should be unafraid of regular expressions. Writing correct regular expressions to define your programming language requires attention and some effort. It's best to start with a toy project exercising most idioms of your language and test your configuration carefully before attempting on a larger project. Also, check once in a while MacroExpressions website to see if there is a contributed configuration for your programming language. And if you would like to contribute your (tested!) configuration, please, email to snob@macroexpressions.com.