C meta-programming for the masses with C%: cmod

BY SIRIO BOLAÑOS PUCHET

seirios@member.fsf.org

FOSDEM'22
• C% is an experimental meta-programming language.
- C% is an experimental meta-programming language.
  - Spelled “C mod”, meaning “C with mods”.
What is C%: cmod

- C% is an experimental meta-programming language.
  - Spelled “C mod”, meaning “C with mods”.
  - Supports both C-specific and generic meta-programming.
What is C%: cmod

• C% is an experimental meta-programming language.
  ◦ Spelled “C mod”, meaning “C with mods”.
  ◦ Supports both C-specific and generic meta-programming.
  ◦ Context-dependent syntax with statement-like and function-like keywords.
C% is an experimental meta-programming language.

- Spelled “C mod”, meaning “C with mods”.
- Supports both C-specific and generic meta-programming.
- Context-dependent syntax with statement-like and function-like keywords.
What is C%: cmod

- **C%** is an experimental meta-programming language.
  - Spelled “C mod”, meaning “C with mods”.
  - Supports both C-specific and generic meta-programming.
  - Context-dependent syntax with statement-like and function-like keywords.
- **cmod** is an interpreter / pre-processor for C%. 
• **C%** is an experimental meta-programming language.
  - Spelled “C mod”, meaning “C with mods”.
  - Supports both C-specific and generic meta-programming.
  - Context-dependent syntax with statement-like and function-like keywords.

• **cmod** is an interpreter / pre-processor for C%.
  - Written in C99 and C%, employs a Flex/Bison parser.
What is C%: cmod

• C% is an experimental meta-programming language.
  ◦ Spelled “C mod”, meaning “C with mods”.
  ◦ Supports both C-specific and generic meta-programming.
  ◦ Context-dependent syntax with statement-like and function-like keywords.

• cmod is an interpreter / pre-processor for C%.
  ◦ Written in C99 and C%, employs a Flex/Bison parser.
  ◦ Released under the GPLv3, runs under POSIX.
What is C%: cmod

• C% is an experimental meta-programming language.
  ◦ Spelled “C mod”, meaning “C with mods”.
  ◦ Supports both C-specific and generic meta-programming.
  ◦ Context-dependent syntax with statement-like and function-like keywords.

• cmod is an interpreter / pre-processor for C%.
  ◦ Written in C99 and C%, employs a Flex/Bison parser.
  ◦ Released under the GPLv3, runs under POSIX.
  ◦ 3+ years in development.
%comment This is a simple example
%snippet print_greet (who) %{
  puts("Hello " $S{who} ")
};
}%
%recall print_greet (`World')(`FOSDEM')(`C%`)
puts("Hello " "World" ")
puts("Hello " "FOSDEM" ")
puts("Hello " "C%" ")
Example №2: map snippet to table

```plaintext
%snippet print_greet:v2 (who,func,preargs,postargs) %{
    ${func}(${preargs}”Hello " $S{who} "!"${postargs});%
%
}%
%
%# static table with tab-separated values %#   /* C comment */
%table who (name,func,preargs,postargs) %{
    | World | puts | %nul | %nul |
    | FOSDEM | fprintf | fp, | %nul |
    | C% | fputs | %nul | ,fp |
%
}%
%map who print_greet:v2
```
/* C comment */
%recall `print_greet:v2` (%<< World >>=%,%<< puts >>=%,%<< >>=%,%<< >>=%)
%recall `print_greet:v2` (%<< FOSDEM >>=%,%<< fprintf >>=%,%<< fp, >>=%,%<< >>=%)
%recall `print_greet:v2` (%<< C% >>=%,%<< fputs >>=%,%<< >>=%,%<< ,fp >>=%)

Example №2: map snippet to table
Example №2: map snippet to table

/* C comment */
puts("Hello " "World" "!");
fprintf(fp,"Hello " "FOSDEM" "!");
fputs("Hello " "C%" "!",fp);
1. Input file is parsed and each C% keyword gets evaluated eagerly.
How cmod works

1. Input file is parsed and each C% keyword gets evaluated eagerly.
2. Output goes into a temporary file, which becomes the next input file.
How cmod works

1. Input file is parsed and each C% keyword gets evaluated eagerly.
2. Output goes into a temporary file, which becomes the next input file.
3. Parsing loop proceeds until there are no more C% keywords in the input or evaluation limit is reached (configurable).
How cmod works

1. Input file is parsed and each C% keyword gets evaluated eagerly.
2. Output goes into a temporary file, which becomes the next input file.
3. Parsing loop proceeds until there are no more C% keywords in the input or evaluation limit is reached (configurable).

- Any non-C% code is passed-through verbatim.
How cmod works

1. Input file is parsed and each C% keyword gets evaluated eagerly.
2. Output goes into a temporary file, which becomes the next input file.
3. Parsing loop proceeds until there are no more C% keywords in the input or evaluation limit is reached (configurable).
   - Any non-C% code is passed-through verbatim.
   - Valid UTF-8 text is passed-through verbatim (8-bit scanner).
How cmod works

1. Input file is parsed and each C% keyword gets evaluated eagerly.
2. Output goes into a temporary file, which becomes the next input file.
3. Parsing loop proceeds until there are no more C% keywords in the input or evaluation limit is reached (configurable).

- Any non-C% code is passed-through verbatim.
- Valid UTF-8 text is passed-through verbatim (8-bit scanner).
- Parsing is sensitive to spacing in some places (e.g. snippets).
How cmd works

1. Input file is parsed and each C% keyword gets evaluated eagerly.
2. Output goes into a temporary file, which becomes the next input file.
3. Parsing loop proceeds until there are no more C% keywords in the input or evaluation limit is reached (configurable).

- Any non-C% code is passed-through verbatim.
- Valid UTF-8 text is passed-through verbatim (8-bit scanner).
- Parsing is sensitive to spacing in some places (e.g. snippets).
- Individual parsing passes can be inspected for debugging.
Example №3: map complex lambda to table

%table `nice folks` (greet, name, func, preargs, postargs) %{
  Hello    World    puts    %nul    %nul
  Howdy    FOSDEM   fprintf fp    %nul
  Hi        C%       fputs    %nul    fp
%
%map [sort=1] `nice folks` %{
  ${func}(%strcmp($b{preargs},`````,${preargs},`))
    $S{greet} " " $S{name} "!"
    %strcmp($b{postargs},`````,`, ${postargs}`));
%
}
Example №3: map complex lambda to table

```c
fputs(%strcmp(``, `, `, `, `, `, `)"Hi" " " ""C%" "!"%strcmp(`fp`, `, `, `, `, `, fp`));
fprintf(%strcmp(`fp`, `, `, `, `, `, `, fp, `)"Howdy" " " "FOSDEM" "!"%strcmp(``, `, `, `, `, `, `, `, `));
puts(%strcmp(``, `, `, `, `, `, `, `)"Hello" " " "World" "!"%strcmp(``, `, `, `, `, `, `, `, `));
```
Example №3: map complex lambda to table

fputs("Hi " "C% "!", fp);
fprintf(fp, "Howdy " "FOSDEM "!");
puts("Hello " "World "!");
Example №4: pipe to python and process output

```plaintext
%table-json who:v3 (greet,name) %{
[["Hello", "World"], ["Howdy", "FOSDEM"], ["Hi", "C%"]]
}%

strgsub (`puts`,`printf`,%<%
pipe [env=`func=puts`] `python3` %{
from os import getenv
f = getenv("func")
greet = [ %map who:v3 %{ $S{greet}, %} ]
who = [ %map who:v3 %{ $S{name}, %} ]
for g, w in zip(greet, who):
    print(' {}'.format(f, g, w));
}>>%)
```
Example №4: pipe to python and process output

```python
%delay(1) strgsub(`puts`, `printf`, %<<
%pipe [env=`func=puts`] `python3` %{
    from os import getenv
    f = getenv("func")
    greet = [ "Hello", "Howdy", "Hi", ]
    who = [ "World", "FOSDEM", "C%", ]
    for g, w in zip(greet, who):
        print('{} {} {} {} !"'.format(f, g, w));
}>>%)
```
Example №4: pipe to python and process output

`%strgsub(~`puts~, ~`printf~, ~%<<
    puts("Hello " "World" "!");
    puts("Howdy " "FOSDEM" "!");
    puts("Hi " "C%" "!");
>>%`
Example №4: pipe to python and process output

printf("Hello " "World" "!");
printf("Howdy " "FOSDEM" "!");
printf("Hi " "C%" "!");
%include Evaluate contents of another file in search path.
%once Define an include/repeat guard.
%snippet (%*) Define a parameterized verbatim code snippet.
%recall (%|) Insert evaluated code snippet.
%pipe (%!) Run command and capture output.
%table or %table-json Define static data table in TSV or JSON format.
%map Map snippet or lambda to data table.
%delay (%@) Delay evaluation for a number of parsing passes.
%defined Print text conditionally on resource being defined.
%strcmp Print text conditionally on string comparison.
%comment (%//) or %# Comment until end-of-line or block comment.
%table-stack Create new table by stacking other tables.
%intop Perform arithmetic operation with integers.
%strstr Check substring presence.
%strlen Compute string length.
%strgsub Replace all occurrences of search pattern.
%strsubcat Replace single pattern match or append at end.
%table-nrow Get number of rows in table.
%table-maxlen Compute maximum string length in table column.
%table-find Find row index of matching value in row column.
Example №5: define C struct and helpers

```c
%table keyval (type,name,init,dup,free) %{
    char* key NULL ${y} = strdup(${x}); free(${x});
    double value 0.0 ${y} = ${x};          %nul
%
}

struct keyval {
%map keyval %{
    ${type} ${name};
%}
};

struct keyval keyval_new(void) {
    return (struct keyval){
        %map keyval %{
            %map keyval %{
                .${name} = ${init},
            %}
        %}
    };                                        
}
```
Example №5: define C struct and helpers

```c
struct keyval keyval_dup(const struct keyval x) {
    struct keyval y;
    %map keyval {%
        %snippet [redef] keyval:dup (x,y) %{ ${dup} %}
        %recall keyval:dup (`x.${name}`, `y.${name}`)
    %}
    return y;
}

struct keyval keyval_free(struct keyval x) {
    %map keyval {%
        %snippet [redef] keyval:free (x) %{ ${free} %}
        %recall keyval:free (`x.${name}`)
        x.${name} = ${init};
    %}
    return x;
}
```
struct keyval {
    char* key;
    double value;
};

struct keyval keyval_new(void) {
    return (struct keyval){
        .key = NULL,
        .value = 0.0,
    };
}

struct keyval keyval_dup(const struct keyval x) {
    struct keyval y;
    y.key = strdup(x.key); y.value = x.value; return y;
}

struct keyval keyval_free(struct keyval x) {
    free(x.key); x.key = NULL;
    x.value = 0.0;
    return x;
}
/ In library header file grid3d.hm
%prefix g3d;
%proto [named] struct grid3d* alloc(enum g3d_type type, // data type
    size_t nx,       // x dimension
    size_t ny,       // y dimension
    size_t nz,       // z dimension
    bool alloc       // allocate memory?
    );

// In library code file grid3d.cm
%include "grid3d.hm"
%def [named] alloc {
    /* do stuff */
}

// In user code file main.c
struct grid3d *x = g3d_alloc(.nx=100, .ny=100, .nz=100,
    .type=G3D_FLOAT32, .alloc=true);
Example №6: define C function with named arguments

```c
// In library code file grid3d.c
struct g3d_alloc__args {
    enum g3d_type type;
    size_t nx;
    size_t ny;
    size_t nz;
    bool alloc;
};
struct grid3d * _g3d Alloc ( const struct g3d_alloc__args argv ) ;
#define g3d_alloc(...) _g3d_alloc((__VA_ARGS__ ) )

struct grid3d * _g3d Alloc ( const struct g3d_alloc__args argv ) {
    /* do stuff */
}

// In user code file main.c
struct grid3d * x = g3d_alloc(.nx=100, .ny=100, .nz=100, .type=G3D_FLOAT32, .alloc=true);
```
C-specific C% keywords

%typedef Define a type, including function types and named arguments.
%proto Define a function prototype*, with function type or named arguments.
%def Define a function with known function type or prototype.
%enum Define enum from table, with optional helper functions.
%foreach Iterate over array of known size.
%switch Switch cases over non-integer variable* (array, string, or struct).
%prefix Set prefix for functions and enums.
%unused Silence unused variable warning: (void)variable;.
%free Free and clear pointer: { free(ptr); ptr = NULL; }.
%arrlen Get length of static array: (sizeof(array)/sizeof(*(array))).

*cmod has a built-in partial C parser to handle declarators and compound initializers.
C% standard library

- Written in C% itself (using tables, snippets, etc.)
C% standard library

- Written in C% itself (using tables, snippets, etc.)
- Provides convenience in performing common tasks, defining data types, etc.
• Written in C% itself (using tables, snippets, etc.)
• Provides convenience in performing common tasks, defining data types, etc.
• Use is entirely optional.
C% standard library

- Written in C% itself (using tables, snippets, etc.)
- Provides convenience in performing common tasks, defining data types, etc.
- Use is entirely optional.

  - `autoarr` Definition of auto-growing array types
  - `common` Snippets for common, simple tasks.
  - `getopt` Automated parsing of CLI options.
  - `logging` Logging macros.
  - `ralloc` Retrying memory allocation functions.
  - `retval` Standardized propagating return values.
  - `variant` Definition of tagged unions.
I love programming in C (C99 to be precise), but coding in it can get tedious.
• I love programming in C (C99 to be precise), but coding in it can get tedious.
• The simplicity of C means the burden is on the programmer, but also the power.
Why C%

• I love programming in C (C99 to be precise), but coding in it can get tedious.
• The simplicity of C means the burden is on the programmer, but also the power.
• C% is an attempt to make the C programmer's life easier and more fun!
Why C%

• I love programming in C (C99 to be precise), but coding in it can get tedious.
• The simplicity of C means the burden is on the programmer, but also the power.
• C% is an attempt to make the C programmer's life easier and more fun!

Pros

+ Meta-programming opens up a whole new universe of possibilities!
Why C%

- I love programming in C (C99 to be precise), but coding in it can get tedious.
- The simplicity of C means the burden is on the programmer, but also the power.
- C% is an attempt to make the C programmer's life easier and more fun!

Pros

+ Meta-programming opens up a whole new universe of possibilities!
+ Generated code is inspectable and checked by the compiler, it's still C!
Why C%

• I love programming in C (C99 to be precise), but coding in it can get tedious.
• The simplicity of C means the burden is on the programmer, but also the power.
• C% is an attempt to make the C programmer's life easier and more fun!

Pros

+ Meta-programming opens up a whole new universe of possibilities!
+ Generated code is inspectable and checked by the compiler, it's still C!

Cons

- Additional step in compilation pipeline (although it's fast).
Why C%

• I love programming in C (C99 to be precise), but coding in it can get tedious.
• The simplicity of C means the burden is on the programmer, but also the power.
• C% is an attempt to make the C programmer's life easier and more fun!

Pros

+ Meta-programming opens up a whole new universe of possibilities!
+ Generated code is inspectable and checked by the compiler, it's still C!

Cons

- Additional step in compilation pipeline (although it's fast).
- Additional source of bugs (although it can help reduce them).
Reusability. Avoid code duplication.
Reusability. Avoid code duplication.

Consistency. Use same data across different locations.
C%'s aims

**Reusability.** Avoid code duplication.

**Consistency.** Use same data across different locations.

**Efficiency.** Perform common tasks quick and easy.
C%'s aims

**Reusability.** Avoid code duplication.

**Consistency.** Use same data across different locations.

**Efficiency.** Perform common tasks quick and easy.

**Concise.** Write and work with concise code.
C%'s aims

**Reusability.** Avoid code duplication.

**Consistency.** Use same data across different locations.

**Efficiency.** Perform common tasks quick and easy.

**Concision.** Write and work with concise code.

**Expressivity.** Better express the intent of code.
Reusability. Avoid code duplication.

Consistency. Use same data across different locations.

Efficiency. Perform common tasks quick and easy.

Concision. Write and work with concise code.

Expressivity. Better express the intent of code.

Transparency. Hide nothing from the programmer.
Reusability. Avoid code duplication.
Consistency. Use same data across different locations.
Efficiency. Perform common tasks quick and easy.
Concision. Write and work with concise code.
Expressivity. Better express the intent of code.
Transparency. Hide nothing from the programmer.
Abstraction. Handle similar things in a uniform manner.
C%'s aims

**Reusability.** Avoid code duplication.

**Consistency.** Use same data across different locations.

**Efficiency.** Perform common tasks quick and easy.

**Concision.** Write and work with concise code.

**Expressivity.** Better express the intent of code.

**Transparency.** Hide nothing from the programmer.

**Abstraction.** Handle similar things in a uniform manner.

**Extensibility.** Easily add new functionality.
C%’s aims

- **Reusability.** Avoid code duplication.
- **Consistency.** Use same data across different locations.
- **Efficiency.** Perform common tasks quick and easy.
- **Concision.** Write and work with concise code.
- **Expressivity.** Better express the intent of code.
- **Transparency.** Hide nothing from the programmer.
- **Abstraction.** Handle similar things in a uniform manner.
- **Extensibility.** Easily add new functionality.
- **Simplicity.** Keep the language simple but powerful.
C%'s aims

**Reusability.** Avoid code duplication.

**Consistency.** Use same data across different locations.

**Efficiency.** Perform common tasks quick and easy.

**Concision.** Write and work with concise code.

**Expressivity.** Better express the intent of code.

**Transparency.** Hide nothing from the programmer.

**Abstraction.** Handle similar things in a uniform manner.

**Extensibility.** Easily add new functionality.

**Simplicity.** Keep the language simple but powerful.

*Trust the programmer and don't prevent the programmer from doing what needs to be done!*
Thank you!
Thank you!

If you like this project, please contribute or donate crypto, but most of all, have fun!

For more information, please visit the project repo:
https://gitlab.com/seirios/cmod
Thank you!

If you like this project, please contribute or donate crypto, but most of all, have fun!

For more information, please visit the project repo:
https://gitlab.com/seirios/cmod

This document has been written using GNU TeXMACS; see www.texmacs.org.