CNoEvil

James Milne

Contents

CNoEvil v3.0.0

Using hash version: 7a2cc152cce77d

CNoEvil abuses the hell out of the C pre-processor, and other C language features, to create a language that is still technically C, but looks and behaves differently, whilst remaining fully compatible with C.

Expected Behaviour

Some definitions can produce warnings. Hide these by defining `EVIL_NO_WARN` before including `evil.h`.

Some definitions can produce errors. There is no option to hide these.

Definitions are expected to be created before the `evil.h` file is included.

e.g.

```
#define EVIL_HASH

#include "evil.h"
```

Libraries Available by Default:

- Evil_Bool (Exclude by defining `EVIL_NO_BOOL` before including evil.h)
- Evil_Comment (Exclude by defining `EVIL_NO_COMMENT` before including evil.h)
- Evil_Flow (Exclude by defining `EVIL_NO_FLOW` before including evil.h)
- Evil_Int (Exclude by defining `EVIL_NO_INT` before including evil.h)
- Evil_IO (Exclude by defining `EVIL_NO_IO` before including evil.h)
- Evil_Klass (Exclude by defining `EVIL_NO_KLASS` before including evil.h)
- Evil_Main (Exclude by defining `EVIL_NO_MAIN` before including evil.h)
- Evil_Proc (Exclude by defining `EVIL_NO_PROC` before including evil.h)
- Evil_Specifier (Exclude by defining `EVIL_NO_SPECIFIER` before including evil.h)
- Evil_Structures (Exclude by defining `EVIL_NO_STRUCT` before including evil.h)

Other Libraries Available:

- Evil Assert (Import be defining `EVIL_ASSERT` before including evil.h)
- Evil Bit (Import by defining `EVIL_BIT` before including evil.h)
- Evil Cli (Import be defining `EVIL_CLI` before including evil.h)
- Evil Coroutine (Import be defining `EVIL_COROUTINE` before including evil.h)
- Evil Encode (Import by defining `EVIL_ENCODE` before including evil.h)
- Evil Hash (Import be defining `EVIL_HASH` before including evil.h)
- Evil Help (Import be defining `EVIL_HELP` before including evil.h)
- Evil Lambda (Import be defining `EVIL_LAMBDA` before including evil.h)
- Evil List (Import be defining `EVIL_LIST` before including evil.h)
- Evil Log (Import be defining `EVIL_LOG` before including evil.h)
- Evil Malloc (Import be defining `EVIL_MALLOC` before including evil.h)
- Evil Math (Import be defining `EVIL_MATH` before including evil.h)
- Evil Random (Import be defining `EVIL_RANDOM` before including evil.h)

Libraries:

Evil Bool

Exclude by defining `EVIL_NO_BOOL` before including evil.h

This library ensures that the identifies `true`, `false` and `bool` are defined.

It can be thought of as an equivalent to stdbool.h

Evil Comment

Exclude by defining `EVIL_NO_COMMENT` before including evil.h

Allows you to use the `comment(...)` syntax for creating comments.

Example

```
comment(1 + 2 = 3);
```

`comment` can take any valid identifier. You may want to use strings normally.

Evil Flow

Exclude by defining `EVIL_NO_FLOW` before including evil.h

Defines the keywords for most of CNoEvil's syntax.

- then - A keyword, used to follow some constructs (such as If, While, etc.)
- end - A keyword, used to close functions, and some other constructs.
- If - A keyword, used to replace C's bracket'd `if`. i.e. Equivalent to `if(`.
- For - A keyword, used to replace C's bracket'd `for`. i.e. Equivalent to `for(`.
- While - A keyword, used to replace C's bracket'd `while`. i.e. Equivalent to `while(`.
- Do - A keyword, replaces C's `do`, and opens the block automatically.
- Switch(T) - A macro, creates and opens the block of a switch statement.

Example

```
If 1 + 2 == 3 then
    return 1;
Else
    return 0;
end
```

Evil Int

Exclude by defining `EVIL_NO_INT` before including evil.h

Defines up to two type specifiers:

- Number (if `int64_t` is supported). Equivalent to `int64_t`.
- Decimal. Equivalent to `long double`.

Defines up to two macros:

- MaxNumber (if `int64_t` is supported). Equivalent to `INT64_MAX`.
- MaxDecimal. Equivalent to `LDBL_MAX`.

Example

```
Decimal(y);
Number(x) = 12;
```

Evil IO

Exclude by defining `EVIL_NO_IO` before including evil.h

display(T) - Prints a representation of the given value to stdout.

displayf(F, T) - Prints a representation of the given value to F, a `FILE*`.

displayln(T) - Prints a representation of the given value to stdout, followed by a system-compatible line ending.

displayfln(F, T) - Prints a representation of the given value to F, a `FILE*`, followed by a system-compatible line ending.

endl - A keyword. Prints a system-comaptible line ending to stdout.

endlf(F) - Prints a system-comaptible line ending to F, a `FILE*`.

repr_type(T) - Returns a `char*` containing a text representation of the type. Optimisation may effect results. Returns "Unknown" for any type that cannot be accounted for.

Example

```
displayln("Hello, World!");
displayln(213);
displayln(repr_type("Hello, World!"));
```

Evil Main

Exclude by defining `EVIL_NO_MAIN` before including evil.h

- Main - A keyword, expected to be followed by keyword `then`. Automatically makes argc and argv available. Use instead of the `main` function. Exclude by defining `EVIL_NO_MAIN`.

Example

```
#include "evil.h"

Main then
  displayln("Hello, World!");
end
```

Evil Proc

Exclude by defining `EVIL_NO_PROC` before including evil.h

Introduces two syntactic elements:

- declare(Name, ReturnType, …); - A variadic macro. Arguments are as in C function arguments. Declares a C function.
- proc(Name, ReturnType, …) - A variadic macro. Arguments are as in C function arguments. Creates the start of a C function, that is, it is followed by a function body.

Example

```
#include "evil.h"

declare(add2, int, int a);

proc(add2, int, int a)
  return a + 2;
end

Main then
  displayln(add2(2));
end
```

Evil Specifier

Exclude by defining `EVIL_NO_SPECIFIER` before including evil.h

contant(Type, Name, Value) - A macro, generates a `const`.

storage(Type, Name, Value) - A macro, generates a `static`.

Example

```
constant(int, x, 12);
storage(int, y, 12);
```

Evil Structures

Exclude by defining `EVIL_NO_STRUCT` before including evil.h

Struct(Name) - Starts a struct definition.

Union(Name) - Start a union definition.

Typedef - A keyword, exactly equivalent to `typedef`.

BitField(Name, Type, Width) - Used for defining a Bitfield inside a struct.

Example

```
Struct(Pair)
    int a;
    int b;
end;

struct Pair x = {1, 2};
```

Evil Klass

Exclude by defining `EVIL_NO_KLASS` before including evil.h

This module is still under construction.

Evil Assert

Import by defining `EVIL_ASSERT` before including evil.h

Defines two identifiers:

- Assert(statement);
- AssertMsg(statement, reason);

If NDEBUG is defined, they become no-ops.

Otherwise, if the statement is true, the halt the program and spit out a helpful trace.

Example

```
Assert(false == true);
AssertMsg(false == true, "This is impossible.");
```

Evil Cli

Import by defining `EVIL_CLI` before including evil.h

This module provides a variety of functions for working with the terminal, by using ANSI escape sequences.

Warnings

- If EVIL_ASSERT is not defined, then cli_fg_256 and cli_bg_256 will be unavailable.
- None of the module's functions check if the function is supported by current terminal.

Definitions

- cli_reset() - Removes any active effects and colors from the terminal.
- cli_fg_256(N) - Takes a number between 0-255, and applies the corresponding color to the terminal text.
- cli_bg_256(N) - Takes a number between 0-255, and applies the corresponding color to the terminal background.
- cli_fg_rgb(R,G,B) - Takes three values (Red, Green, Blue), each of which is a number in the range 0-255, and applies the corresponding truecolor to the terminal text.
- cli_bg_rgb(R,G,B) - Takes three values (Red, Green, Blue), each of which is a number in the range 0-255, and applies the corresponding truecolor to the terminal background.
- cli_cursor_up(N) - Moves the console cursor up N lines.
- cli_cursor_down(N) - Moves the console cursor down N lines.
- cli_cursor_right(N) - Moves the console cursor right N characters.
- cli_cursor_left(N) - Moves the console cursor left N characters.
- cli_cursor_save() - Saves the current cursor position.
- cli_cursor_restore() - Moves the cursor to the last saved position.
- cli_screen_clear() - Clears the current terminal screen.
- cli_screen_clear_before() - Clears the current terminal screen, before the cursor.
- cli_screen_clear_after() - Clears the current terminal screen, after the cursor.
- cli_line_clear() - Clears the current terminal line.
- cli_line_clear_before() - Clears the current terminal line, before the cursor.
- cli_line_clear_after() - Clears the current terminal line, after the cursor.
- cli_effect_reset() - Removes any active effects from the terminal.
- cli_effect_bold() - Activates the bold text terminal effect.
- cli_effect_underline() - Activates the underlined text terminal effect.
- cli_effect_reverse() - Activates the reversed text terminal effect.
- cli_effect_blink() - Activates the blinking text terminal effect. (Often disabled on modern terminals).
- cli_effect_invisible() - Activates the invisible text terminal effect.
- cli_fg_black() - Sets the terminal text to simple black.
- cli_fg_red() - Sets the terminal text to simple red.
- cli_fg_green() - Sets the terminal text to simple green.
- cli_fg_yellow() - Sets the terminal text to simple yellow.

- cli_fg_blue() - Sets the terminal text to simple blue.
- cli_fg_magenta() - Sets the terminal text to simple magenta.
- cli_fg_cyan() - Sets the terminal text to simple cyan.
- cli_fg_white() - Sets the terminal text to simple white.
- cli_fg_bright_black() - Sets the terminal text to complex black.
- cli_fg_bright_red() - Sets the terminal text to complex red.
- cli_fg_bright_green() - Sets the terminal text to complex green.
- cli_fg_bright_yellow() - Sets the terminal text to complex yellow.
- cli_fg_bright_blue() - Sets the terminal text to complex blue.
- cli_fg_bright_magenta() - Sets the terminal text to complex magenta.
- cli_fg_bright_cyan() - Sets the terminal text to complex cyan.
- cli_fg_bright_white() - Sets the terminal text to complex white.
- cli_bg_black() - Sets the terminal background to simple black.
- cli_bg_red() - Sets the terminal background to simple red.
- cli_bg_green() - Sets the terminal background to simple green.
- cli_bg_yellow() - Sets the terminal background to simple yellow.
- cli_bg_blue() - Sets the terminal background to simple blue.
- cli_bg_magenta() - Sets the terminal background to simple magenta.
- cli_bg_cyan() - Sets the terminal background to simple cyan.
- cli_bg_white() - Sets the terminal background to simple white.
- cli_bg_bright_black() - Sets the terminal background to complex black.
- cli_bg_bright_red() - Sets the terminal background to complex red.
- cli_bg_bright_green() - Sets the terminal background to complex green.
- cli_bg_bright_yellow() - Sets the terminal background to complex yellow.
- cli_bg_bright_blue() - Sets the terminal background to complex blue.
- cli_bg_bright_magenta() - Sets the terminal background to complex magenta.
- cli_bg_bright_cyan() - Sets the terminal background to complex cyan.
- cli_bg_bright_white() - Sets the terminal background to complex white.

Example

```
#define EVIL_IO
#define EVIL_CLI
#include "evil.h"

Main then
  cli_fg_magenta();
  cli_bg_white();
  displayln("Coloured text!");
  cli_reset();
end
```

Evil Coroutine

Import by defining `EVIL_COROUTINE` before including evil.h

Warnings:

- Lambda's don't play well with Coroutines. Avoid using them in the body of a coroutine.
- Coroutine's don't support nesting. It may work sometimes, other times it may explode.

Definitions

coroutine() - A macro that marks the beginning of a coroutine body.

co_return(T) - A macro that marks a return from a coroutine.

co_end() - A macro that closes a coroutine body.

Example

```
  #define EVIL_IO
  #define EVIL_COROUTINE
  #include "evil.h"
  proc(example, int)
  storage(int, i, 0);
  coroutine();
  While true then
    co_return(++i);
  end
  co_end();
  return i;
end


Main then
  displayln(example());
  displayln(example());
  displayln(example());
  displayln(example());
  displayln(example());
  displayln(example());
  displayln(example());
  displayln(example());
end
```

Evil Hash

Import by defining `EVIL_HASH` before including evil.h

Warnings

- if int64_t is unavailable, then jenkins64, fletcher64 won't be defined.
- These are *not* cryptographic safe hashes.

Definitions

- jenkins64(char* key, size_t length) - Hash a given string into a uint64_t. Based on Jenkins One-At-A-Time hash.
- jenkins32(char* key, size_t length) - Hash a given string into a uint32_t. Based on Jenkins One-At-A-Time hash.
- fletcher64(char* key, size_t length) - Hash a given string into a uint64_t. Based on Fletcher's checksum.
- fletcher32(char* key, size_t length) - Hash a given string into a uint32_t. Based on Fletcher's checksum.
- fletcher16(char* key, size_t length) - Hash a given string into a uint16_t. Based on Fletcher's checksum.
- adler32(char* key, size_t length) - Hash a given string into a uint32_t. Based on Adler-32.

Example

```
#define EVIL_HASH
#include "evil.h"

Main then
  displayln(jenkins64("Hello, World!", 12));
  displayln(jenkins32("Hello, World!", 12));

  displayln(fletcher64("Hello, World!", 12));
  displayln(fletcher32("Hello, World!", 12));
  displayln(fletcher16("Hello, World!", 12));

  displayln(adler32("Hello, World!", 12));
end
```

Evil Help

Import by defining `EVIL_HELP` before including evil.h

Defines two functions:

- void evil_explain(const char* s)
- void evil_manual(void)

evil_manual prints this entire document to stdout.

evil_explain looks up an identifier and prints some information about it to stdout.

If `EVIL_NO_HELP_MANUAL` is defined, then evil_manual won't be created.

Evil Lambda

Import by defining `EVIL_LAMBDA` before including evil.h

functionPointer = lambda(returnType, body)

Lambda is a slightly more advanced feature, which requires the user how to construct and use function pointers.

```
### Example

    #define EVIL_LAMBDA
    #include "evil.h"

    Main then
      int (*max)(int, int) = lambda(int,
                                    (int x, int y) {
                                     return x > y ? x : y;
                                    });

      displayln(max(1, 2));
    end
```

Evil Malloc

Import by defining `EVIL_MALLOC` before including evil.h

Provides checked_malloc:

`checked_malloc(object, object_type, buffer, fail_msg, exit_q)`

- object - The identifier being assigned to.
- object_type - The type of the identifier being assigned to.
- buffer - The size of the buffer to pass to malloc.
- fail_msg - A string to print to stderr if it fails to allocate.
- exit_q - A boolean. If true, terminates the program.

Example

```
#define EVIL_MALLOC
#include "evil.h"

Main then
  comment("This will exit if malloc fails.");
  checked_malloc(x, char*, sizeof(char) * 6, "Out of Memory", true);

  x[0] = 'H';
  x[1] = 'e';
  x[2] = 'l';
  x[3] = 'l';
  x[4] = 'o';
  x[5] = '

  displayln(x);
  free(x);
end
```

Evil Math

Import by defining `EVIL_MATH` before including evil.h

Definitions

- add(a, b)
- take(a, b)
- multiply(a, b)
- divide(a, b)
- math.h

Evil Random

Import by defining `EVIL_RANDOM` before including evil.h

Warnings

- `randomseed` and `random` use C-rand, which is not cryptographically strong.

Definitions

- `randomseed(void)` - Seed the random generator.
- `random(min, max)` - Get a random integer in a given range.

Evil Log

Import by defining `EVIL_LOG` before including evil.h

Warnings

- `debug(char*)` is a no-op, unless `DEBUG_LOG` is defined.

Definitions

- `message(char* )`

Logs a message to stdout. If (char*)message_file has a length > 0, then logs to that file in append mode. If (char*)log_file has a length > 0, then logs to that file in append mode.

- `warning(char* )`

Logs a message to stdout. If (char*)warning_file has a length > 0, then logs to that file in append mode. If (char*)log_file has a length > 0, then logs to that file in append mode.

- `critical(char* )`

Logs a message to stderr. If (char*)critical_file has a length > 0, then logs to that file in append mode. If (char*)log_file has a length > 0, then logs to that file in append mode.

- `error(char* )`

Logs a message to stderr. If (char*)error_file has a length > 0, then logs to that file in append mode. If (char*)log_file has a length > 0, then logs to that file in append mode.

- `info(char* )`

Logs a message to stdout. If (char*)info_file has a length > 0, then logs to that file in append mode. If (char*)log_file has a length > 0, then logs to that file in append mode.

- `debug(char* )`

If DEBUG_LOG defined, logs a message to stdout. If (char*)debug_file has a length > 0, then logs to that file in append mode. If (char*)log_file has a length > 0, then logs to that file in append mode.

- `debug(char* )`

If DEBUG_LOG not defined, doesn't do anything.

Evil Bit

Import by defining `EVIL_BIT` before including evil.h

Adds worded bit operators.

Defines:

- BIT_AND(a, b)
- BIT_OR(a, b)
- BIT_XOR(a, b)
- BIT_NOT(a)
- BIT_RSHIFT(a, b)
- BIT_LSHIFT(a, b)

Evil Encode

Import by defining `EVIL_ENCODE` before including evil.h

This module is still under construction.

Examples

assert

```
#define EVIL_ASSERT
#include "evil.h"

comment("This demonstrates an assertion, with a message.");

/*
  Running it produces something like:

  Assertion `x < 0` failed: "This should always trigger."
  examples/assert.c @ line 17
  in `main`

*/

Main then
  int x = 1;
  AssertMsg(x < 0, "This should always trigger.");
end
```

bool

```
#include "evil.h"

comment("C's booleans are available by default.");

Main then
  bool x = false;
  displayln(x);

  displayln(true);

  displayln(false);

  displayln(true != false);
end
```

checked_malloc

```
#define EVIL_MALLOC
#include "evil.h"

Main then
  comment("This will exit if malloc fails.");
  checked_malloc(x, char*, sizeof(char) * 6, "Out of Memory", true);

  x[0] = 'H';
  x[1] = 'e';
  x[2] = 'l';
  x[3] = 'l';
  x[4] = 'o';
  x[5] = '\0';

  displayln(x);
  free(x);
end
```

comment

```
#include "evil.h"

comment("This is a demonstration of the comment \
    function. It expands to C-comments.");

comment(1 + 2 = 3);

Main then
  displayln("Hello, World!");
end
```

const

```
#include "evil.h"

comment("CNoEvil has a declarative way of creating \
    constants and storage-class specifiers.");

constant(int, x, 12);
storage(int, y, 2);

Main then
  displayln(x);
  displayln(y);
end
```

coroutine

```
// Silence coroutine warnings, so our test check doesn't complain.
// Make sure you know them before you do this.
#define EVIL_NO_WARN
#define EVIL_COROUTINE
#include "evil.h"

comment("A demonstration of a coroutine.");

proc(example, int)
  storage(int, i, 0);
  coroutine();
  While true then
    co_return(++i);
  end
  co_end();
  return i;
end

Main then
  displayln(example());
  displayln(example());
  displayln(example());
  displayln(example());
  displayln(example());
  displayln(example());
  displayln(example());
  displayln(example());
end
```

declare_and_proc

```
#include "evil.h"

comment("This is a function declerations.");
declare(add2, int, int a);

comment("This is that same function, but defined.");
proc(add2, int, int a)
  return a + 2;
end

Main then
  displayln(add2(2));
end
```

evil_cli

```
#define EVIL_ASSERT
#define EVIL_CLI
#include "evil.h"

Main then
  cli_fg_magenta();
  cli_bg_white();
  displayln("Coloured text!");
  cli_reset();
end
```

evil_manual

```
#define EVIL_HELP
#include "evil.h"

comment("evil_manual prints the manual to stdout.");

Main then
  evil_manual();
end
```

explain

```
#define EVIL_HELP
#include "evil.h"

#ifndef identifier
  #define identifier "EVIL_BOOL"
#endif

Main then
  evil_explain(identifier);
end
```

flow

```
#include "evil.h"

comment("CNoEvil has several whitespace-agnostic flow helpers.");

proc(gt, bool, int a, int b)
  If a > b then
    return true;
  Else
    return false;
  end
end

Main then
  While gt(1, 2) then
    displayln("Wat?");
  end

  int x = 1;
  Switch(x)
    Case(1)
      displayln("Woot!");
      break;
    default:
      displayln("Huh?");
  end

end
```

hash

```
#define EVIL_HASH
#include "evil.h"

Main then
  displayln(jenkins64("Hello, World!", 12));
  displayln(jenkins32("Hello, World!", 12));

  displayln(fletcher64("Hello, World!", 12));
  displayln(fletcher32("Hello, World!", 12));
  displayln(fletcher16("Hello, World!", 12));

  displayln(adler32("Hello, World!", 12));
end
```

helloworld

```
#include "evil.h"

comment("This is a simple program that prints to the screen.");
comment("Note the lack of formatter - it's generic.");

Main then
  displayln("Hello, World!");
end
```

klasses

```
#include "evil.h"

Class(String,
    // Values in self
    { size_t length; char* content; },
    // Initialiser
    {
        self->length = length;
        self->content = malloc(sizeof(char) * (length + 1));
        if(!self->content) {
            return false;
        }

        for(size_t i = 0; i < length; i++) {
            self->content[i] = content[i];
        }
        self->content[length] = 0;
    },
    // Init arguments (optional)
    size_t length, const char* content);

method(String, put, int, {
  return printf("%s\n", self->content);
});

method(String, del, void, {
    free(self->content);
    self->content = NULL;
});

Main then

  // Construct a new instance of our class
  new(x, String, 13, "Hello, World!");

  // Call a method the send way
  send(String, put, x);

  // Call the shorthand
  $(String, put, x);

  // Call a method the C way
  String_put(&x);

  // Cleanup
  send(String, del, x);

end
```

lambda

```
#define EVIL_LAMBDA
#include "evil.h"

Main then
  int (*max)(int, int) = lambda(int,
                                (int x, int y) {
                                   return x > y ? x : y;
                                });

  displayln(max(1, 2));
end
```

morse

```
#define EVIL_ENCODE
#include "evil.h"

Main then

  // Encode it
  char* s = morse_encode("Hello, World!", 14);
  printf("%s\n", s);

  // Decode it
  char* s2 = morse_decode(s, strlen(s));
  printf("%s\n", s2);

  // Clean up
  free(s);
  free(s2);

end
```

number

```
#include "evil.h"

comment("By default you get two simple type specifiers.");

comment("Number for int64_t. Decimal for long double.");

Main then
  Number(x) = 12;
  displayln(x);
  displayln(MaxNumber);

  Decimal(y) = 12.2;
  displayln(y);
  displayln(MaxDecimal);
end
```

random

```
#define EVIL_NO_WARN
#define EVIL_RANDOM
#include "evil.h"

Main then

  randomseed();

  int x = random(0, 10);
  displayln(x);

end
```

struct

```
#include "evil.h"

comment("This is a demonstration of \
    creating a struct.");

comment("Carefully note that it is still \
    followed by a semicolon.");

Struct(Pair)
  int a;
  int b;
end;

comment("You can typedef and create at the same time, \
 like C, but this is clearer.");

Typedef
  struct Pair
  Pair;

Main then
  Pair x = {1, 2};
  displayln(x.a);
end
```

Identifiers

`EVIL_NO_WARN`

Silences CnoEvil compile-time warnings, when defined.

Do not use unless you understand the full implications.

`true`

bool - defined in `EVIL_BOOL`.

`false`

bool - defined in `EVIL_BOOL`.

`bool`

type - defined in `EVIL_BOOL`.

`__bool_true_false_are_defined`

Value of 1 - defined in `EVIL_BOOL`.

`EVIL_BOOL`

module - Adds booleans.

Available by default.

Exclude by defining `EVIL_NO_BOOL`

`EVIL_NO_BOOL`

Exclude the `EVIL_BOOL` module when defined.

`EVIL_COMMENT`

module - Adds a comment syntax.

Available by default.

Exclude by defining `EVIL_NO_COMMENT`

`EVIL_NO_COMMENT`

Exclude the `EVIL_COMMENT` module when defined.

`comment`

`comment(statement);` - Make comments like normal functions.

Defined in `EVIL_COMMENT`.

`EVIL_FLOW`

module - Adds a new syntax.

Available by default.

Exclude by defining `EVIL_NO_FLOW`

EVIL_NO_FLOW

Exclude the `EVIL_FLOW` module when defined.

then

`then` - Meant to follow most flow statements.

Defined in `EVIL_FLOW`.

end

`end` - Meant to finish most flow statements.

Defined in `EVIL_FLOW`.

If

`If` - Opens an if statement without the need for braces.

Defined in `EVIL_FLOW`.

Else

`Else` - Closes one body, injects `else`, and opens a new body.

Defined in `EVIL_FLOW`.

For

`For` - Opens a for statement without the need for braces.

Defined in `EVIL_FLOW`.

While

`While` - Opens a while statement without the need for braces.

Defined in `EVIL_FLOW`.

Do

`Do` - Opens a do statement without the need for braces.

Defined in `EVIL_FLOW`.

Switch

`Switch(x)` - Opens a switch statement without the need for a following brace.

Defined in `EVIL_FLOW`.

Case

`Case(x)` - Opens a case in a syntax-consistent way.

Defined in `EVIL_FLOW`.

`EVIL_INT`

module - Adds number specifiers.

Available by default.

Exclude by defining `EVIL_NO_INT`

`EVIL_NO_INT`

Exclude the `EVIL_INT` module when defined.

`Number`

`Number(x)` - declare a number type. Equivalent to `int64_t`. If int64_t is not supported, this method is unavailable.

Defined in `EVIL_INT`.

`MaxNumber`

`MaxNumber` - The maximum value for a Number type.

Defined in `EVIL_INT`.

`Decimal`

`Decimal(x)` - declare a float type. Equivalent to `long double`.

Defined in `EVIL_INT`.

`MaxDecimal`

`MaxDecimal` - The maximum value for a Decimal type.

Defined in `EVIL_INT`.

`EVIL_IO`

module - Adds IO helpers.

Available by default.

Exclude by defining `EVIL_NO_IO`

`EVIL_NO_IO`

Exclude the `EVIL_IO` module when defined.

`display_format`

```
display_format(x);
```

Creates a string that is probably a format specifier for the given object.

Defined in `EVIL_IO`.

`display`

```
display(x);
```

Prints a representation of the given object to stdout.

Also see: displayf, displayln, displayfln, endl.

Defined in `EVIL_IO`.

`displayf`

```
displayf(file, x);
```

Prints a representation of the given object to the `FILE*` given as the first argument.

Also see: display, displayln, displayfln, endl.

Defined in `EVIL_IO`.

`displayln`

```
displayln(x);
```

Prints a representation of the given object to stdout, followed by a system-dependant newline.

Also see: displayf, display, displayfln, endl.

Defined in `EVIL_IO`.

`displayfln`

```
displayfln(file, x);
```

Prints a representation of the given object to the `FILE*` given as the first argument, followed by a system-dependant newline.

Also see: displayf, display, displayln, endl.

Defined in `EVIL_IO`.

`display_format`

```
display_format(x);
```

Creates a string that is probably a format specifier for the given object.

Defined in `EVIL_IO`.

`endl`

```
endl;
```

Sends a system-dependant newline to stdout.

Also see: endlf

Defined in `EVIL_IO`.

`endlf`

```
endlf(file);
```

Sends a system-dependant newline to the `FILE*` given as the first argument.

Also see: endl

Defined in `EVIL_IO`.

repr_type

repr_type(x)

Creates a string representation of the type that the object probably is.

Defined in EVIL_IO.

stdio

The stdio module.

Imported by EVIL_IO.

EVIL_MAIN

module - Adds Main.

Available by default.

Exclude by defining EVIL_NO_MAIN

EVIL_NO_MAIN

Excludes ‘EVIL_MAIN1 when defined.

Main

Opens the main function, and defines argc and argv.

See also: argc, argv

Defined in EVIL_MAIN.

argc

Automatically available under Main.

An int representing the number of arguments given on the command line.

See also: Main, argv

argv

Automatically available under Main.

A char*[] containing the arguments given on the command line.

See also: Main, argc

EVIL_PROC

module - adds proc and declare.

Available by default.

Exclude by defining EVIL_NO_PROC

EVIL_NO_PROC

Exclude EVIL_PROC when defined

proc

proc(name, return type, args…) - Opens the body of a function definition.

Defined in EVIL_PROC.

declare

declare(name, return type, args…); - Creates a function decleration.

Defined in EVIL_PROC.

EVIL_SPECIFIER

module - Adds constant and storage.

Available by default.

Exclude by defining EVIL_NO_SPECIFIER.

EVIL_NO_SPECIFIER

Exclude EVIL_SPECIFIER when defined.

constant

constant(type, name, value); - Create constant value.

Defined in EVIL_SPECIFIER.

storage

storage(type, name, value); - Create what the C standard calls a “storage-class specifier” value, a static value.

Defined in EVIL_SPECIFIER.

EVIL_STRUCTURES

module - adds Struct, Union, Typedef and BitField.

Available by default.

Exclude by defining EVIL_NO_STRUCT

EVIL_NO_STRUCT

Exclude EVIL_STRUCTURES by defining.

Struct

Struct(name) - Opens a new structure definition.

Defined in EVIL_STRUCTURES.

Union

Union(name) - Opens a new union definition.

Defined in EVIL_STRUCTURES.

`BitField`

BitField(name, type, width); - Creates a bitfield specification.

Defined in `EVIL_STRUCTURES`.

`Typedef`

Typedef - Creates a typedef in the same manner as `typedef`.

Defined in `EVIL_STRUCTURES`.

`EVIL_ASSERT`

module - An assertion library.

Defines: Assert, AssertMsg

See also: NDEBUG

Import by defining `EVIL_ASSERT` before including `evil.h`.

`NDEBUG`

If defined, Assert and AssertMsg become no-nops.

`AssertMsg`

AssertMsg(statement, message); - If statement is not true, print a stacktrace with a given message, then raise SIGABRT.

Defined in `EVIL_ASSERT`.

`Assert`

Assert(statement, message); - If statement is not true, print a stacktrace, then raise SIGABRT.

Defined in `EVIL_ASSERT`.

`EVIL_MALLOC`

module - A 'safer' malloc.

Defines: checked_malloc

Import by defining `EVIL_MALLOC` before including `evil.h`

`checked_malloc`

```
checked_malloc(object, object_type, buffer, fail_msg, exit_q);
```

- object - The identifier being assigned to.
- object_type - The type of the identifier being assigned to.
- buffer - The size of the buffer to pass to malloc.
- fail_msg - A string to print to stderr if it fails to allocate.
- exit_q - A boolean. If true, terminates the program.

Defined in `EVIL_MALLOC`.

EVIL_LAMBDA

module - Lambda support.

Defines - `lambda`

Import by defining `EVIL_LAMBDA` before including `evil.h`

lambda

`lambda(return type, body)`

A lambda returns a function pointer, and takes a return type and body definition.

Defined in `EVIL_LAMBDA`

Best to check the examples and `evil_manual` for this one.

EVIL_RANDOM

module - Random numbers.

Defines: `randomseed` and `random`

Import by defining `EVIL_RANDOM` before including `evil.h`

randomseed

`randomseed()` - Seeds the C-rand number generator.

Defined in `EVIL_RANDOM`.

random

`random(min, max)` - Takes two `int`s, and produces an `int` within the range.

Defined in `EVIL_RANDOM`.

EVIL_HELP

module - Provides help documentation.

Defines: `evil_manual` and `evil_explain`

See also: `EVIL_NO_HELP_MANUAL`

Import by defining `EVIL_HELP` before including `evil.h`.

EVIL_NO_HELP_MANUAL

If defined before `EVIL_HELP` then `evil_manual` won't be defined.

This should help reduce the size of the eventual binary.

evil_manual

```
evil_manual();
```

Prints general help to stdout.

Defined in `EVIL_HELP`

evil_explain

`evil_explain(const char* s);`

Prints help about a specific identifier.

Defined in `EVIL_HELP`

EVIL_COROUTINE

module - Adds coroutine support.

Defines: `coroutine`, `co_return` and `co_end`.

Import by defining `EVIL_COROUTINE` before including `evil.h`.

coroutine

`coroutine();` - Begin a coroutine body.

Defined in `EVIL_COROUTINE`.

co_return

`co_return(value)` - Yield a value.

Defined in `EVIL_COROUTINE`

co_end

`co_end();` - End the coroutine body.

Defined in `EVIL_COROUTINE`

EVIL_MATH

module - Adds better math support.

Links to `math.h`.

Defines: `add`, `take`, `divide` and `multiply` generics.

Import by defining `EVIL_MATH` before including `evil.h`.

add

`add(a, b)` - Translated to `( a + b )`, allowing it to act as a generic.

Defined in `EVIL_MATH`.

take

`take(a, b)` - Translated to `( a - b )`, allowing it to act as a generic.

Defined in `EVIL_MATH`.

multiply

`multiply(a, b)` - Translated to `( a * b )`, allowing it to act as a generic.

Defined in `EVIL_MATH`.

divide

`divide(a, b)` - Translated to `( a / b )`, allowing it to act as a generic.

Defined in `EVIL_MATH`.

EVIL_HASH

module - Provides string hash functions.

Defines: `jenkins64`, `jenkins32`, `fletcher64`, `fletcher32`, `fletcher16`, `adler32`.

Import by defining `EVIL_HASH` before including `evil.h`.

jenkins64

uint64_t jenkins64(char* key, size_t length);

Only defined if int64_t available.

The same input should always get the same output.

This is an implementation of a Jenkins One-At-A-Time Hash.

This is not a cryptographic hash function.

Defined in `EVIL_HASH`.

jenkins32

uint32_t jenkins32(char* key, size_t length);

The same input should always get the same output.

This is an implementation of a Jenkins One-At-A-Time Hash.

This is not a cryptographic hash function.

Defined in `EVIL_HASH`.

fletcher64

uint64_t fletcher64(char* key, size_t length);

Only defined if int64_t available.

The same input should always get the same output.

This is an implementation of a Fletcher's Checksum.

This is not a cryptographic hash function.

Defined in `EVIL_HASH`.

fletcher32

uint32_t fletcher32(char* key, size_t length);

The same input should always get the same output.

This is an implementation of a Fletcher's Checksum.

This is not a cryptographic hash function.

Defined in `EVIL_HASH`.

`fletcher16`

uint16_t fletcher16(char* key, size_t length);

The same input should always get the same output.

This is an implementation of a Fletcher's Checksum.

This is not a cryptographic hash function.

Defined in `EVIL_HASH`.

`adler32`

uint32_t adler32(char* key, size_t length);

The same input should always get the same output.

This is an implementation of a Adler-32 checksum.

This is not a cryptographic hash function.

Defined in `EVIL_HASH`.

`EVIL_BIT`

module - defines macros for worded bit operators.

Defines: `BIT_AND`, `BIT_OR`, `BIT_XOR`, `BIT_NOT`, `BIT_RSHIFT`, `BIT_LSHIFT`.

Import by defining `EVIL_BIT` before including `evil.h`.

`BIT_AND`

BIT_AND(a, b) - Return a bitwise AND.

Defined in `EVIL_BIT`

`BIT_OR`

BIT_OR(a, b) - Return a bitwise OR.

Defined in `EVIL_BIT`

`BIT_XOR`

BIT_XOR(a, b) - Return a bitwise XOR.

Defined in `EVIL_BIT`

`BIT_NOT`

BIT_NOT(a, b) - Return a bitwise NOT.

Defined in `EVIL_BIT`

`BIT_RSHIFT`

BIT_RSHIFT(a, b) - Return a bitwise right shift.

Defined in `EVIL_BIT`

`BIT_LSHIFT`

BIT_LSHIFT(a, b) - Return a bitwise left shift.

Defined in `EVIL_BIT`

`EVIL_LOG`

module - a helpful logging module.

Defines:

- message_file
- warning_file
- critical_file
- error_file
- info_file
- debug_file
- log_file
- message
- warning
- critical
- error
- info
- debug

See also: DEBUG_LOG

`DEBUG_LOG`

If defined, then `debug` works as expected.

If *not* defined, then `debug` is a no-op.

Referenced in `EVIL_LOG`.

`debug_file`

`char* debug_file`

See `debug`.

Defined in `EVIL_LOG`

`debug`

void debug(char* str); - When `EVIL_LOG` defined, prints a structured message to stdout.

If `debug_file` is not 0-length, then prints a structured message to the file found at that path.

Defined in `EVIL_LOG`.

`info_file`

`char* info_file`

See `info`.

Defined in `EVIL_LOG`

`info`

void info(char* str); - Prints a structured message to stdout.

If `info_file` is not 0-length, then prints a structured message to the file found at that path.

Defined in `EVIL_LOG`.

`error_file`

`char* error_file`

See `error`.

Defined in `EVIL_LOG`

`error`

void error(char* str); - Prints a structured message to stderr.

If `error_file` is not 0-length, then prints a structured message to the file found at that path.

Defined in `EVIL_LOG`.

`critical_file`

`char* critical_file`

See `critical`.

Defined in `EVIL_LOG`

`critical`

void critical(char* str); - Prints a structured message to stderr.

If `critical_file` is not 0-length, then prints a structured message to the file found at that path.

Defined in `EVIL_LOG`.

`warning_file`

`char* warning_file`

See `warning`.

Defined in `EVIL_LOG`

`warning`

void warning(char* str); - Prints a structured message to stdout.

If `warning_file` is not 0-length, then prints a structured message to the file found at that path.

Defined in `EVIL_LOG`.

`message_file`

`char* message_file`

See `message`.

Defined in `EVIL_LOG`

`message`

void message(char* str); - Prints a structured message to stdout.

If `message_file` is not 0-length, then prints a structured message to the file found at that path.

Defined in `EVIL_LOG`.

Source

```
1
2  #ifndef CNOEVIL
3  #define CNOEVIL "3.0.0"
4
5  // Programmer's Notes
6  //
7  // Every part of CNOEvil must be able to do one of:
8  // * Be excluded by a preprocessor definition (e.g. #define EVIL_NO_MODULE)
9  // * Be included by a preprocessor definition (e.g. #define EVIL_MODULE)
10 //
11 // We reserve EVIL_ for module names.
12 // Long module names are discouraged.
13 //
14 // We try and extend C. Now and then we might have to break it, but that is
15 // truly exceptional, and must be thoroughly documented.
16 //
17 // CNoEvil is intended to be intuitive to write,
18 // and clear to read.
19 //
20 // C11 features are fine. We assume a modern compiler.
21 //
22 // GNU-only features are discouraged, but I could be convinced to switch only to them.
23 // If you really need it, consider having both GNU and Clang extensions.
24 // If that's not possible - the module *must* be optional, and *must* have an #error
    ↪  directive
25 // for if the needed feature is unavailable.
26 //
27 // Whitespace sensitivity is abhorrent, and should never be introduced.
28 //
29 // Your code should make sense once it's expanded. (Human readable. Not only machine
    ↪  readable.)
30 //
31 // Libraries that might be expected to instantly be available in a higher-level language can
    ↪  be
32 // automatically included. All others must be specified.
33 //
34 // CNoEvil is a 1-file header library.
35 // That doesn't mean we're bound to macros-only, but it's probably a good
36 // idea to limit the number of functions.
37 //
38 // Yes, we have lots of header files. They all get compiled into a single one.
39 //
40 // All macros and functions _must_ have a matching counterpart in:
41 // * evil_help routines
42 // * examples/ folder
43 //
44 // CNoEvil aims to be a language atop C, aimed at making the programmer's life
45 // *simpler*. Simpler =/= easier. The programmer can be expected to know C.
46 //
47
```

```
48 /*
49 Copyright 2020 James Milne,
50
51 Redistribution and use in source and binary forms, with or without modification, are
    ↪ permitted provided that the following conditions are met:
52
53 1. The licensee acknowledges that this software is utterly insane in it's nature, and not
    ↪ fit for any purpose.
54
55 2. Redistributions of source code must retain the above copyright notice, this list of
    ↪ conditions and the following disclaimer.
56
57 3. Redistributions in binary form must reproduce the above copyright notice, this list of
    ↪ conditions and the following disclaimer in the documentation and/or other materials
    ↪ provided with the distribution.
58
59 4. Neither the name of the copyright holder nor the names of its contributors may be used to
    ↪ endorse or promote products derived from this software without specific prior written
    ↪ permission.
60
61 ***THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY
    ↪ EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
    ↪ MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL
    ↪ THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
    ↪ SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT
    ↪ OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
    ↪ INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
    ↪ LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
    ↪ OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.***
62 */
63
64 // Build stats:
65
66 // Size: 98905
67 // Built: 1580532171
68 // Machine: 5.4.10-arch1-1 x86_64 GNU/Linux
69 // EVIL_HASH_VER == sh -c 'cat include/head.h $(find include -name "*.h" -not -name head.h
    ↪ -not -name tail.h -print0 | sort -z | xargs -r0 echo) include/tail.h | sha512sum | tr -d
    ↪ " -" | tail -c 15'
70
71 #define EVIL_HASH_VER "7a2cc152cce77d"
72
73
74 #ifdef EVIL_ASSERT
75
76     #ifndef NDEBUG
77
78     #include <stdio.h>
79     #include <signal.h>
80
```

```
81      #define Assert(_expression) if((_expression) != 1) { fprintf(stderr, "Assertion `%s`
         ↪  failed\n%s @ line %d\nin `%s`\n", #_expression, __FILE__, __LINE__, __func__);
         ↪  raise(SIGABRT); }
82
83      #define AssertMsg(_expression, _reason) if((_expression) != 1) { fprintf(stderr,
         ↪  "Assertion `%s` failed: %s\n%s @ line %d\nin `%s`\n", #_expression, #_reason,
         ↪  __FILE__, __LINE__, __func__); raise(SIGABRT); }
84
85      #else
86
87      #define Assert(ignore)((void) 0)
88      #define AssertMsg(ignore, ignore_again)((void) 0)
89
90      #endif
91
92  #endif
93  #ifdef EVIL_BIT
94
95    #define BIT_AND(a, b) (a & b)
96    #define BIT_OR(a, b) (a | b)
97    #define BIT_XOR(a, b) (a ^ b)
98    #define BIT_NOT(a) (~a)
99    #define BIT_RSHIFT(a, b) (a >> b)
100   #define BIT_LSHIFT(a, b) (a << b)
101
102
103 #endif
104 #ifndef EVIL_NO_BOOL
105   // Included by default
106   #include <stdbool.h>
107 #endif
108
109 #ifdef EVIL_CLI
110   //Taken from my library, colors.h (https://github.com/shakna-israel/colors)
111   //Also taken from my library, damned (curses for shell)
       ↪  (https://github.com/shakna-israel/damned)
112   #include <stdio.h>
113
114   #ifndef EVIL_ASSERT
115     #warning "EVIL_CLI depends on EVIL_ASSERT. Without it, cli_fg_256 and cli_bg_256 will be
       ↪  unavailable."
116   #endif
117
118   #define cli_reset() printf("%s", "\x1b[0m"); fflush(stdout)
119
120   #ifdef EVIL_ASSERT
121     #define cli_fg_256(num) Assert(num < 256); printf("%s%d%s", "\x1b[38;5;", num, "m")
122     #define cli_bg_256(num) Assert(num < 256); printf("%s%d%s", "\x1b[48;5;", val, "m")
123   #endif
124
125   // Also known as truecolor.
126   #define cli_fg_rgb(a, b, c) printf("\x1b[38;2;%d;%d;%d", a, b, c)
```

```
127 #define cli_bg_rgb(a, b, c) printf("\x1b[48;2;%d;%d;%d", a, b, c)
128
129 #define cli_fg_black() printf("%s", "\x1b[30m")
130 #define cli_fg_red() printf("%s", "\x1b[31m")
131 #define cli_fg_green() printf("%s", "\x1b[32m")
132 #define cli_fg_yellow() printf("%s", "\x1b[33m")
133 #define cli_fg_blue() printf("%s", "\x1b[34m")
134 #define cli_fg_magenta() printf("%s", "\x1b[35m")
135 #define cli_fg_cyan() printf("%s", "\x1b[36m")
136 #define cli_fg_white() printf("%s", "\x1b[37m")
137
138 #define cli_fg_bright_black() printf("%s", "\x1b[30;1m")
139 #define cli_fg_bright_red() printf("%s", "\x1b[31;1m")
140 #define cli_fg_bright_green() printf("%s", "\x1b[32;1m")
141 #define cli_fg_bright_yellow() printf("%s", "\x1b[33;1m")
142 #define cli_fg_bright_blue() printf("%s", "\x1b[34;1m")
143 #define cli_fg_bright_magenta() printf("%s", "\x1b[35;1m")
144 #define cli_fg_bright_cyan() printf("%s", "\x1b[36;1m")
145 #define cli_fg_bright_white() printf("%s", "\x1b[37;1m")
146
147 #define cli_bg_black() printf("%s", "\x1b[40m")
148 #define cli_bg_red() printf("%s", "\x1b[41m")
149 #define cli_bg_green() printf("%s", "\x1b[42m")
150 #define cli_bg_yellow() printf("%s", "\x1b[43m")
151 #define cli_bg_blue() printf("%s", "\x1b[44m")
152 #define cli_bg_magenta() printf("%s", "\x1b[45m")
153 #define cli_bg_cyan() printf("%s", "\x1b[46m")
154 #define cli_bg_white() printf("%s", "\x1b[47m")
155
156 #define cli_bg_bright_black() printf("%s", "\x1b[40;1m")
157 #define cli_bg_bright_red() printf("%s", "\x1b[41;1m")
158 #define cli_bg_bright_green() printf("%s", "\x1b[42;1m")
159 #define cli_bg_bright_yellow() printf("%s", "\x1b[43;1m")
160 #define cli_bg_bright_blue() printf("%s", "\x1b[44;1m")
161 #define cli_bg_bright_magenta() printf("%s", "\x1b[45;1m")
162 #define cli_bg_bright_cyan() printf("%s", "\x1b[46;1m")
163 #define cli_bg_bright_white() printf("%s", "\x1b[47;1m")
164
165 #define cli_cursor_up(num) printf("\x1b%dA", num)
166 #define cli_cursor_down(num) printf("\x1b%dB", num)
167 #define cli_cursor_left(num) printf("\x1b%dC", num)
168 #define cli_cursor_right(num) printf("\x1b%dD", num)
169 #define cli_cursor_save() printf("\x1bs")
170 #define cli_cursor_restore() printf("\x1bu")
171
172 #define cli_screen_clear() printf("%s", "\x1b[2J\x1b[1;1H"); fflush(stdout)
173 #define cli_screen_clear_before() printf("%s", "\x1b[1J"); fflush(stdout)
174 #define cli_screen_clear_after() printf("%s", "\x1b[0J"); fflush(stdout)
175
176 #define cli_line_clear() printf("%s", "\x1b[2K"); fflush(stdout)
177 #define cli_line_clear_before() printf("%s", "\x1b[1K"); fflush(stdout)
178 #define cli_line_clear_after() printf("%s", "\x1b[0K"); fflush(stdout)
```

```
179
180   #define cli_effect_bold() printf("%s", "\x1b[1m")
181   #define cli_effect_underline() printf("%s", "\x1b[4m")
182   #define cli_effect_reverse() printf("%s", "\x1b[7m")
183   #define cli_effect_blink() printf("%s", "\x1b[5m")
184   #define cli_effect_invisible() printf("%s", "\x1b[8m")
185   #define cli_effect_reset() printf("%s", "\x1b[0m")
186 #endif
187
188 #ifndef EVIL_NO_COMMENT
189   // Included by default
190   #define comment(...) /* __VA_ARGS__ */
191 #endif
192
193 #ifdef EVIL_COROUTINE
194   // This lovely hack makes use of switch statements,
195   // And the __LINE__ C macro
196   // It tracks the current state, and switches case.
197   // But... I imagine awful things may happen with an extra semi-colon.
198   // Which would be hard to debug.
199   #if defined(EVIL_LAMBDA) && !defined(EVIL_NO_WARN)
200     // And bad things happen with expression statements.
201     #warning "Lambda's don't play well with Coroutines."
202 " Avoid using them in the body of a coroutine."
203   #endif
204   #ifndef EVIL_NO_WARN
205     #warning "Coroutine's don't support nesting. It may work sometimes,"
206 " other times it may explode."
207   #endif
208
209   // Original macro hack by Robert Elder (c) 2016. Used against
210   // their advice, but with their permission.
211   #define coroutine() static int state=0; switch(state) { case 0:
212   #define co_return(x) { state=__LINE__; return x; case __LINE__:; }
213   #define co_end() }
214 #endif
215
216
217 #ifdef EVIL_ENCODE
218
219   #include <stdlib.h>
220   #include <string.h>
221
222   #ifndef EVIL_NO_WARN
223     #warning "ENCODE module is still under construction."
224   #endif
225
226   // Declerations.
227   char* base16_encode(char* input, size_t length); // NULL on failure.
228   char* base16_decode(char* input, size_t length); // NULL on failure.
229
230   char* morse_encode(char* input, size_t length); // NULL on failure.
```

```c
char* morse_decode(char* input, size_t length); // NULL on failure.

// TODO: base64

// TODO: base32

// base16
char* base16_encode(char* input, size_t length)
{
  static const char* const lut = "0123456789ABCDEF";

  size_t str_pos = 0;
  char* output = malloc(sizeof(char) * (length * 2));
  if(!output) {
    return NULL;
  }

  for(size_t i = 0; i < length; i++) {
    const unsigned char c = input[i];
    output[str_pos] = lut[c >> 4];
    str_pos++;
    output[str_pos] = lut[c & 15];
    str_pos++;
  }
  return output;
}

char* base16_decode(char* input, size_t length)
{
  static const char* const lut = "0123456789ABCDEF";

  // Length must be even
  if(length % 2 != 0) {
    return NULL;
  }

  char* output = malloc(sizeof(char) * (length / 2));

  for(size_t i = 0, j = 0; i < (length/2); i++, j++) {
    output[i] = (input[j] & '@' ? input[j] + 9 : input[j]) << 4, j++;
    output[i] |= (input[j] & '@' ? input[j] + 9 : input[j]) & 0xF;
  }
  output[length/2] = 0;
  return output;
}

// TODO: morse
char* morse_encode(char* input, size_t inlength)
{
  // Get the output length. Morse isn't fixed width.
  size_t length = 0;
  for(size_t i = 0; i < inlength; i++) {
```

```
283         switch(input[i]) {
284           case('A'):
285           case('a'):
286             // .- [space]
287             length += 3;
288             break;
289           case('B'):
290           case('b'):
291             // -... [space]
292             length += 5;
293             break;
294           case('C'):
295           case('c'):
296             // -.-. [space]
297             length += 5;
298             break;
299           case('D'):
300           case('d'):
301             // -.. [space]
302             length += 4;
303             break;
304           case('E'):
305           case('e'):
306             // . [space]
307             length += 2;
308             break;
309           case('F'):
310           case('f'):
311             // ..-. [space]
312             length += 5;
313             break;
314           case('G'):
315           case('g'):
316             // --. [space]
317             length += 4;
318             break;
319           case('H'):
320           case('h'):
321             // .... [space]
322             length += 5;
323             break;
324           case('I'):
325           case('i'):
326             // .. [space]
327             length += 3;
328             break;
329           case('J'):
330           case('j'):
331             // .--- [space]
332             length += 5;
333             break;
334           case('K'):
```

```
335         case('k'):
336           // -.- [space]
337           length += 4;
338           break;
339         case('L'):
340         case('l'):
341           // .--- [space]
342           length += 5;
343           break;
344         case('M'):
345         case('m'):
346           // -- [space]
347           length += 3;
348           break;
349         case('N'):
350         case('n'):
351           // -. [space]
352           length += 3;
353           break;
354         case('O'):
355         case('o'):
356           // --- [space]
357           length += 4;
358           break;
359         case('P'):
360         case('p'):
361           // .--. [space]
362           length += 5;
363           break;
364         case('Q'):
365         case('q'):
366           // --.- [space]
367           length += 5;
368           break;
369         case('R'):
370         case('r'):
371           // .-. [space]
372           length += 4;
373           break;
374         case('S'):
375         case('s'):
376           // ... [space]
377           length += 4;
378           break;
379         case('T'):
380         case('t'):
381           // - [space]
382           length += 2;
383           break;
384         case('U'):
385         case('u'):
386           // ..- [space]
```

```
387             length += 4;
388             break;
389           case('V'):
390           case('v'):
391             // ...- [space]
392             length += 5;
393             break;
394           case('W'):
395           case('w'):
396             // .-- [space]
397             length += 4;
398             break;
399           case('X'):
400           case('x'):
401             // -..- [space]
402             length += 5;
403             break;
404           case('Y'):
405           case('y'):
406             // -.-- [space]
407             length += 5;
408             break;
409           case('Z'):
410           case('z'):
411             // --.. [space]
412             length += 5;
413             break;
414           case('0'):
415             // ----- [space]
416             length += 6;
417             break;
418           case('1'):
419             // .---- [space]
420             length += 6;
421             break;
422           case('2'):
423             // ..--- [space]
424             length += 6;
425             break;
426           case('3'):
427             // ...-- [space]
428             length += 6;
429             break;
430           case('4'):
431             // ....- [space]
432             length += 6;
433             break;
434           case('5'):
435             // ..... [space]
436             length += 6;
437             break;
438           case('6'):
```

```
439           // -.... [space]
440           length += 6;
441           break;
442         case('7'):
443           // --... [space]
444           length += 6;
445           break;
446         case('8'):
447           // ---.. [space]
448           length += 6;
449           break;
450         case('9'):
451           // ----. [space]
452           length += 6;
453           break;
454         default:
455           // Ignore unknown characters
456           break;
457       }
458     }
459
460     // Allocate
461     char* s = malloc(sizeof(char) * (length + 1));
462
463     if(!s) {
464       return NULL;
465     }
466
467     // Iterate
468     size_t out_pos = 0;
469     for(size_t i = 0; i < inlength; i++) {
470       switch(input[i]) {
471         case('A'):
472         case('a'):
473           // .- [space]
474           s[out_pos] = '.';
475           s[out_pos+1] = '-';
476           s[out_pos+2] = ' ';
477           out_pos += 3;
478           break;
479         case('B'):
480         case('b'):
481           // -... [space]
482           s[out_pos] = '-';
483           s[out_pos+1] = '.';
484           s[out_pos+2] = '.';
485           s[out_pos+3] = '.';
486           s[out_pos+4] = ' ';
487           out_pos += 5;
488           break;
489         case('C'):
490         case('c'):
```

```
491             // -.-. [space]
492             s[out_pos] = '-';
493             s[out_pos] = '.';
494             s[out_pos] = '-';
495             s[out_pos] = '.';
496             s[out_pos] = ' ';
497             out_pos += 5;
498             break;
499           case('D'):
500           case('d'):
501             // -.. [space]
502             s[out_pos] = '-';
503             s[out_pos+1] = '.';
504             s[out_pos+2] = '.';
505             s[out_pos+3] = ' ';
506             out_pos += 4;
507             break;
508           case('E'):
509           case('e'):
510             // . [space]
511             s[out_pos] = '.';
512             s[out_pos+1] = ' ';
513             out_pos += 2;
514             break;
515           case('F'):
516           case('f'):
517             // ..-. [space]
518             s[out_pos] = '.';
519             s[out_pos+1] = '.';
520             s[out_pos+2] = '-';
521             s[out_pos+3] = '.';
522             s[out_pos+4] = ' ';
523             out_pos += 5;
524             break;
525           case('G'):
526           case('g'):
527             // --. [space]
528             s[out_pos] = '-';
529             s[out_pos+1] = '-';
530             s[out_pos+2] = '.';
531             s[out_pos+3] = ' ';
532             out_pos += 4;
533             break;
534           case('H'):
535           case('h'):
536             // .... [space]
537             s[out_pos] = '.';
538             s[out_pos+1] = '.';
539             s[out_pos+2] = '.';
540             s[out_pos+3] = '.';
541             s[out_pos+4] = ' ';
542             out_pos += 5;
```

```
543             break;
544           case('I'):
545           case('i'):
546             // .. [space]
547             s[out_pos] = '.';
548             s[out_pos+1] = '.';
549             s[out_pos+2] = ' ';
550             out_pos += 3;
551             break;
552           case('J'):
553           case('j'):
554             // .--- [space]
555             s[out_pos] = '.';
556             s[out_pos+1] = '-';
557             s[out_pos+2] = '-';
558             s[out_pos+3] = '-';
559             s[out_pos+4] = ' ';
560             out_pos += 5;
561             break;
562           case('K'):
563           case('k'):
564             // -.- [space]
565             s[out_pos] = '-';
566             s[out_pos+1] = '.';
567             s[out_pos+2] = '-';
568             s[out_pos+3] = ' ';
569             out_pos += 4;
570             break;
571           case('L'):
572           case('l'):
573             // .-.. [space]
574             s[out_pos] = '.';
575             s[out_pos+1] = '-';
576             s[out_pos+2] = '.';
577             s[out_pos+3] = '.';
578             s[out_pos+4] = ' ';
579             out_pos += 5;
580             break;
581           case('M'):
582           case('m'):
583             // -- [space]
584             s[out_pos] = '-';
585             s[out_pos+1] = '-';
586             s[out_pos+2] = ' ';
587             out_pos += 3;
588             break;
589           case('N'):
590           case('n'):
591             // -. [space]
592             s[out_pos] = '-';
593             s[out_pos+1] = '.';
594             s[out_pos+2] = ' ';
```

```
595           out_pos += 3;
596           break;
597         case('O'):
598         case('o'):
599           // --- [space]
600           s[out_pos] = '-';
601           s[out_pos+1] = '-';
602           s[out_pos+2] = '-';
603           s[out_pos+3] = ' ';
604           out_pos += 4;
605           break;
606         case('P'):
607         case('p'):
608           // .--. [space]
609           s[out_pos] = '.';
610           s[out_pos+1] = '-';
611           s[out_pos+2] = '-';
612           s[out_pos+3] = '.';
613           s[out_pos+4] = ' ';
614           out_pos += 5;
615           break;
616         case('Q'):
617         case('q'):
618           // --.- [space]
619           s[out_pos] = '-';
620           s[out_pos] = '-';
621           s[out_pos] = '.';
622           s[out_pos] = '-';
623           s[out_pos] = ' ';
624           length += 4;
625           break;
626         case('R'):
627         case('r'):
628           // .-. [space]
629           s[out_pos] = '.';
630           s[out_pos+1] = '-';
631           s[out_pos+2] = '.';
632           s[out_pos+3] = ' ';
633           out_pos += 4;
634           break;
635         case('S'):
636         case('s'):
637           // ... [space]
638           s[out_pos] = '.';
639           s[out_pos+1] = '.';
640           s[out_pos+2] = '.';
641           s[out_pos+3] = ' ';
642           out_pos += 4;
643           break;
644         case('T'):
645         case('t'):
646           // - [space]
```

```
647             s[out_pos] = '-';
648             s[out_pos+1] = ' ';
649             out_pos += 2;
650             break;
651           case('U'):
652           case('u'):
653             // ..- [space]
654             s[out_pos] = '.';
655             s[out_pos+1] = '.';
656             s[out_pos+2] = '-';
657             s[out_pos+3] = ' ';
658             out_pos += 4;
659             break;
660           case('V'):
661           case('v'):
662             // ...- [space]
663             s[out_pos] = '.';
664             s[out_pos+1] = '.';
665             s[out_pos+2] = '.';
666             s[out_pos+3] = '-';
667             s[out_pos+4] = ' ';
668             out_pos += 5;
669             break;
670           case('W'):
671           case('w'):
672             // .-- [space]
673             s[out_pos] = '.';
674             s[out_pos+1] = '-';
675             s[out_pos+2] = '-';
676             s[out_pos+3] = ' ';
677             out_pos += 4;
678             break;
679           case('X'):
680           case('x'):
681             // -..- [space]
682             s[out_pos] = '-';
683             s[out_pos+1] = '.';
684             s[out_pos+2] = '.';
685             s[out_pos+3] = '-';
686             s[out_pos+4] = ' ';
687             out_pos += 5;
688             break;
689           case('Y'):
690           case('y'):
691             // -.-- [space]
692             s[out_pos] = '-';
693             s[out_pos+1] = '.';
694             s[out_pos+2] = '-';
695             s[out_pos+3] = '-';
696             s[out_pos+4] = ' ';
697             out_pos += 5;
698             break;
```

```
699         case('Z'):
700         case('z'):
701           // --.. [space]
702           s[out_pos] = '-';
703           s[out_pos+1] = '-';
704           s[out_pos+2] = '.';
705           s[out_pos+3] = '.';
706           s[out_pos+4] = ' ';
707           out_pos += 5;
708           break;
709         case('0'):
710           // ----- [space]
711           s[out_pos] = '-';
712           s[out_pos+1] = '-';
713           s[out_pos+2] = '-';
714           s[out_pos+3] = '-';
715           s[out_pos+4] = '-';
716           s[out_pos+5] = ' ';
717           out_pos += 6;
718           break;
719         case('1'):
720           // .---- [space]
721           s[out_pos] = '.';
722           s[out_pos+1] = '-';
723           s[out_pos+2] = '-';
724           s[out_pos+3] = '-';
725           s[out_pos+4] = '-';
726           s[out_pos+5] = ' ';
727           out_pos += 6;
728           break;
729         case('2'):
730           // ..--- [space]
731           s[out_pos] = '.';
732           s[out_pos+1] = '.';
733           s[out_pos+2] = '-';
734           s[out_pos+3] = '-';
735           s[out_pos+4] = '-';
736           s[out_pos+5] = ' ';
737           out_pos += 6;
738           break;
739         case('3'):
740           // ...-- [space]
741           s[out_pos] = '.';
742           s[out_pos+1] = '.';
743           s[out_pos+2] = '.';
744           s[out_pos+3] = '-';
745           s[out_pos+4] = '-';
746           s[out_pos+5] = ' ';
747           out_pos += 6;
748           break;
749         case('4'):
750           // ....- [space]
```

```
751             s[out_pos] = '.';
752             s[out_pos+1] = '.';
753             s[out_pos+2] = '.';
754             s[out_pos+3] = '.';
755             s[out_pos+4] = '-';
756             s[out_pos+5] = ' ';
757             out_pos += 6;
758             break;
759           case('5'):
760             // ..... [space]
761             s[out_pos] = '.';
762             s[out_pos+1] = '.';
763             s[out_pos+2] = '.';
764             s[out_pos+3] = '.';
765             s[out_pos+4] = '.';
766             s[out_pos+5] = ' ';
767             out_pos += 6;
768             break;
769           case('6'):
770             // -.... [space]
771             s[out_pos] = '-';
772             s[out_pos+1] = '.';
773             s[out_pos+2] = '.';
774             s[out_pos+3] = '.';
775             s[out_pos+4] = '.';
776             s[out_pos+5] = ' ';
777             out_pos += 6;
778             break;
779           case('7'):
780             // --... [space]
781             s[out_pos] = '-';
782             s[out_pos+1] = '-';
783             s[out_pos+2] = '.';
784             s[out_pos+3] = '.';
785             s[out_pos+4] = '.';
786             s[out_pos+5] = ' ';
787             out_pos += 6;
788             break;
789           case('8'):
790             // ---.. [space]
791             s[out_pos] = '-';
792             s[out_pos+1] = '-';
793             s[out_pos+2] = '-';
794             s[out_pos+3] = '.';
795             s[out_pos+4] = '.';
796             s[out_pos+5] = ' ';
797             out_pos += 6;
798             break;
799           case('9'):
800             // ----. [space]
801             s[out_pos] = '-';
802             s[out_pos+1] = '-';
```

```
803             s[out_pos+2] = '-';
804             s[out_pos+3] = '-';
805             s[out_pos+4] = '.';
806             s[out_pos+5] = ' ';
807             out_pos += 6;
808             break;
809           default:
810             // Ignore unknown characters
811             break;
812         }
813       }
814       s[out_pos] = 0;
815
816       return s;
817     }
818
819     char* morse_decode(char* input, size_t inlength)
820     {
821
822       // Check the input makes sense.
823       size_t length = 0;
824       for(size_t i = 0; i < inlength; i++) {
825         switch(input[i]) {
826           case('.'):
827           case('-'):
828             break;
829           case(' '):
830             length++;
831             break;
832           default:
833             // Invalid char, fail.
834             return NULL;
835         }
836       }
837
838       char* s = malloc(sizeof(char) * (length + 1));
839       if(!s) {
840         return NULL;
841       }
842
843       size_t pos = 0;
844       char* token = strtok(input, " ");
845       while(token != NULL) {
846
847         if(strcmp(token, ".-") == 0) {
848           s[pos] = 'A';
849           pos++;
850         } else
851         if(strcmp(token, "-...") == 0) {
852           s[pos] = 'B';
853           pos++;
854         } else
```

```
855         if(strcmp(token, "-.-.") == 0) {
856           s[pos] = 'C';
857           pos++;
858         } else
859         if(strcmp(token, "-..") == 0) {
860           s[pos] = 'D';
861           pos++;
862         } else
863         if(strcmp(token, ".") == 0) {
864           s[pos] = 'E';
865           pos++;
866         } else
867         if(strcmp(token, "..-.") == 0) {
868           s[pos] = 'F';
869           pos++;
870         } else
871         if(strcmp(token, "--.") == 0) {
872           s[pos] = 'G';
873           pos++;
874         } else
875         if(strcmp(token, "....") == 0) {
876           s[pos] = 'H';
877           pos++;
878         } else
879         if(strcmp(token, "..") == 0) {
880           s[pos] = 'I';
881           pos++;
882         } else
883         if(strcmp(token, ".---") == 0) {
884           s[pos] = 'J';
885           pos++;
886         } else
887         if(strcmp(token, "-.-") == 0) {
888           s[pos] = 'K';
889           pos++;
890         } else
891         if(strcmp(token, ".-..") == 0) {
892           s[pos] = 'L';
893           pos++;
894         } else
895         if(strcmp(token, "--") == 0) {
896           s[pos] = 'M';
897           pos++;
898         } else
899         if(strcmp(token, "-.") == 0) {
900           s[pos] = 'N';
901           pos++;
902         } else
903         if(strcmp(token, "---") == 0) {
904           s[pos] = 'O';
905           pos++;
906         } else
```

```
907         if(strcmp(token, ".--.") == 0) {
908           s[pos] = 'P';
909           pos++;
910         } else
911         if(strcmp(token, "--.-") == 0) {
912           s[pos] = 'Q';
913           pos++;
914         } else
915         if(strcmp(token, ".-.") == 0) {
916           s[pos] = 'R';
917           pos++;
918         } else
919         if(strcmp(token, "...") == 0) {
920           s[pos] = 'S';
921           pos++;
922         } else
923         if(strcmp(token, "-") == 0) {
924           s[pos] = 'T';
925           pos++;
926         } else
927         if(strcmp(token, "..-") == 0) {
928           s[pos] = 'U';
929           pos++;
930         } else
931         if(strcmp(token, "...-") == 0) {
932           s[pos] = 'V';
933           pos++;
934         } else
935         if(strcmp(token, ".--") == 0) {
936           s[pos] = 'W';
937           pos++;
938         } else
939         if(strcmp(token, "-..-") == 0) {
940           s[pos] = 'X';
941           pos++;
942         } else
943         if(strcmp(token, "-.--") == 0) {
944           s[pos] = 'Y';
945           pos++;
946         } else
947         if(strcmp(token, "--..") == 0) {
948           s[pos] = 'Z';
949           pos++;
950         } else
951         if(strcmp(token, ".----") == 0) {
952           s[pos] = '1';
953           pos++;
954         } else
955         if(strcmp(token, "..---") == 0) {
956           s[pos] = '2';
957           pos++;
958         } else
```

```
959         if(strcmp(token, "...--") == 0) {
960           s[pos] = '3';
961           pos++;
962         } else
963         if(strcmp(token, "....-") == 0) {
964           s[pos] = '4';
965           pos++;
966         } else
967         if(strcmp(token, ".....") == 0) {
968           s[pos] = '5';
969           pos++;
970         } else
971         if(strcmp(token, "-....") == 0) {
972           s[pos] = '6';
973           pos++;
974         } else
975         if(strcmp(token, "--...") == 0) {
976           s[pos] = '7';
977           pos++;
978         } else
979         if(strcmp(token, "---..") == 0) {
980           s[pos] = '8';
981           pos++;
982         } else
983         if(strcmp(token, "----.") == 0) {
984           s[pos] = '9';
985           pos++;
986         } else
987         if(strcmp(token, "-----") == 0) {
988           s[pos] = '0';
989           pos++;
990         } else
991
992         {
993           // This should be unreachable!
994           free(s);
995           return NULL;
996         }
997
998         token = strtok(NULL, " ");
999       }
1000
1001      s[pos] = 0;
1002
1003      return s;
1004
1005    }
1006
1007    // TODO: ROT13
1008
1009    // TODO: ITA2
1010
```

```
#endif

#ifndef EVIL_NO_FLOW
  // Included by default

  #define then ){
  #define end }
  #define If if(
  #define Else } else {
  #define For for(
  #define While while(
  #define Do do{
  #define Switch(x) switch(x){
  #define Case(x) case x:
#endif

#ifdef EVIL_HASH
  #include <stdint.h>
  #include <stddef.h>

  #pragma GCC diagnostic push
  #pragma GCC diagnostic ignored "-Wsign-conversion"
  #pragma GCC diagnostic ignored "-Wconversion"

  #ifdef INT64_MAX
  uint64_t jenkins64(char* key, size_t length) {
    size_t i = 0;
    uint64_t hash = 0;
    while(i != length) {
      hash += (uint64_t)key[i++];
      hash += hash << 10;
      hash += hash >> 6;
    }
    hash += hash << 3;
    hash ^= hash >> 11;
    hash += hash << 15;
    return hash;
  }
  uint64_t fletcher64(char* key, size_t length) {
    uint64_t a = 0;
    uint64_t b = 0;
    for(size_t i = 0; i < length; i++) {
      a = (a + (uint64_t)key[i]) % 4294967295;
      b = (b + a) % 4294967295;
    }
    return (b << 16) | a;
  }
  #endif

  uint32_t jenkins32(char* key, size_t length) {
    size_t i = 0;
    uint32_t hash = 0;
```

```
    while(i != length) {
      hash += key[i++];
      hash += hash << 10;
      hash += hash >> 6;
    }
    hash += hash << 3;
    hash ^= hash >> 11;
    hash += hash << 15;
    return hash;
  }
  uint32_t adler32(char* key, size_t length) {
    uint32_t sa = 1;
    uint32_t sb = 0;
    for(size_t i = 0; i < length; i++) {
      sa = (sa + (uint32_t)key[i]) % 65521;
      sb = (sb + sa) % 65521;
    }
    return (sb << 16) | sa;
  }
  uint32_t fletcher32(char* key, size_t length) {
    uint32_t a = 0;
    uint32_t b = 0;
    for(size_t i = 0; i < length; i++) {
      a = (a + (uint32_t)key[i]) % 65535;
      b = (b + a) % 65535;
    }
    return (b << 16) | a;
  }
  uint16_t fletcher16(char* key, size_t length) {
    uint16_t a = 0;
    uint16_t b = 0;
    for(size_t i = 0; i < length; i++) {
      a = (a + (uint16_t)key[i]) % 255;
      b = (b + a) % 255;
    }
    return (b << 16) | a;
  }

  #pragma GCC diagnostic pop

#endif
#ifdef EVIL_HELP
  // Excluded by default: Includes a lot of strings.


  #include <stdio.h>
  #include <string.h>

  #ifndef EVIL_NO_HELP_MANUAL
  void evil_manual(void);
  #endif

```

```
1115  void evil_explain(const char* token);
1116
1117  #ifndef EVIL_NO_WARN
1118  #warning "Not Yet Fully Implemented"
1119  #endif
1120
1121  #ifndef EVIL_NO_HELP_MANUAL
1122  void evil_manual(void) {
1123    printf("# CNoEvil v%s\n\n", CNOEVIL);
1124    printf("Using hash version: %s\n\n", EVIL_HASH_VER);
1125
1126    printf("%s\n\n", "CNoEvil abuses the hell out of the C pre-processor, and other C
      ↪ language features, to create a language that is still technically C, but looks and
      ↪ behaves differently, whilst remaining fully compatible with C.");
1127
1128    printf("%s", "\n---\n\n");
1129    printf("%s\n\n", "# Expected Behaviour");
1130    printf("%s\n\n", "Some definitions can produce warnings. Hide these by defining
      ↪ `EVIL_NO_WARN` before including `evil.h`.");
1131    printf("%s\n\n", "Some definitions can produce errors. There is no option to hide
      ↪ these.");
1132    printf("%s\n\n", "Definitions are expected to be created before the `evil.h` file is
      ↪ included.");
1133    printf("%s\n\n", "e.g.");
1134
1135    printf("%s\n\n", "    #define EVIL_HASH");
1136    printf("%s\n\n", "    #include \"evil.h\"");
1137
1138    printf("%s", "\n---\n\n");
1139    printf("%s\n\n", "# Libraries Available by Default:");
1140    printf("%s\n\n", "* Evil_Bool (Exclude by defining `EVIL_NO_BOOL` before including
      ↪ evil.h)");
1141    printf("%s\n\n", "* Evil_Comment (Exclude by defining `EVIL_NO_COMMENT` before including
      ↪ evil.h)");
1142    printf("%s\n\n", "* Evil_Flow (Exclude by defining `EVIL_NO_FLOW` before including
      ↪ evil.h)");
1143    printf("%s\n\n", "* Evil_Int (Exclude by defining `EVIL_NO_INT` before including
      ↪ evil.h)");
1144    printf("%s\n\n", "* Evil_IO (Exclude by defining `EVIL_NO_IO` before including
      ↪ evil.h)");
1145    printf("%s\n\n", "* Evil_Klass (Exclude by defining `EVIL_NO_KLASS` before including
      ↪ evil.h)");
1146    printf("%s\n\n", "* Evil_Main (Exclude by defining `EVIL_NO_MAIN` before including
      ↪ evil.h)");
1147    printf("%s\n\n", "* Evil_Proc (Exclude by defining `EVIL_NO_PROC` before including
      ↪ evil.h)");
1148    printf("%s\n\n", "* Evil_Specifier (Exclude by defining `EVIL_NO_SPECIFIER` before
      ↪ including evil.h)");
1149    printf("%s\n\n", "* Evil_Structures (Exclude by defining `EVIL_NO_STRUCT` before
      ↪ including evil.h)");
1150    printf("%s", "\n---\n\n");
1151
```

```
1152     printf("%s\n\n", "# Other Libraries Available:");
1153     printf("%s\n\n", "* Evil Assert (Import be defining `EVIL_ASSERT` before including
     ↪ evil.h)");
1154     printf("%s\n\n", "* Evil Bit (Import by defining `EVIL_BIT` before including evil.h)");
1155     printf("%s\n\n", "* Evil Cli (Import be defining `EVIL_CLI` before including evil.h)");
1156     printf("%s\n\n", "* Evil Coroutine (Import be defining `EVIL_COROUTINE` before including
     ↪ evil.h)");
1157     printf("%s\n\n", "* Evil Encode (Import by defining `EVIL_ENCODE` before including
     ↪ evil.h)");
1158     printf("%s\n\n", "* Evil Hash (Import be defining `EVIL_HASH` before including
     ↪ evil.h)");
1159     printf("%s\n\n", "* Evil Help (Import be defining `EVIL_HELP` before including
     ↪ evil.h)");
1160     printf("%s\n\n", "* Evil Lambda (Import be defining `EVIL_LAMBDA` before including
     ↪ evil.h)");
1161     printf("%s\n\n", "* Evil List (Import be defining `EVIL_LIST` before including
     ↪ evil.h)");
1162     printf("%s\n\n", "* Evil Log (Import be defining `EVIL_LOG` before including evil.h)");
1163     printf("%s\n\n", "* Evil Malloc (Import be defining `EVIL_MALLOC` before including
     ↪ evil.h)");
1164     printf("%s\n\n", "* Evil Math (Import be defining `EVIL_MATH` before including
     ↪ evil.h)");
1165     printf("%s\n\n", "* Evil Random (Import be defining `EVIL_RANDOM` before including
     ↪ evil.h)");
1166 
1167     printf("%s", "\n---\n\n");
1168     printf("%s\n\n", "# Libraries:");
1169 
1170     printf("%s", "\n---\n\n");
1171     printf("%s\n\n", "## Evil Bool");
1172     printf("%s\n\n", "Exclude by defining `EVIL_NO_BOOL` before including evil.h");
1173     printf("%s\n\n", "This library ensures that the identifies `true`, `false` and `bool`
     ↪ are defined.");
1174     printf("%s\n\n", "It can be thought of as an equivalent to stdbool.h");
1175 
1176     printf("%s", "\n---\n\n");
1177     printf("%s\n\n", "## Evil Comment");
1178     printf("%s\n\n", "Exclude by defining `EVIL_NO_COMMENT` before including evil.h");
1179     printf("%s\n\n", "Allows you to use the `comment(...)` syntax for creating comments.");
1180     printf("%s\n\n", "### Example");
1181 
1182     printf("%s\n\n", "    comment(1 + 2 = 3);");
1183 
1184     printf("%s\n\n", "`comment` can take any valid identifier. You may want to use strings
     ↪ normally.");
1185 
1186     printf("%s", "\n---\n\n");
1187     printf("%s\n\n", "## Evil Flow");
1188     printf("%s\n\n", "Exclude by defining `EVIL_NO_FLOW` before including evil.h");
1189     printf("%s\n\n", "Defines the keywords for most of CNoEvil's syntax.");
1190     printf("%s\n\n", "* then - A keyword, used to follow some constructs (such as If, While,
     ↪ etc.)");
```

```
1191    printf("%s\n\n", "* end - A keyword, used to close functions, and some other
    ↪ constructs.");
1192    printf("%s\n\n", "* If - A keyword, used to replace C's bracket'd `if`. i.e. Equivalent
    ↪ to `if(`.");
1193    printf("%s\n\n", "* For - A keyword, used to replace C's bracket'd `for`. i.e.
    ↪ Equivalent to `for(`.");
1194    printf("%s\n\n", "* While - A keyword, used to replace C's bracket'd `while`. i.e.
    ↪ Equivalent to `while(`.");
1195    printf("%s\n\n", "* Do - A keyword, replaces C's `do`, and opens the block
    ↪ automatically.");
1196    printf("%s\n\n", "* Switch(T) - A macro, creates and opens the block of a switch
    ↪ statement.");
1197    printf("%s\n\n", "### Example");
1198
1199    printf("%s\n", "    If 1 + 2 == 3 then");
1200    printf("%s\n", "        return 1;");
1201    printf("%s\n", "    Else");
1202    printf("%s\n", "        return 0;");
1203    printf("%s\n\n", "    end");
1204
1205    printf("%s", "\n---\n\n");
1206    printf("%s\n\n", "## Evil Int");
1207    printf("%s\n\n", "Exclude by defining `EVIL_NO_INT` before including evil.h");
1208    printf("%s\n\n", "Defines up to two type specifiers:");
1209    printf("%s\n\n", "* Number (if `int64_t` is supported). Equivalent to `int64_t`.");
1210    printf("%s\n\n", "* Decimal. Equivalent to `long double`.");
1211    printf("%s\n\n", "Defines up to two macros:");
1212    printf("%s\n\n", "* MaxNumber (if `int64_t` is supported). Equivalent to `INT64_MAX`.");
1213    printf("%s\n\n", "* MaxDecimal. Equivalent to `LDBL_MAX`.");
1214    printf("%s\n\n", "### Example");
1215
1216    printf("%s\n", "    Decimal(y);");
1217    printf("%s\n\n", "    Number(x) = 12;");
1218
1219    printf("%s", "\n---\n\n");
1220    printf("%s\n\n", "## Evil IO");
1221    printf("%s\n\n", "Exclude by defining `EVIL_NO_IO` before including evil.h");
1222    printf("%s\n\n", "display(T) - Prints a representation of the given value to stdout.");
1223    printf("%s\n\n", "displayf(F, T) - Prints a representation of the given value to F, a
    ↪ `FILE*`.");
1224    printf("%s\n\n", "displayln(T) - Prints a representation of the given value to stdout,
    ↪ followed by a system-compatible line ending.");
1225    printf("%s\n\n", "displayfln(F, T) - Prints a representation of the given value to F, a
    ↪ `FILE*`, followed by a system-compatible line ending.");
1226    printf("%s\n\n", "endl - A keyword. Prints a system-comaptible line ending to stdout.");
1227    printf("%s\n\n", "endlf(F) - Prints a system-comaptible line ending to F, a `FILE*`.");
1228    printf("%s\n\n", "repr_type(T) - Returns a `char*` containing a text representation of
    ↪ the type. Optimisation may effect results. Returns \"Unknown\" for any type that cannot
    ↪ be accounted for.");
1229    printf("%s\n\n", "### Example");
1230    printf("%s\n", "    displayln(\"Hello, World!\");");
1231    printf("%s\n", "    displayln(213);");
```

```
1232     printf("%s\n\n", "    displayln(repr_type(\"Hello, World!\"));");
1233
1234     printf("%s", "\n---\n\n");
1235     printf("%s\n\n", "## Evil Main");
1236     printf("%s\n\n", "Exclude by defining `EVIL_NO_MAIN` before including evil.h");
1237     printf("%s\n\n", "* Main - A keyword, expected to be followed by keyword `then`.
     ↪ Automatically makes argc and argv available. Use instead of the `main` function. Exclude
     ↪ by defining `EVIL_NO_MAIN`.");
1238     printf("%s\n\n", "### Example");
1239
1240     printf("%s\n\n", "    #include \"evil.h\"");
1241     printf("%s\n", "    Main then");
1242     printf("%s\n", "      displayln(\"Hello, World!\");");
1243     printf("%s\n\n", "    end");
1244
1245     printf("%s", "\n---\n\n");
1246     printf("%s\n\n", "## Evil Proc");
1247     printf("%s\n\n", "Exclude by defining `EVIL_NO_PROC` before including evil.h");
1248     printf("%s\n\n", "Introduces two syntactic elements:");
1249     printf("%s\n\n", "* declare(Name, ReturnType, ...); - A variadic macro. Arguments are as
     ↪ in C function arguments. Declares a C function.");
1250     printf("%s\n\n", "* proc(Name, ReturnType, ...) - A variadic macro. Arguments are as in
     ↪ C function arguments. Creates the start of a C function, that is, it is followed by a
     ↪ function body.");
1251     printf("%s\n\n", "### Example");
1252
1253       printf("%s\n\n","    #include \"evil.h\"");
1254       printf("%s\n\n","    declare(add2, int, int a);");
1255       printf("%s\n","    proc(add2, int, int a)");
1256       printf("%s\n","      return a + 2;");
1257       printf("%s\n\n","    end");
1258       printf("%s\n","    Main then");
1259       printf("%s\n","      displayln(add2(2));");
1260       printf("%s\n\n","    end");
1261
1262     printf("%s", "\n---\n\n");
1263     printf("%s\n\n", "## Evil Specifier");
1264     printf("%s\n\n", "Exclude by defining `EVIL_NO_SPECIFIER` before including evil.h");
1265     printf("%s\n\n", "contant(Type, Name, Value) - A macro, generates a `const`.");
1266     printf("%s\n\n", "storage(Type, Name, Value) - A macro, generates a `static`.");
1267     printf("%s\n\n", "### Example");
1268     printf("%s\n", "    constant(int, x, 12);");
1269     printf("%s\n\n", "    storage(int, y, 12);");
1270
1271     printf("%s", "\n---\n\n");
1272     printf("%s\n\n", "## Evil Structures");
1273     printf("%s\n\n", "Exclude by defining `EVIL_NO_STRUCT` before including evil.h");
1274     printf("%s\n\n", "Struct(Name) - Starts a struct definition.");
1275     printf("%s\n\n", "Union(Name) - Start a union definition.");
1276     printf("%s\n\n", "Typedef - A keyword, exactly equivalent to `typedef`.");
1277     printf("%s\n\n", "BitField(Name, Type, Width) - Used for defining a Bitfield inside a
     ↪ struct.");
```

```
1278     printf("%s\n\n", "### Example");
1279     printf("%s\n", "    Struct(Pair)");
1280     printf("%s\n", "        int a;");
1281     printf("%s\n", "        int b;");
1282     printf("%s\n\n", "    end;");
1283     printf("%s\n\n", "    struct Pair x = {1, 2};");
1284 
1285     printf("%s", "\n---\n\n");
1286     printf("%s\n\n", "## Evil Klass");
1287     printf("%s\n\n", "Exclude by defining `EVIL_NO_KLASS` before including evil.h");
1288     printf("%s\n\n", "This module is still under construction.");
1289     // TODO
1290 
1291     printf("%s", "\n---\n\n");
1292     printf("%s\n\n", "## Evil Assert");
1293     printf("%s\n\n", "Import by defining `EVIL_ASSERT` before including evil.h");
1294     printf("%s\n\n", "Defines two identifiers:");
1295     printf("%s\n\n", "* Assert(statement);");
1296     printf("%s\n\n", "* AssertMsg(statement, reason);");
1297     printf("%s\n\n", "If NDEBUG is defined, they become no-ops.");
1298     printf("%s\n\n", "Otherwise, if the statement is true, the halt the program and spit out
     ↪ a helpful trace.");
1299 
1300     printf("%s\n\n", "### Example");
1301     printf("%s\n", "    Assert(false == true);");
1302     printf("%s\n\n", "    AssertMsg(false == true, \"This is impossible.\");");
1303 
1304     printf("%s", "\n---\n\n");
1305     printf("%s\n\n", "## Evil Cli");
1306     printf("%s\n\n", "Import by defining `EVIL_CLI` before including evil.h");
1307     printf("%s\n\n", "This module provides a variety of functions for working with the
     ↪ terminal, by using ANSI escape sequences.");
1308     printf("%s\n\n", "### Warnings");
1309     printf("%s\n\n", "* If EVIL_ASSERT is not defined, then cli_fg_256 and cli_bg_256 will
     ↪ be unavailable.");
1310     printf("%s\n\n", "* None of the module's functions check if the function is supported by
     ↪ current terminal.");
1311     printf("%s\n\n", "### Definitions");
1312     printf("%s\n\n", "* cli_reset() - Removes any active effects and colors from the
     ↪ terminal.");
1313     printf("%s\n\n", "* cli_fg_256(N) - Takes a number between 0-255, and applies the
     ↪ corresponding color to the terminal text.");
1314     printf("%s\n\n", "* cli_bg_256(N) - Takes a number between 0-255, and applies the
     ↪ corresponding color to the terminal background.");
1315     printf("%s\n\n", "* cli_fg_rgb(R,G,B) - Takes three values (Red, Green, Blue), each of
     ↪ which is a number in the range 0-255, and applies the corresponding truecolor to the
     ↪ terminal text.");
1316     printf("%s\n\n", "* cli_bg_rgb(R,G,B) - Takes three values (Red, Green, Blue), each of
     ↪ which is a number in the range 0-255, and applies the corresponding truecolor to the
     ↪ terminal background.");
1317     printf("%s\n\n", "* cli_cursor_up(N) - Moves the console cursor up N lines.");
1318     printf("%s\n\n", "* cli_cursor_down(N) - Moves the console cursor down N lines.");
```

```
1319     printf("%s\n\n", "* cli_cursor_right(N) - Moves the console cursor right N
     ↪ characters.");
1320     printf("%s\n\n", "* cli_cursor_left(N) - Moves the console cursor left N characters.");
1321     printf("%s\n\n", "* cli_cursor_save() - Saves the current cursor position.");
1322     printf("%s\n\n", "* cli_cursor_restore() - Moves the cursor to the last saved
     ↪ position.");
1323     printf("%s\n\n", "* cli_screen_clear() - Clears the current terminal screen.");
1324     printf("%s\n\n", "* cli_screen_clear_before() - Clears the current terminal screen,
     ↪ before the cursor.");
1325     printf("%s\n\n", "* cli_screen_clear_after() - Clears the current terminal screen, after
     ↪ the cursor.");
1326     printf("%s\n\n", "* cli_line_clear() - Clears the current terminal line.");
1327     printf("%s\n\n", "* cli_line_clear_before() - Clears the current terminal line, before
     ↪ the cursor.");
1328     printf("%s\n\n", "* cli_line_clear_after() - Clears the current terminal line, after the
     ↪ cursor.");
1329     printf("%s\n\n", "* cli_effect_reset() - Removes any active effects from the
     ↪ terminal.");
1330     printf("%s\n\n", "* cli_effect_bold() - Activates the bold text terminal effect.");
1331     printf("%s\n\n", "* cli_effect_underline() - Activates the underlined text terminal
     ↪ effect.");
1332     printf("%s\n\n", "* cli_effect_reverse() - Activates the reversed text terminal
     ↪ effect.");
1333     printf("%s\n\n", "* cli_effect_blink() - Activates the blinking text terminal effect.
     ↪ (Often disabled on modern terminals).");
1334     printf("%s\n\n", "* cli_effect_invisible() - Activates the invisible text terminal
     ↪ effect.");
1335     printf("%s\n\n", "* cli_fg_black() - Sets the terminal text to simple black.");
1336     printf("%s\n\n", "* cli_fg_red() - Sets the terminal text to simple red.");
1337     printf("%s\n\n", "* cli_fg_green() - Sets the terminal text to simple green.");
1338     printf("%s\n\n", "* cli_fg_yellow() - Sets the terminal text to simple yellow.");
1339     printf("%s\n\n", "* cli_fg_blue() - Sets the terminal text to simple blue.");
1340     printf("%s\n\n", "* cli_fg_magenta() - Sets the terminal text to simple magenta.");
1341     printf("%s\n\n", "* cli_fg_cyan() - Sets the terminal text to simple cyan.");
1342     printf("%s\n\n", "* cli_fg_white() - Sets the terminal text to simple white.");
1343     printf("%s\n\n", "* cli_fg_bright_black() - Sets the terminal text to complex black.");
1344     printf("%s\n\n", "* cli_fg_bright_red() - Sets the terminal text to complex red.");
1345     printf("%s\n\n", "* cli_fg_bright_green() - Sets the terminal text to complex green.");
1346     printf("%s\n\n", "* cli_fg_bright_yellow() - Sets the terminal text to complex
     ↪ yellow.");
1347     printf("%s\n\n", "* cli_fg_bright_blue() - Sets the terminal text to complex blue.");
1348     printf("%s\n\n", "* cli_fg_bright_magenta() - Sets the terminal text to complex
     ↪ magenta.");
1349     printf("%s\n\n", "* cli_fg_bright_cyan() - Sets the terminal text to complex cyan.");
1350     printf("%s\n\n", "* cli_fg_bright_white() - Sets the terminal text to complex white.");
1351     printf("%s\n\n", "* cli_bg_black() - Sets the terminal background to simple black.");
1352     printf("%s\n\n", "* cli_bg_red() - Sets the terminal background to simple red.");
1353     printf("%s\n\n", "* cli_bg_green() - Sets the terminal background to simple green.");
1354     printf("%s\n\n", "* cli_bg_yellow() - Sets the terminal background to simple yellow.");
1355     printf("%s\n\n", "* cli_bg_blue() - Sets the terminal background to simple blue.");
1356     printf("%s\n\n", "* cli_bg_magenta() - Sets the terminal background to simple
     ↪ magenta.");
```

```
1357      printf("%s\n\n", "* cli_bg_cyan() - Sets the terminal background to simple cyan.");
1358      printf("%s\n\n", "* cli_bg_white() - Sets the terminal background to simple white.");
1359      printf("%s\n\n", "* cli_bg_bright_black() - Sets the terminal background to complex
      ↪ black.");
1360      printf("%s\n\n", "* cli_bg_bright_red() - Sets the terminal background to complex
      ↪ red.");
1361      printf("%s\n\n", "* cli_bg_bright_green() - Sets the terminal background to complex
      ↪ green.");
1362      printf("%s\n\n", "* cli_bg_bright_yellow() - Sets the terminal background to complex
      ↪ yellow.");
1363      printf("%s\n\n", "* cli_bg_bright_blue() - Sets the terminal background to complex
      ↪ blue.");
1364      printf("%s\n\n", "* cli_bg_bright_magenta() - Sets the terminal background to complex
      ↪ magenta.");
1365      printf("%s\n\n", "* cli_bg_bright_cyan() - Sets the terminal background to complex
      ↪ cyan.");
1366      printf("%s\n\n", "* cli_bg_bright_white() - Sets the terminal background to complex
      ↪ white.");
1367      printf("%s\n\n", "## Example");
1368      printf("%s\n\n", "```");
1369      printf("%s\n", "    #define EVIL_IO");
1370      printf("%s\n", "    #define EVIL_CLI");
1371      printf("%s\n", "    #include \"evil.h\"");
1372      printf("\n");
1373      printf("%s\n", "    Main then");
1374      printf("%s\n", "      cli_fg_magenta();");
1375      printf("%s\n", "      cli_bg_white();");
1376      printf("%s\n", "      displayln(\"Coloured text!\");");
1377      printf("%s\n", "      cli_reset();");
1378      printf("%s\n", "    end");
1379      printf("%s\n\n", "```");
1380
1381      printf("%s", "\n---\n\n");
1382      printf("%s\n\n", "## Evil Coroutine");
1383      printf("%s\n\n", "Import by defining `EVIL_COROUTINE` before including evil.h");
1384      printf("%s\n\n", "### Warnings:");
1385      printf("%s\n\n", "* Lambda's don't play well with Coroutines. Avoid using them in the
      ↪ body of a coroutine.");
1386      printf("%s\n\n", "* Coroutine's don't support nesting. It may work sometimes, other
      ↪ times it may explode.");
1387      printf("%s\n\n", "### Definitions");
1388      printf("%s\n\n", "coroutine() - A macro that marks the beginning of a coroutine body.");
1389      printf("%s\n\n", "co_return(T) - A macro that marks a return from a coroutine.");
1390      printf("%s\n\n", "co_end() - A macro that closes a coroutine body.");
1391      printf("%s\n\n", "### Example");
1392      printf("%s\n", "```");
1393      printf("%s\n", "    #define EVIL_IO");
1394      printf("%s\n", "    #define EVIL_COROUTINE");
1395      printf("%s\n", "    #include \"evil.h\"");
1396      printf("%s\n", "    proc(example, int)");
1397      printf("%s\n", "    storage(int, i, 0);");
1398      printf("%s\n", "    coroutine();");
```

```
1399      printf("%s\n", "    While true then");
1400      printf("%s\n", "      co_return(++i);");
1401      printf("%s\n", "    end");
1402      printf("%s\n", "    co_end();");
1403      printf("%s\n", "    return i;");
1404      printf("%s\n", "  end");
1405      printf("\n\n");
1406      printf("%s\n", "  Main then");
1407      printf("%s\n", "    displayln(example());");
1408      printf("%s\n", "    displayln(example());");
1409      printf("%s\n", "    displayln(example());");
1410      printf("%s\n", "    displayln(example());");
1411      printf("%s\n", "    displayln(example());");
1412      printf("%s\n", "    displayln(example());");
1413      printf("%s\n", "    displayln(example());");
1414      printf("%s\n", "    displayln(example());");
1415      printf("%s\n", "  end");
1416      printf("%s\n\n", "```");
1417
1418      printf("%s", "\n---\n\n");
1419      printf("%s\n\n", "## Evil Hash");
1420      printf("%s\n\n", "Import by defining `EVIL_HASH` before including evil.h");
1421      printf("%s\n\n", "### Warnings");
1422      printf("%s\n\n", "* if int64_t is unavailable, then jenkins64, fletcher64 won't be
      ↪ defined.");
1423      printf("%s\n\n", "* These are _not_ cryptographic safe hashes.");
1424      printf("%s\n\n", "### Definitions");
1425      printf("%s\n\n", "* jenkins64(char* key, size_t length) - Hash a given string into a
      ↪ uint64_t. Based on Jenkins One-At-A-Time hash.");
1426      printf("%s\n\n", "* jenkins32(char* key, size_t length) - Hash a given string into a
      ↪ uint32_t. Based on Jenkins One-At-A-Time hash.");
1427
1428      printf("%s\n\n", "* fletcher64(char* key, size_t length) - Hash a given string into a
      ↪ uint64_t. Based on Fletcher's checksum.");
1429      printf("%s\n\n", "* fletcher32(char* key, size_t length) - Hash a given string into a
      ↪ uint32_t. Based on Fletcher's checksum.");
1430      printf("%s\n\n", "* fletcher16(char* key, size_t length) - Hash a given string into a
      ↪ uint16_t. Based on Fletcher's checksum.");
1431
1432      printf("%s\n\n", "* adler32(char* key, size_t length) - Hash a given string into a
      ↪ uint32_t. Based on Adler-32.");
1433
1434      printf("%s\n\n", "### Example");
1435      printf("%s\n\n", "```");
1436      printf("%s\n", "    #define EVIL_HASH");
1437      printf("%s\n", "    #include \"evil.h\"");
1438      printf("%c", '\n');
1439      printf("%s\n", "    Main then");
1440      printf("%s\n", "      displayln(jenkins64(\"Hello, World!\", 12));");
1441      printf("%s\n", "      displayln(jenkins32(\"Hello, World!\", 12));");
1442      printf("%c", '\n');
1443      printf("%s\n", "      displayln(fletcher64(\"Hello, World!\", 12));");
```

```
1444     printf("%s\n", "      displayln(fletcher32(\"Hello, World!\", 12));");
1445     printf("%s\n", "      displayln(fletcher16(\"Hello, World!\", 12));");
1446     printf("%c", '\n');
1447     printf("%s\n", "      displayln(adler32(\"Hello, World!\", 12));");
1448     printf("%s\n", "    end");
1449     printf("%s\n\n", "```");
1450 
1451     printf("%s", "\n---\n\n");
1452     printf("%s\n\n", "## Evil Help");
1453     printf("%s\n\n", "Import by defining `EVIL_HELP` before including evil.h");
1454     printf("%s\n\n", "Defines two functions:");
1455     printf("%s\n\n", "* void evil_explain(const char* s)");
1456     printf("%s\n\n", "* void evil_manual(void)");
1457     printf("%s\n\n", "evil_manual prints this entire document to stdout.");
1458     printf("%s\n\n", "evil_explain looks up an identifier and prints some information about
     ↪ it to stdout.");
1459     printf("%s\n\n", "If `EVIL_NO_HELP_MANUAL` is defined, then evil_manual won't be
     ↪ created.");
1460 
1461     printf("%s", "\n---\n\n");
1462     printf("%s\n\n", "## Evil Lambda");
1463     printf("%s\n\n", "Import by defining `EVIL_LAMBDA` before including evil.h");
1464     printf("%s\n\n", "functionPointer = lambda(returnType, body)");
1465     printf("%s\n\n", "Lambda is a slightly more advanced feature, which requires the user
     ↪ how to construct and use function pointers.");
1466     printf("%s\n", "```");
1467     printf("%s\n\n", "### Example");
1468     printf("%s\n", "    #define EVIL_LAMBDA");
1469     printf("%s\n", "    #include \"evil.h\"");
1470     printf("%c", '\n');
1471     printf("%s\n", "    Main then");
1472     printf("%s\n", "      int (*max)(int, int) = lambda(int,");
1473     printf("%s\n", "                                    (int x, int y) {");
1474     printf("%s\n", "                                     return x > y ? x : y;");
1475     printf("%s\n", "                                    });");
1476     printf("%c", '\n');
1477     printf("%s\n", "      displayln(max(1, 2));");
1478     printf("%s\n", "    end");
1479     printf("%s\n\n", "```");
1480 
1481     printf("%s", "\n---\n\n");
1482     printf("%s\n\n", "## Evil Malloc");
1483     printf("%s\n\n", "Import by defining `EVIL_MALLOC` before including evil.h");
1484     printf("%s\n\n", "Provides checked_malloc:");
1485     printf("%s\n\n", "`checked_malloc(object, object_type, buffer, fail_msg, exit_q)`");
1486     printf("%s\n\n", "* object - The identifier being assigned to.");
1487     printf("%s\n\n", "* object_type - The type of the identifier being assigned to.");
1488     printf("%s\n\n", "* buffer - The size of the buffer to pass to malloc.");
1489     printf("%s\n\n", "* fail_msg - A string to print to stderr if it fails to allocate.");
1490     printf("%s\n\n", "* exit_q - A boolean. If true, terminates the program.");
1491     printf("%s\n\n", "### Example");
1492     printf("%s\n", "```");
```

```
1493     printf("%s\n","    #define EVIL_MALLOC");
1494     printf("%s\n","    #include \"evil.h\"");
1495     printf("%c", '\n');
1496     printf("%s\n","    Main then");
1497     printf("%s\n","      comment(\"This will exit if malloc fails.\");");
1498     printf("%s\n","      checked_malloc(x, char*, sizeof(char) * 6, \"Out of Memory\",
     ↪ true);");
1499     printf("%c", '\n');
1500     printf("%s\n","      x[0] = 'H';");
1501     printf("%s\n","      x[1] = 'e';");
1502     printf("%s\n","      x[2] = 'l';");
1503     printf("%s\n","      x[3] = 'l';");
1504     printf("%s\n","      x[4] = 'o';");
1505     printf("%s\n","      x[5] = '\0';");
1506     printf("%c", '\n');
1507     printf("%s\n","      displayln(x);");
1508     printf("%s\n","      free(x);");
1509     printf("%s\n","    end");
1510     printf("%s\n\n", "```");
1511 
1512     printf("%s", "\n---\n\n");
1513     printf("%s\n\n", "## Evil Math");
1514     printf("%s\n\n", "Import by defining `EVIL_MATH` before including evil.h");
1515     printf("%s\n\n", "### Definitions");
1516     printf("%s\n\n", "* add(a, b)");
1517     printf("%s\n\n", "* take(a, b)");
1518     printf("%s\n\n", "* multiply(a, b)");
1519     printf("%s\n\n", "* divide(a, b)");
1520     printf("%s\n\n", "* math.h");
1521 
1522     printf("%s", "\n---\n\n");
1523     printf("%s\n\n", "## Evil Random");
1524     printf("%s\n\n", "Import by defining `EVIL_RANDOM` before including evil.h");
1525     printf("%s\n\n", "### Warnings");
1526     printf("%s\n\n", "* `randomseed` and `random` use C-rand, which is not cryptographically
     ↪ strong.");
1527     printf("%s\n\n", "### Definitions");
1528     printf("%s\n\n", "* `randomseed(void)` - Seed the random generator.");
1529     printf("%s\n\n", "* `random(min, max)` - Get a random integer in a given range.");
1530 
1531     printf("%s", "\n---\n\n");
1532     printf("%s\n\n", "## Evil Log");
1533     printf("%s\n\n", "Import by defining `EVIL_LOG` before including evil.h");
1534     printf("%s\n\n", "### Warnings");
1535     printf("%s\n\n", "* `debug(char*)` is a no-op, unless `DEBUG_LOG` is defined.");
1536     printf("%s\n\n", "### Definitions");
1537     printf("%s\n\n", "* `message(char* )` \n\n Logs a message to stdout. If
     ↪ (char\\*)message_file has a length > 0, then logs to that file in append mode. If
     ↪ (char\\*)log_file has a length > 0, then logs to that file in append mode.");
1538     printf("%s\n\n", "* `warning(char* )` \n\n Logs a message to stdout. If
     ↪ (char\\*)warning_file has a length > 0, then logs to that file in append mode. If
     ↪ (char\\*)log_file has a length > 0, then logs to that file in append mode.");
```

```
1539      printf("%s\n\n", "* `critical(char* )` \n\n Logs a message to stderr. If
      ↪  (char\\*)critical_file has a length > 0, then logs to that file in append mode. If
      ↪  (char\\*)log_file has a length > 0, then logs to that file in append mode.");
1540      printf("%s\n\n", "* `error(char* )` \n\n Logs a message to stderr. If
      ↪  (char\\*)error_file has a length > 0, then logs to that file in append mode. If
      ↪  (char\\*)log_file has a length > 0, then logs to that file in append mode.");
1541      printf("%s\n\n", "* `info(char* )` \n\n Logs a message to stdout. If (char\\*)info_file
      ↪  has a length > 0, then logs to that file in append mode. If (char\\*)log_file has a
      ↪  length > 0, then logs to that file in append mode.");
1542      printf("%s\n\n", "* `debug(char* )` \n\n If DEBUG_LOG defined, logs a message to stdout.
      ↪  If (char\\*)debug_file has a length > 0, then logs to that file in append mode. If
      ↪  (char*)log_file has a length > 0, then logs to that file in append mode.");
1543      printf("%s\n\n", "* `debug(char* )` \n\n If DEBUG_LOG not defined, doesn't do
      ↪  anything.");
1544
1545      printf("%s", "\n---\n\n");
1546      printf("%s\n\n", "## Evil Bit");
1547      printf("%s\n\n", "Import by defining `EVIL_BIT` before including evil.h");
1548      printf("%s\n\n", "Adds worded bit operators.");
1549      printf("%s\n\n", "# Defines:");
1550      printf("%s\n\n", "* BIT_AND(a, b)");
1551      printf("%s\n\n", "* BIT_OR(a, b)");
1552      printf("%s\n\n", "* BIT_XOR(a, b)");
1553      printf("%s\n\n", "* BIT_NOT(a)");
1554      printf("%s\n\n", "* BIT_RSHIFT(a, b)");
1555      printf("%s\n\n", "* BIT_LSHIFT(a, b)");
1556
1557      printf("%s", "\n---\n\n");
1558      printf("%s\n\n", "## Evil Encode");
1559      printf("%s\n\n", "Import by defining `EVIL_ENCODE` before including evil.h");
1560      printf("%s\n\n", "This module is still under construction.");
1561      // TODO
1562    }
1563    #endif
1564
1565  void evil_explain(const char* token) {
1566    if(strncmp("EVIL_NO_WARN", token, 12) == 0) {
1567      printf("%s\n\n", "Silences CnoEvil compile-time warnings, when defined.");
1568      printf("%s\n\n", "Do not use unless you understand the full implications.");
1569    } else
1570    // EVIL_BOOL (default)
1571    if(strncmp("true", token, 6) == 0) {
1572      printf("%s\n\n", "bool - defined in `EVIL_BOOL`.");
1573    } else
1574    if(strncmp("false", token, 6) == 0) {
1575      printf("%s\n\n", "bool - defined in `EVIL_BOOL`.");
1576    } else
1577    if(strncmp("bool", token, 5) == 0) {
1578      printf("%s\n\n", "type - defined in `EVIL_BOOL`.");
1579    } else
1580    if(strncmp("__bool_true_false_are_defined", token, 27) == 0) {
1581      printf("%s\n\n", "Value of 1 - defined in `EVIL_BOOL`.");
```

```
1582    } else
1583    if(strncmp("EVIL_BOOL", token, 9) == 0) {
1584      printf("%s\n\n","module - Adds booleans.");
1585      printf("%s\n\n","Available by default.");
1586      printf("%s\n\n","Exclude by defining `EVIL_NO_BOOL`");
1587    } else
1588    if(strncmp("EVIL_NO_BOOL", token, 12) == 0) {
1589      printf("%s\n\n","Exclude the `EVIL_BOOL` module when defined.");
1590    } else
1591    // EVIL_COMMENT (default)
1592    if(strncmp("EVIL_COMMENT", token, 11) == 0) {
1593      printf("%s\n\n","module - Adds a comment syntax.");
1594      printf("%s\n\n","Available by default.");
1595      printf("%s\n\n","Exclude by defining `EVIL_NO_COMMENT`");
1596    } else
1597    if(strncmp("EVIL_NO_COMMENT", token, 15) == 0) {
1598      printf("%s\n\n","Exclude the `EVIL_COMMENT` module when defined.");
1599    } else
1600    if(strncmp("comment", token, 7) == 0) {
1601      printf("%s\n\n","`comment(statement);` - Make comments like normal functions.");
1602      printf("%s\n\n", "Defined in `EVIL_COMMENT`.");
1603    } else
1604    // EVIL_FLOW (default)
1605    if(strncmp("EVIL_FLOW", token, 10) == 0) {
1606      printf("%s\n\n","module - Adds a new syntax.");
1607      printf("%s\n\n","Available by default.");
1608      printf("%s\n\n","Exclude by defining `EVIL_NO_FLOW`");
1609    } else
1610    if(strncmp("EVIL_NO_FLOW", token, 12) == 0) {
1611      printf("%s\n\n","Exclude the `EVIL_FLOW` module when defined.");
1612    } else
1613    if(strncmp("then", token, 5) == 0) {
1614      printf("%s\n\n","`then` - Meant to follow most flow statements.");
1615      printf("%s\n\n", "Defined in `EVIL_FLOW`.");
1616    } else
1617    if(strncmp("end", token, 4) == 0) {
1618      printf("%s\n\n","`end` - Meant to finish most flow statements.");
1619      printf("%s\n\n", "Defined in `EVIL_FLOW`.");
1620    } else
1621    if(strncmp("If", token, 3) == 0) {
1622      printf("%s\n\n","`If` - Opens an if statement without the need for braces.");
1623      printf("%s\n\n", "Defined in `EVIL_FLOW`.");
1624    } else
1625    if(strncmp("Else", token, 5) == 0) {
1626      printf("%s\n\n","`Else` - Closes one body, injects `else`, and opens a new body.");
1627      printf("%s\n\n", "Defined in `EVIL_FLOW`.");
1628    } else
1629    if(strncmp("For", token, 4) == 0) {
1630      printf("%s\n\n","`For` - Opens a for statement without the need for braces.");
1631      printf("%s\n\n", "Defined in `EVIL_FLOW`.");
1632    } else
1633    if(strncmp("While", token, 6) == 0) {
```

```
1634      printf("%s\n\n","`While` - Opens a while statement without the need for braces.");
1635      printf("%s\n\n", "Defined in `EVIL_FLOW`.");
1636    } else
1637    if(strncmp("Do", token, 3) == 0) {
1638      printf("%s\n\n","`Do` - Opens a do statement without the need for braces.");
1639      printf("%s\n\n", "Defined in `EVIL_FLOW`.");
1640    } else
1641    if(strncmp("Switch", token, 7) == 0) {
1642      printf("%s\n\n","`Switch(x)` - Opens a switch statement without the need for a following
      ↪ brace.");
1643      printf("%s\n\n", "Defined in `EVIL_FLOW`.");
1644    } else
1645    if(strncmp("Case", token, 5) == 0) {
1646      printf("%s\n\n","`Case(x)` - Opens a case in a syntax-consistent way.");
1647      printf("%s\n\n", "Defined in `EVIL_FLOW`.");
1648    } else
1649    // EVIL_INT (default)
1650    if(strncmp("EVIL_INT", token, 9) == 0) {
1651      printf("%s\n\n","module - Adds number specifiers.");
1652      printf("%s\n\n","Available by default.");
1653      printf("%s\n\n","Exclude by defining `EVIL_NO_INT`");
1654    } else
1655    if(strncmp("EVIL_NO_INT", token, 11) == 0) {
1656      printf("%s\n\n","Exclude the `EVIL_INT` module when defined.");
1657    } else
1658    if(strncmp("Number", token, 7) == 0) {
1659      printf("%s\n\n","`Number(x)` - declare a number type. Equivalent to `int64_t`. If
      ↪ int64_t is not supported, this method is unavailable.");
1660      printf("%s\n\n", "Defined in `EVIL_INT`.");
1661    } else
1662    if(strncmp("Decimal", token, 8) == 0) {
1663      printf("%s\n\n","`Decimal(x)` - declare a float type. Equivalent to `long double`.");
1664      printf("%s\n\n", "Defined in `EVIL_INT`.");
1665    } else
1666    if(strncmp("MaxDecimal", token, 11) == 0) {
1667      printf("%s\n\n","`MaxDecimal` - The maximum value for a Decimal type.");
1668      printf("%s\n\n", "Defined in `EVIL_INT`.");
1669    } else
1670    if(strncmp("MaxNumber", token, 10) == 0) {
1671      printf("%s\n\n","`MaxNumber` - The maximum value for a Number type.");
1672      printf("%s\n\n", "Defined in `EVIL_INT`.");
1673    } else
1674    // EVIL_IO (default)
1675    if(strncmp("EVIL_IO", token, 7) == 0) {
1676      printf("%s\n\n","module - Adds IO helpers.");
1677      printf("%s\n\n","Available by default.");
1678      printf("%s\n\n","Exclude by defining `EVIL_NO_IO`");
1679    } else
1680    if(strncmp("EVIL_NO_IO", token, 10) == 0) {
1681      printf("%s\n\n","Exclude the `EVIL_IO` module when defined.");
1682    } else
1683    // display_format
```

```
1684   if(strncmp("display_format", token, 15) == 0) {
1685     printf("%s\n\n", "`display_format(x);`");
1686     printf("%s\n\n", "Creates a string that is probably a format specifier for the given
       ↪ object.");
1687     printf("%s\n\n", "Defined in `EVIL_IO`.");
1688   } else
1689   // display
1690   if(strncmp("display", token, 15) == 0) {
1691     printf("%s\n\n", "`display(x);`");
1692     printf("%s\n\n", "Prints a representation of the given object to stdout.");
1693     printf("%s\n\n", "Also see: displayf, displayln, displayfln, endl.");
1694     printf("%s\n\n", "Defined in `EVIL_IO`.");
1695   } else
1696   // displayf
1697   if(strncmp("displayf", token, 16) == 0) {
1698     printf("%s\n\n", "`displayf(file, x);`");
1699     printf("%s\n\n", "Prints a representation of the given object to the `FILE*` given as
       ↪ the first argument.");
1700     printf("%s\n\n", "Also see: display, displayln, displayfln, endl.");
1701     printf("%s\n\n", "Defined in `EVIL_IO`.");
1702   } else
1703   // displayln
1704   if(strncmp("displayln", token, 17) == 0) {
1705     printf("%s\n\n", "`displayln(x);`");
1706     printf("%s\n\n", "Prints a representation of the given object to stdout, followed by a
       ↪ system-dependant newline.");
1707     printf("%s\n\n", "Also see: displayf, display, displayfln, endl.");
1708     printf("%s\n\n", "Defined in `EVIL_IO`.");
1709   } else
1710   // displayfln
1711   if(strncmp("displayfln", token, 18) == 0) {
1712     printf("%s\n\n", "`displayfln(file, x);`");
1713     printf("%s\n\n", "Prints a representation of the given object to the `FILE*` given as
       ↪ the first argument, followed by a system-dependant newline.");
1714     printf("%s\n\n", "Also see: displayf, display, displayln, endl.");
1715     printf("%s\n\n", "Defined in `EVIL_IO`.");
1716   } else
1717   // repr_type
1718   if(strncmp("repr_type", token, 10) == 0) {
1719     printf("%s\n\n", "`repr_type(x)`");
1720     printf("%s\n\n", "Creates a string representation of the type that the object probably
       ↪ is.");
1721     printf("%s\n\n", "Defined in `EVIL_IO`.");
1722   } else
1723   // endl
1724   if(strncmp("endl", token, 5) == 0) {
1725     printf("%s\n\n", "`endl;`");
1726     printf("%s\n\n", "Sends a system-dependant newline to stdout.");
1727     printf("%s\n\n", "Also see: endlf");
1728     printf("%s\n\n", "Defined in `EVIL_IO`.");
1729   } else
1730   // endlf
```

```
1731    if(strncmp("endlf", token, 5) == 0) {
1732      printf("%s\n\n", "`endlf(file);`");
1733      printf("%s\n\n", "Sends a system-dependant newline to the `FILE*` given as the first
      ↪  argument.");
1734      printf("%s\n\n", "Also see: endl");
1735      printf("%s\n\n", "Defined in `EVIL_IO`.");
1736    } else
1737    // stdio
1738    if(strncmp("stdio", token, 6) == 0) {
1739      printf("%s\n\n", "The stdio module.");
1740      printf("%s\n\n", "Imported by `EVIL_IO`.");
1741    } else
1742    // EVIL_MAIN (default)
1743    if(strncmp("EVIL_MAIN", token, 9) == 0) {
1744      printf("%s\n\n","module - Adds `Main`.");
1745      printf("%s\n\n","Available by default.");
1746      printf("%s\n\n","Exclude by defining `EVIL_NO_MAIN`");
1747    } else
1748    if(strncmp("EVIL_NO_MAIN", token, 13) == 0) {
1749      printf("%s\n\n", "Excludes `EVIL_MAIN1 when defined.");
1750    } else
1751    if(strncmp("Main", token, 5) == 0) {
1752      printf("%s\n\n", "Opens the `main` function, and defines argc and argv.");
1753      printf("%s\n\n", "See also: argc, argv");
1754      printf("%s\n\n", "Defined in `EVIL_MAIN`.");
1755    } else
1756    if(strncmp("argc", token, 5) == 0) {
1757      printf("%s\n\n", "Automatically available under `Main`.");
1758      printf("%s\n\n", "An `int` representing the number of arguments given on the command
      ↪  line.");
1759      printf("%s\n\n", "See also: Main, argv");
1760    } else
1761    if(strncmp("argv", token, 5) == 0) {
1762      printf("%s\n\n", "Automatically available under `Main`.");
1763      printf("%s\n\n", "A `char*[]` containing the arguments given on the command line.");
1764      printf("%s\n\n", "See also: Main, argc");
1765    } else
1766    // EVIL_PROC (default)
1767    if(strncmp("EVIL_PROC", token, 9) == 0) {
1768      printf("%s\n\n", "module - adds `proc` and `declare`.");
1769      printf("%s\n\n", "Available by default.");
1770      printf("%s\n\n", "Exclude by defining `EVIL_NO_PROC`");
1771    } else
1772    if(strncmp("EVIL_NO_PROC", token, 13) == 0) {
1773      printf("%s\n\n", "Exclude `EVIL_PROC` when defined");
1774    } else
1775    if(strncmp("proc", token, 5) == 0) {
1776      printf("%s\n\n", "proc(name, return type, args...) - Opens the body of a function
      ↪  definition.");
1777      printf("%s\n\n", "Defined in `EVIL_PROC`.");
1778    } else
1779    if(strncmp("declare", token, 5) == 0) {
```

```
1780      printf("%s\n\n", "declare(name, return type, args...); - Creates a function
      ↪ decleration.");
1781      printf("%s\n\n", "Defined in `EVIL_PROC`.");
1782    } else
1783    // EVIL_SPECIFIER (default)
1784    if(strncmp("EVIL_SPECIFIER", token, 15) == 0) {
1785      printf("%s\n\n", "module - Adds `constant` and `storage`.");
1786      printf("%s\n\n", "Available by default.");
1787      printf("%s\n\n", "Exclude by defining `EVIL_NO_SPECIFIER`.");
1788    } else
1789    if(strncmp("EVIL_NO_SPECIFIER", token, 17) == 0) {
1790      printf("%s\n\n", "Exclude `EVIL_SPECIFIER` when defined.");
1791    } else
1792    if(strncmp("constant", token, 8) == 0) {
1793      printf("%s\n\n", "constant(type, name, value); - Create constant value.");
1794      printf("%s\n\n", "Defined in `EVIL_SPECIFIER`.");
1795    } else
1796    if(strncmp("storage", token, 8) == 0) {
1797      printf("%s\n\n", "storage(type, name, value); - Create what the C standard calls a
      ↪ \"storage-class specifier\" value, a `static` value.");
1798      printf("%s\n\n", "Defined in `EVIL_SPECIFIER`.");
1799    } else
1800    // EVIL_STRUCTURES (default)
1801    if(strncmp("EVIL_STRUCTURES", token, 10) == 0) {
1802      printf("%s\n\n", "module - adds Struct, Union, Typedef and BitField.");
1803      printf("%s\n\n", "Available by default.");
1804      printf("%s\n\n", "Exclude by defining `EVIL_NO_STRUCT`");
1805    } else
1806    if(strncmp("EVIL_NO_STRUCT", token, 14) == 0) {
1807      printf("%s\n\n", "Exclude `EVIL_STRUCTURES` by defining.");
1808    } else
1809    if(strncmp("Struct", token, 6) == 0) {
1810      printf("%s\n\n", "Struct(name) - Opens a new structure definition.");
1811      printf("%s\n\n", "Defined in `EVIL_STRUCTURES`.");
1812    } else
1813    if(strncmp("Union", token, 6) == 0) {
1814      printf("%s\n\n", "Union(name) - Opens a new union definition.");
1815      printf("%s\n\n", "Defined in `EVIL_STRUCTURES`.");
1816    } else
1817    if(strncmp("BitField", token, 6) == 0) {
1818      printf("%s\n\n", "BitField(name, type, width); - Creates a bitfield specification.");
1819      printf("%s\n\n", "Defined in `EVIL_STRUCTURES`.");
1820    } else
1821    if(strncmp("Typedef", token, 6) == 0) {
1822      printf("%s\n\n", "Typedef - Creates a typedef in the same manner as `typedef`.");
1823      printf("%s\n\n", "Defined in `EVIL_STRUCTURES`.");
1824    } else
1825    // EVIL_ASSERT
1826    if(strncmp("EVIL_ASSERT", token, 11) == 0) {
1827      printf("%s\n\n", "module - An assertion library.");
1828      printf("%s\n\n", "Defines: Assert, AssertMsg");
1829      printf("%s\n\n", "See also: NDEBUG");
```

```
1830      printf("%s\n\n", "Import by defining `EVIL_ASSERT` before including `evil.h`.");
1831    } else
1832    if(strncmp("NDEBUG", token, 6) == 0) {
1833      printf("%s\n\n", "If defined, Assert and AssertMsg become no-nops.");
1834    } else
1835    if(strncmp("AssertMsg", token, 9) == 0) {
1836      printf("%s\n\n", "AssertMsg(statement, message); - If statement is not true, print a
      ↪ stacktrace with a given message, then raise SIGABRT.");
1837      printf("%s\n\n", "Defined in `EVIL_ASSERT`.");
1838    } else
1839    if(strncmp("Assert", token, 6) == 0) {
1840      printf("%s\n\n", "Assert(statement, message); - If statement is not true, print a
      ↪ stacktrace, then raise SIGABRT.");
1841      printf("%s\n\n", "Defined in `EVIL_ASSERT`.");
1842    } else
1843    // EVIL_MALLOC
1844    if(strncmp("EVIL_MALLOC", token, 11) == 0) {
1845      printf("%s\n\n", "module - A 'safer' malloc.");
1846      printf("%s\n\n", "Defines: checked_malloc");
1847      printf("%s\n\n", "Import by defining `EVIL_MALLOC` before including `evil.h`");
1848    } else
1849    if(strncmp("checked_malloc", token, 14) == 0) {
1850      printf("%s\n\n", "`checked_malloc(object, object_type, buffer, fail_msg, exit_q);`");
1851      printf("%s\n\n", "* object - The identifier being assigned to.");
1852      printf("%s\n\n", "* object_type - The type of the identifier being assigned to.");
1853      printf("%s\n\n", "* buffer - The size of the buffer to pass to malloc.");
1854      printf("%s\n\n", "* fail_msg - A string to print to stderr if it fails to allocate.");
1855      printf("%s\n\n", "* exit_q - A boolean. If true, terminates the program.");
1856      printf("%s\n\n", "Defined in `EVIL_MALLOC`.");
1857    } else
1858    // EVIL_LAMBDA
1859    if(strncmp("EVIL_LAMBDA", token, 10) == 0) {
1860      printf("%s\n\n", "module - Lambda support.");
1861      printf("%s\n\n", "Defines - `lambda`");
1862      printf("%s\n\n", "Import by defining `EVIL_LAMBDA` before including `evil.h`");
1863    } else
1864    if(strncmp("lambda", token, 7) == 0) {
1865      printf("%s\n\n", "`lambda(return type, body)`");
1866      printf("%s\n\n", "A lambda returns a function pointer, and takes a return type and body
      ↪ definition.");
1867      printf("%s\n\n", "Defined in `EVIL_LAMBDA`");
1868      printf("%s\n\n", "Best to check the examples and `evil_manual` for this one.");
1869    } else
1870    // EVIL_RANDOM
1871    if(strncmp("EVIL_RANDOM", token, 11) == 0) {
1872      printf("%s\n\n", "module - Random numbers.");
1873      printf("%s\n\n", "Defines: `randomseed` and `random`");
1874      printf("%s\n\n", "Import by defining `EVIL_RANDOM` before including `evil.h`");
1875    } else
1876    if(strncmp("randomseed", token, 10) == 0) {
1877      printf("%s\n\n", "`randomseed()` - Seeds the C-rand number generator.");
1878      printf("%s\n\n", "Defined in `EVIL_RANDOM`.");
```

```
1879    } else
1880    if(strncmp("random", token, 6) == 0) {
1881      printf("%s\n\n", "`random(min, max)` - Takes two `int`s, and produces an `int` within
        ↪ the range.");
1882      printf("%s\n\n", "Defined in `EVIL_RANDOM`.");
1883    } else
1884    // EVIL_HELP
1885    if(strncmp("EVIL_HELP", token, 9) == 0) {
1886      printf("%s\n\n", "module - Provides help documentation.");
1887      printf("%s\n\n", "Defines: `evil_manual` and `evil_explain`");
1888      printf("%s\n\n", "See also: `EVIL_NO_HELP_MANUAL`");
1889      printf("%s\n\n", "Import by defining `EVIL_HELP` before including `evil.h`.");
1890    } else
1891    if(strncmp("EVIL_NO_HELP_MANUAL", token, 18) == 0) {
1892      printf("%s\n\n", "If defined before `EVIL_HELP` then `evil_manual` won't be defined.");
1893      printf("%s\n\n", "This should help reduce the size of the eventual binary.");
1894    } else
1895    if(strncmp("evil_manual", token, 11) == 0) {
1896      printf("%s\n\n", "`evil_manual();`");
1897      printf("%s\n\n", "Prints general help to stdout.");
1898      printf("%s\n\n", "Defined in `EVIL_HELP`");
1899    } else
1900    if(strncmp("evil_explain", token, 12) == 0) {
1901      printf("%s\n\n", "`evil_explain(const char* s);`");
1902      printf("%s\n\n", "Prints help about a specific identifier.");
1903      printf("%s\n\n", "Defined in `EVIL_HELP`");
1904    } else
1905    // EVIL_COROUTINE
1906    if(strncmp("EVIL_COROUTINE", token, 14) == 0) {
1907      printf("%s\n\n", "module - Adds coroutine support.");
1908      printf("%s\n\n", "Defines: `coroutine`, `co_return` and `co_end`.");
1909      printf("%s\n\n", "Import by defining `EVIL_COROUTINE` before including `evil.h`.");
1910    } else
1911    if(strncmp("coroutine", token, 9) == 0) {
1912      printf("%s\n\n", "`coroutine();` - Begin a coroutine body.");
1913      printf("%s\n\n", "Defined in `EVIL_COROUTINE`.");
1914    } else
1915    if(strncmp("co_return", token, 9) == 0) {
1916      printf("%s\n\n", "`co_return(value)` - Yield a value.");
1917      printf("%s\n\n", "Defined in `EVIL_COROUTINE`");
1918    } else
1919    if(strncmp("co_end", token, 6) == 0) {
1920      printf("%s\n\n", "`co_end();` - End the coroutine body.");
1921      printf("%s\n\n", "Defined in `EVIL_COROUTINE`");
1922    } else
1923    // EVIL_MATH
1924    if(strncmp("EVIL_MATH", token, 9) == 0) {
1925      printf("%s\n\n", "module - Adds better math support.");
1926      printf("%s\n\n", "Links to `math.h`.");
1927      printf("%s\n\n", "Defines: `add`, `take`, `divide` and `multiply` generics.");
1928      printf("%s\n\n", "Import by defining `EVIL_MATH` before including `evil.h`.");
1929    } else
```

```
1930    if(strncmp("add", token, 4) == 0) {
1931      printf("%s\n\n", "`add(a, b)` - Translated to `( a + b )`, allowing it to act as a
        ↪ generic.");
1932      printf("%s\n\n", "Defined in `EVIL_MATH`.");
1933    } else
1934    if(strncmp("take", token, 4) == 0) {
1935      printf("%s\n\n", "`take(a, b)` - Translated to `( a - b )`, allowing it to act as a
        ↪ generic.");
1936      printf("%s\n\n", "Defined in `EVIL_MATH`.");
1937    } else
1938    if(strncmp("multiply", token, 4) == 0) {
1939      printf("%s\n\n", "`multiply(a, b)` - Translated to `( a * b )`, allowing it to act as a
        ↪ generic.");
1940      printf("%s\n\n", "Defined in `EVIL_MATH`.");
1941    } else
1942    if(strncmp("divide", token, 4) == 0) {
1943      printf("%s\n\n", "`divide(a, b)` - Translated to `( a / b )`, allowing it to act as a
        ↪ generic.");
1944      printf("%s\n\n", "Defined in `EVIL_MATH`.");
1945    } else
1946    // EVIL_HASH
1947    if(strncmp("EVIL_HASH", token, 9) == 0) {
1948      printf("%s\n\n", "module - Provides string hash functions.");
1949      printf("%s\n\n", "Defines: `jenkins64`, `jenkins32`, `fletcher64`, `fletcher32`,
        ↪ `fletcher16`, `adler32`.");
1950      printf("%s\n\n", "Import by defining `EVIL_HASH` before including `evil.h`.");
1951    } else
1952    if(strncmp("jenkins64", token, 9) == 0) {
1953      printf("%s\n\n", "uint64_t jenkins64(char* key, size_t length);");
1954      printf("%s\n\n", "Only defined if int64_t available.");
1955      printf("%s\n\n", "The same input should always get the same output.");
1956      printf("%s\n\n", "This is an implementation of a Jenkins One-At-A-Time Hash.");
1957      printf("%s\n\n", "This is not a cryptographic hash function.");
1958      printf("%s\n\n", "Defined in `EVIL_HASH`.");
1959    } else
1960    if(strncmp("jenkins32", token, 9) == 0) {
1961      printf("%s\n\n", "uint32_t jenkins32(char* key, size_t length);");
1962      printf("%s\n\n", "The same input should always get the same output.");
1963      printf("%s\n\n", "This is an implementation of a Jenkins One-At-A-Time Hash.");
1964      printf("%s\n\n", "This is not a cryptographic hash function.");
1965      printf("%s\n\n", "Defined in `EVIL_HASH`.");
1966    } else
1967    if(strncmp("fletcher64", token, 10) == 0) {
1968      printf("%s\n\n", "uint64_t fletcher64(char* key, size_t length);");
1969      printf("%s\n\n", "Only defined if int64_t available.");
1970      printf("%s\n\n", "The same input should always get the same output.");
1971      printf("%s\n\n", "This is an implementation of a Fletcher's Checksum.");
1972      printf("%s\n\n", "This is not a cryptographic hash function.");
1973      printf("%s\n\n", "Defined in `EVIL_HASH`.");
1974    } else
1975    if(strncmp("fletcher32", token, 10) == 0) {
1976      printf("%s\n\n", "uint32_t fletcher32(char* key, size_t length);");
```

```
1977      printf("%s\n\n", "The same input should always get the same output.");
1978      printf("%s\n\n", "This is an implementation of a Fletcher's Checksum.");
1979      printf("%s\n\n", "This is not a cryptographic hash function.");
1980      printf("%s\n\n", "Defined in `EVIL_HASH`.");
1981    } else
1982    if(strncmp("fletcher16", token, 10) == 0) {
1983      printf("%s\n\n", "uint16_t fletcher16(char* key, size_t length);");
1984      printf("%s\n\n", "The same input should always get the same output.");
1985      printf("%s\n\n", "This is an implementation of a Fletcher's Checksum.");
1986      printf("%s\n\n", "This is not a cryptographic hash function.");
1987      printf("%s\n\n", "Defined in `EVIL_HASH`.");
1988    } else
1989    if(strncmp("adler32", token, 8) == 0) {
1990      printf("%s\n\n", "uint32_t adler32(char* key, size_t length);");
1991      printf("%s\n\n", "The same input should always get the same output.");
1992      printf("%s\n\n", "This is an implementation of a Adler-32 checksum.");
1993      printf("%s\n\n", "This is not a cryptographic hash function.");
1994      printf("%s\n\n", "Defined in `EVIL_HASH`.");
1995    } else
1996    // EVIL_BIT
1997    if(strncmp("EVIL_BIT", token, 9) == 0) {
1998      printf("%s\n\n", "module - defines macros for worded bit operators.");
1999      printf("%s\n\n", "Defines: `BIT_AND`, `BIT_OR`, `BIT_XOR`, `BIT_NOT`, `BIT_RSHIFT`,
      ↪  `BIT_LSHIFT`.");
2000      printf("%s\n\n", "Import by defining `EVIL_BIT` before including `evil.h`.");
2001    } else
2002    if(strncmp("BIT_AND", token, 8) == 0) {
2003      printf("%s\n\n", "BIT_AND(a, b) - Return a bitwise AND.");
2004      printf("%s\n\n", "Defined in `EVIL_BIT`");
2005    } else
2006    if(strncmp("BIT_OR", token, 7) == 0) {
2007      printf("%s\n\n", "BIT_OR(a, b) - Return a bitwise OR.");
2008      printf("%s\n\n", "Defined in `EVIL_BIT`");
2009    } else
2010    if(strncmp("BIT_XOR", token, 8) == 0) {
2011      printf("%s\n\n", "BIT_XOR(a, b) - Return a bitwise XOR.");
2012      printf("%s\n\n", "Defined in `EVIL_BIT`");
2013    } else
2014    if(strncmp("BIT_NOT", token, 8) == 0) {
2015      printf("%s\n\n", "BIT_NOT(a, b) - Return a bitwise NOT.");
2016      printf("%s\n\n", "Defined in `EVIL_BIT`");
2017    } else
2018    if(strncmp("BIT_RSHIFT", token, 10) == 0) {
2019      printf("%s\n\n", "BIT_RSHIFT(a, b) - Return a bitwise right shift.");
2020      printf("%s\n\n", "Defined in `EVIL_BIT`");
2021    } else
2022    if(strncmp("BIT_LSHIFT", token, 10) == 0) {
2023      printf("%s\n\n", "BIT_LSHIFT(a, b) - Return a bitwise left shift.");
2024      printf("%s\n\n", "Defined in `EVIL_BIT`");
2025    } else
2026    // EVIL_LOG
2027    if(strncmp("EVIL_LOG", token, 8) == 0) {
```

```
2028      printf("%s\n\n", "module - a helpful logging module.");
2029      printf("%s\n\n", "Defines:");
2030      printf("%s\n\n", "* message_file");
2031      printf("%s\n\n", "* warning_file");
2032      printf("%s\n\n", "* critical_file");
2033      printf("%s\n\n", "* error_file");
2034      printf("%s\n\n", "* info_file");
2035      printf("%s\n\n", "* debug_file");
2036      printf("%s\n\n", "* log_file");
2037      printf("%s\n\n", "* message");
2038      printf("%s\n\n", "* warning");
2039      printf("%s\n\n", "* critical");
2040      printf("%s\n\n", "* error");
2041      printf("%s\n\n", "* info");
2042      printf("%s\n\n", "* debug");
2043      printf("%s\n\n", "See also: DEBUG_LOG");
2044    } else
2045    if(strncmp("DEBUG_LOG", token, 10) == 0) {
2046      printf("%s\n\n", "If defined, then `debug` works as expected.");
2047      printf("%s\n\n", "If _not_ defined, then `debug` is a no-op.");
2048      printf("%s\n\n", "Referenced in `EVIL_LOG`.");
2049    } else
2050    if(strncmp("debug_file", token, 10) == 0) {
2051      printf("%s\n\n", "`char* debug_file`");
2052      printf("%s\n\n", "See `debug`.");
2053      printf("%s\n\n", "Defined in `EVIL_LOG`");
2054    } else
2055    if(strncmp("debug", token, 6) == 0) {
2056      printf("%s\n\n", "void debug(char* str); - When `EVIL_LOG` defined, prints a structured
       ↪ message to stdout.");
2057      printf("%s\n\n", "If `debug_file` is not 0-length, then prints a structured message to
       ↪ the file found at that path.");
2058      printf("%s\n\n", "Defined in `EVIL_LOG`.");
2059    } else
2060    if(strncmp("info_file", token, 10) == 0) {
2061      printf("%s\n\n", "`char* info_file`");
2062      printf("%s\n\n", "See `info`.");
2063      printf("%s\n\n", "Defined in `EVIL_LOG`");
2064    } else
2065    if(strncmp("info", token, 6) == 0) {
2066      printf("%s\n\n", "void info(char* str); - Prints a structured message to stdout.");
2067      printf("%s\n\n", "If `info_file` is not 0-length, then prints a structured message to
       ↪ the file found at that path.");
2068      printf("%s\n\n", "Defined in `EVIL_LOG`.");
2069    } else
2070    if(strncmp("error_file", token, 10) == 0) {
2071      printf("%s\n\n", "`char* error_file`");
2072      printf("%s\n\n", "See `error`.");
2073      printf("%s\n\n", "Defined in `EVIL_LOG`");
2074    } else
2075    if(strncmp("error", token, 6) == 0) {
2076      printf("%s\n\n", "void error(char* str); - Prints a structured message to stderr.");
```

```
2077      printf("%s\n\n", "If `error_file` is not 0-length, then prints a structured message to
      ↪  the file found at that path.");
2078      printf("%s\n\n", "Defined in `EVIL_LOG`.");
2079    } else
2080    if(strncmp("critical_file", token, 10) == 0) {
2081      printf("%s\n\n", "`char* critical_file`");
2082      printf("%s\n\n", "See `critical`.");
2083      printf("%s\n\n", "Defined in `EVIL_LOG`");
2084    } else
2085    if(strncmp("critical", token, 6) == 0) {
2086      printf("%s\n\n", "void critical(char* str); - Prints a structured message to stderr.");
2087      printf("%s\n\n", "If `critical_file` is not 0-length, then prints a structured message
      ↪  to the file found at that path.");
2088      printf("%s\n\n", "Defined in `EVIL_LOG`.");
2089    } else
2090    if(strncmp("warning_file", token, 10) == 0) {
2091      printf("%s\n\n", "`char* warning_file`");
2092      printf("%s\n\n", "See `warning`.");
2093      printf("%s\n\n", "Defined in `EVIL_LOG`");
2094    } else
2095    if(strncmp("warning", token, 6) == 0) {
2096      printf("%s\n\n", "void warning(char* str); - Prints a structured message to stdout.");
2097      printf("%s\n\n", "If `warning_file` is not 0-length, then prints a structured message to
      ↪  the file found at that path.");
2098      printf("%s\n\n", "Defined in `EVIL_LOG`.");
2099    } else
2100    if(strncmp("message_file", token, 10) == 0) {
2101      printf("%s\n\n", "`char* message_file`");
2102      printf("%s\n\n", "See `message`.");
2103      printf("%s\n\n", "Defined in `EVIL_LOG`");
2104    } else
2105    if(strncmp("message", token, 6) == 0) {
2106      printf("%s\n\n", "void message(char* str); - Prints a structured message to stdout.");
2107      printf("%s\n\n", "If `message_file` is not 0-length, then prints a structured message to
      ↪  the file found at that path.");
2108      printf("%s\n\n", "Defined in `EVIL_LOG`.");
2109    } else
2110
2111    // TODO: EVIL_CLI 54
2112
2113    // TODO: EVIL_KLASS
2114
2115    // TODO: EVIL_ENCODE
2116
2117    // NOT YET IMPLEMENTED
2118    // TODO: EVIL_STRING
2119    // TODO: EVIL_PARSE_ARG
2120    // TODO: EVIL_LIST
2121
2122    {
2123      printf("%s: %s\n", "No documentation found for", token);
2124    }
```

```
}

#endif

#ifndef EVIL_NO_INT
  // Included by default.

  #include <stdint.h>
  #include <float.h>
  #ifdef INT64_MAX
    #define Number(x) int64_t x
    #define MaxNumber INT64_MAX
  #endif
  #define Decimal(x) long double x
  #define MaxDecimal LDBL_MAX
  // TODO: itoa, convenience.
#endif
#ifndef EVIL_NO_IO
  // The IO Module.
  // Included by default. To pretend C is high-level.


  // User wants IO, give the all the IO.
  #include <stdio.h>

  // Yes, Generics. (aka type-switch). It's C11 only,
  // but who cares.
  // stdint identifiers (inttypes.h) should be catered for by the below.
  // Original display_format macro by Robert Gamble, (c) 2012
  // used with permission.
  // Expanded upon to incorporate const, volatile and const volatile types,
  // as they don't get selected for. (static does for obvious reasons).

  // Whilst volatile types can change between accesses, technically using a
  // _Generic _shouldn't_ access it, but compile to the right choice.

  #define display_format(x) _Generic((x), \
    char: "%c", \
    signed char: "%hhd", \
    unsigned char: "%hhu", \
    signed short: "%hd", \
    unsigned short: "%hu", \
    signed int: "%d", \
    unsigned int: "%u", \
    long int: "%ld", \
    unsigned long int: "%lu", \
    long long int: "%lld", \
    unsigned long long int: "%llu", \
    float: "%f", \
    double: "%f", \
    long double: "%Lf", \
    char *: "%s", \
```

```
2177      void *: "%p", \
2178      volatile char: "%c", \
2179      volatile signed char: "%hhd", \
2180      volatile unsigned char: "%hhu", \
2181      volatile signed short: "%hd", \
2182      volatile unsigned short: "%hu", \
2183      volatile signed int: "%d", \
2184      volatile unsigned int: "%u", \
2185      volatile long int: "%ld", \
2186      volatile unsigned long int: "%lu", \
2187      volatile long long int: "%lld", \
2188      volatile unsigned long long int: "%llu", \
2189      volatile float: "%f", \
2190      volatile double: "%f", \
2191      volatile long double: "%Lf", \
2192      volatile char *: "%s", \
2193      volatile void *: "%p", \
2194      const char: "%c", \
2195      const signed char: "%hhd", \
2196      const unsigned char: "%hhu", \
2197      const signed short: "%hd", \
2198      const unsigned short: "%hu", \
2199      const signed int: "%d", \
2200      const unsigned int: "%u", \
2201      const long int: "%ld", \
2202      const unsigned long int: "%lu", \
2203      const long long int: "%lld", \
2204      const unsigned long long int: "%llu", \
2205      const float: "%f", \
2206      const double: "%f", \
2207      const long double: "%Lf", \
2208      const char *: "%s", \
2209      const void *: "%p", \
2210      const volatile char: "%c", \
2211      const volatile signed char: "%hhd", \
2212      const volatile unsigned char: "%hhu", \
2213      const volatile signed short: "%hd", \
2214      const volatile unsigned short: "%hu", \
2215      const volatile signed int: "%d", \
2216      const volatile unsigned int: "%u", \
2217      const volatile long int: "%ld", \
2218      const volatile unsigned long int: "%lu", \
2219      const volatile long long int: "%lld", \
2220      const volatile unsigned long long int: "%llu", \
2221      const volatile float: "%f", \
2222      const volatile double: "%f", \
2223      const volatile long double: "%Lf", \
2224      const volatile char *: "%s", \
2225      const volatile void *: "%p", \
2226      default: "%d")
2227
2228    // The main printing function.
```

```
#define display(x) printf(display_format(x), x)
#define displayf(f, x) fprintf(f, display_format(x), x)

// Windows has a different line ending.
#if defined(_WIN32) || defined(__WIN32) || defined(WIN32) || defined(__WIN32__) ||
 ↪  defined(_WIN64) || defined(__WIN64) || defined(WIN64) || defined(__WIN64__) ||
 ↪  defined(__WINNT) || defined(__WINNT__) || defined(WINNT)
  #define displayln(x) do { printf(display_format(x), x); printf("%s", "\r\n"); } while(0)
  #define displayfln(f, x) do { fprintf(f, display_format(x), x); printf("%s", "\r\n"); }
   ↪  while(0)
#else
  #define displayln(x) do { printf(display_format(x), x); printf("%c", '\n'); } while(0)
  #define displayfln(f, x) do { fprintf(f, display_format(x), x); printf("%c", '\n'); }
   ↪  while(0)
#endif

// Basically a _Generic.
#define repr_type(x) _Generic((0,x), \
  char: "char", \
  signed char: "signed char", \
  unsigned char: "unsigned char", \
  signed short: "signed short", \
  unsigned short: "unsigned short", \
  signed int: "signed int", \
  unsigned int: "unsigned int", \
  long int: "long int", \
  unsigned long int: "unsigned long int", \
  long long int: "long long int", \
  unsigned long long int: "unsigned long long int", \
  float: "float", \
  double: "double", \
  long double: "long double", \
  char *: "char pointer", \
  void *: "void pointer", \
  volatile char: "volatile char", \
  volatile signed char: "volatile signed char", \
  volatile unsigned char: "volatile unsigned char", \
  volatile signed short: "volatile signed short", \
  volatile unsigned short: "volatile unsigned short", \
  volatile signed int: "volatile signed int", \
  volatile unsigned int: "volatile unsigned int", \
  volatile long int: "volatile long int", \
  volatile unsigned long int: "volatile unsigned long int", \
  volatile long long int: "volatile long long int", \
  volatile unsigned long long int: "volatile unsigned long long int", \
  volatile float: "volatile float", \
  volatile double: "volatile double", \
  volatile long double: "volatile long double", \
  volatile char *: "volatile char pointer", \
  volatile void *: "volatile void pointer", \
  const char: "const char", \
  const signed char: "const signed char", \
```

```
2277      const unsigned char: "const unsigned char", \
2278      const signed short: "const signed short", \
2279      const unsigned short: "const unsigned short", \
2280      const signed int: "const signed int", \
2281      const unsigned int: "const unsigned int", \
2282      const long int: "const long int", \
2283      const unsigned long int: "const unsigned long int", \
2284      const long long int: "const long long int", \
2285      const unsigned long long int: "const unsigned long long int", \
2286      const float: "const float", \
2287      const double: "const double", \
2288      const long double: "const long double", \
2289      const char *: "const char pointer", \
2290      const void *: "const void pointer", \
2291      const volatile char: "const volatile char", \
2292      const volatile signed char: "const volatile signed char", \
2293      const volatile unsigned char: "const volatile unsigned char", \
2294      const volatile signed short: "const volatile signed short", \
2295      const volatile unsigned short: "const volatile unsigned short", \
2296      const volatile signed int: "const volatile signed int", \
2297      const volatile unsigned int: "const volatile unsigned int", \
2298      const volatile long int: "const volatile long int", \
2299      const volatile unsigned long int: "const volatile unsigned long int", \
2300      const volatile long long int: "const volatile long long int", \
2301      const volatile unsigned long long int: "const volatile unsigned long long int", \
2302      const volatile float: "const volatile float", \
2303      const volatile double: "const volatile double", \
2304      const volatile long double: "const volatile long double", \
2305      const volatile char *: "const volatile char pointer", \
2306      const volatile void *: "const volatile void pointer", \
2307      default: "Unknown")
2308
2309
2310    // endl, just a symbol that can be used to produce the normal
2311    // line ending.
2312    // endlf can take a file to print to.
2313    // e.g. ```display(x); display(y); endl;```
2314    //  ```endlf(FILE* x);```
2315    // Windows has a different line ending.
2316    #if defined(_WIN32) || defined(__WIN32) || defined(WIN32) || defined(__WIN32__) ||
       ↪  defined(_WIN64) || defined(__WIN64) || defined(WIN64) || defined(__WIN64__) ||
       ↪  defined(__WINNT) || defined(__WINNT__) || defined(WINNT)
2317      #define endl printf("%s", "\r\n")
2318      #define endlf(f) fprintf(f, "%s", "\r\n")
2319    #else
2320      #define endl printf("%c", '\n')
2321      #define endlf(f) fprintf(f, "%c", '\n')
2322    #endif
2323
2324  #endif
2325  #ifndef EVIL_NO_KLASS
2326    // Available by default
```

```
2327
2328    #ifndef EVIL_NO_WARN
2329      #warning "KLASS module is still under construction."
2330    #endif
2331
2332    // The user will probably need this.
2333    // Annoying to drag in the whole stdlib, but can't be
2334    // helped.
2335    #include <stdlib.h>
2336
2337    /*
2338
2339      TODO:
2340
2341      * Can we find a way to bundle class names into the definition
2342        in such a way that send can guess the function to call
2343        correctly?
2344
2345      * Can we do inheritance in any way at all?
2346
2347      * Documentation
2348
2349    */
2350
2351    #define new(obj, class, ...) class obj; class ## _new (&obj, ## __VA_ARGS__)
2352
2353    #define method(class, name, ret, body, ...) ret class ## _ ## name (class * self, ##
      ↪  __VA_ARGS__) body
2354
2355    // GNU Extension $ is a valid identifier
2356    #define $ send
2357
2358    #define send(class, method, obj, ...) class ## _ ## method (&obj, ## __VA_ARGS__)
2359
2360    #define Class(name, values, new, ...); \
2361  struct name values; \
2362  typedef struct name name; \
2363  int name ## _new (name * self, ## __VA_ARGS__)  new // Generate the initialiser
2364
2365  #endif
2366
2367  #ifdef EVIL_LAMBDA
2368    // This requires nested functions to be allowed.
2369    // Only GCC supports it.
2370    // ... Unconfirmed if Clang does. It might.
2371    #if defined(__clang__) || !defined(__GNUC__)
2372      #error "Lambda requires a GNU compiler."
2373    #endif
2374    // A cleaner, but slightly more cumbersome lambda:
2375    #define lambda(ret_type, _body) ({ ret_type _ _body _; })
2376    // e.g. int (*max)(int, int) = lambda (int, (int x, int y) { return x > y ? x : y; });
2377    // Pros:
```

```
2378    // * Woot, easier to pass, as the user has to know the signature anyway.
2379    // * Name not part of lambda definition. More lambda-y.
2380    // * Body of function inside macro, feels more like a lambda.
2381    // * Uses expression disgnator (GCC-only), which creates a properly constructed function
        ↪  pointer.
2382    // * It *may* work under Clang too!
2383    // Cons:
2384    // * The signature isn't constructed for the user, they have to both know and understand
        ↪  it.
2385  #endif
2386
2387  #ifdef EVIL_LIST
2388
2389    // TODO: gc-backed linked list library
2390    #error "Not Yet Implemented"
2391
2392  #endif
2393  #ifdef EVIL_LOG
2394
2395    // File definitions for the user to control.
2396    char* message_file = "";
2397    char* warning_file = "";
2398    char* critical_file = "";
2399    char* error_file = "";
2400    char* info_file = "";
2401    char* debug_file = "";
2402    char* log_file = "";
2403
2404    // Libraries we need
2405    #include <stdio.h>
2406    #include <time.h>
2407    #include <assert.h>
2408
2409    void message(char* str) {
2410      time_t now;
2411      struct tm* timeinfo;
2412      time(&now);
2413      timeinfo = localtime(&now);
2414
2415      if(log_file[0] != '\0') {
2416        // Opening in append mode is Not ANSI C, but ISO C
2417        FILE* logFile = fopen(log_file, "a");
2418        assert(logFile != NULL);
2419        fprintf(logFile, "%s\n%u %s%s\n%s\n", "---MESSAGE---", (unsigned)time(NULL),
        ↪  asctime(timeinfo), str, "---END MESSAGE---");
2420        fclose(logFile);
2421      }
2422      if(message_file[0] != '\0') {
2423        // Opening in append mode is Not ANSI C, but ISO C
2424        FILE* messageFile = fopen(message_file, "a");
2425        assert(messageFile != NULL);
```

```
2426       fprintf(messageFile, "%s\n%u %s%s\n%s\n", "---MESSAGE---", (unsigned)time(NULL),
       ↪  asctime(timeinfo), str, "---END MESSAGE---");
2427       fclose(messageFile);
2428     }
2429     //NOTE: asctime has a newline.
2430     printf("%s\n%u %s%s\n%s\n", "---MESSAGE---", (unsigned)time(NULL), asctime(timeinfo),
       ↪  str, "---END MESSAGE---");
2431   }
2432
2433   void warning(char* str) {
2434     time_t now;
2435     struct tm* timeinfo;
2436     time(&now);
2437     timeinfo = localtime(&now);
2438
2439     if(log_file[0] != '\0') {
2440         FILE* logFile = fopen(log_file, "a");
2441         assert(logFile != NULL);
2442         fprintf(logFile, "%s\n%u %s%s\n%s\n", "---WARNING---", (unsigned)time(NULL),
       ↪  asctime(timeinfo), str, "---END WARNING---");
2443         fclose(logFile);
2444     }
2445
2446     if(warning_file[0] != '\0') {
2447       //NOTE: asctime has a newline.
2448       FILE* warningFile = fopen(warning_file, "a");
2449       assert(warningFile);
2450       fprintf(warningFile, "\x1b[30m%s\n%u %s%s\n%s\n\x1b[0m", "---WARNING---",
       ↪  (unsigned)time(NULL), asctime(timeinfo), str, "---END WARNING---");
2451       fclose(warningFile);
2452     }
2453     printf("\x1b[30m%s\n%u %s%s\n%s\n\x1b[0m", "---WARNING---", (unsigned)time(NULL),
       ↪  asctime(timeinfo), str, "---END WARNING---");
2454   }
2455
2456   void critical(char* str) {
2457     time_t now;
2458     struct tm* timeinfo;
2459     time(&now);
2460     timeinfo = localtime(&now);
2461
2462     if(log_file[0] != '\0') {
2463         FILE* logFile = fopen(log_file, "a");
2464         assert(logFile != NULL);
2465         fprintf(logFile, "%s\n%u %s%s\n%s\n", "---CRITICAL---", (unsigned)time(NULL),
       ↪  asctime(timeinfo), str, "---END CRITICAL---");
2466         fclose(logFile);
2467     }
2468
2469     if(critical_file[0] != '\0') {
2470       //NOTE: asctime has a newline.
2471       FILE* criticalFile = fopen(critical_file, "a");
```

```
2472        assert(criticalFile);
2473        fprintf(criticalFile, "%s\n%u %s%s\n%s\n", "---CRITICAL---", (unsigned)time(NULL),
        ↪  asctime(timeinfo), str, "---END CRITICAL---");
2474        fclose(criticalFile);
2475      }
2476      fprintf(stderr, "\x1b[5m\x1b[33m%s\n%u %s%s\n%s\n\x1b[0m", "---CRITICAL---",
        ↪  (unsigned)time(NULL), asctime(timeinfo), str, "---END CRITICAL---");
2477    }
2478
2479    void error(char* str) {
2480      time_t now;
2481      struct tm* timeinfo;
2482      time(&now);
2483      timeinfo = localtime(&now);
2484
2485      if(log_file[0] != '\0') {
2486          FILE* logFile = fopen(log_file, "a");
2487          assert(logFile != NULL);
2488          fprintf(logFile, "%s\n%u %s%s\n%s\n", "---ERROR---", (unsigned)time(NULL),
        ↪  asctime(timeinfo), str, "---END ERROR---");
2489          fclose(logFile);
2490      }
2491
2492      if(error_file[0] != '\0') {
2493        //NOTE: asctime has a newline.
2494        FILE* errorFile = fopen(error_file, "a");
2495        assert(errorFile);
2496        fprintf(errorFile, "%s\n%u %s%s\n%s\n", "---ERROR---", (unsigned)time(NULL),
        ↪  asctime(timeinfo), str, "---END ERROR---");
2497        fclose(errorFile);
2498      }
2499      fprintf(stderr, "\x1b[35m%s\n%u %s%s\n%s\n\x1b[0m", "---ERROR---", (unsigned)time(NULL),
        ↪  asctime(timeinfo), str, "---END ERROR---");
2500    }
2501
2502    void info(char* str) {
2503      time_t now;
2504      struct tm* timeinfo;
2505      time(&now);
2506      timeinfo = localtime(&now);
2507
2508      if(log_file[0] != '\0') {
2509          FILE* logFile = fopen(log_file, "a");
2510          assert(logFile != NULL);
2511          fprintf(logFile, "%s\n%u %s%s\n%s\n", "---INFO---", (unsigned)time(NULL),
        ↪  asctime(timeinfo), str, "---END INFO---");
2512          fclose(logFile);
2513      }
2514
2515      if(info_file[0] != '\0') {
2516        //NOTE: asctime has a newline.
2517        FILE* infoFile = fopen(info_file, "a");
```

```
2518        assert(infoFile);
2519        fprintf(infoFile, "%s\n%u %s%s\n%s\n", "---INFO---", (unsigned)time(NULL),
    ↪  asctime(timeinfo), str, "---END INFO---");
2520        fclose(infoFile);
2521      }
2522      printf("%s\n%u %s%s\n%s\n", "---INFO---", (unsigned)time(NULL), asctime(timeinfo), str,
    ↪  "---END INFO---");
2523    }
2524
2525    #ifdef DEBUG_LOG
2526      void debug(char* str) {
2527          time_t now;
2528          struct tm* timeinfo;
2529          time(&now);
2530          timeinfo = localtime(&now);
2531
2532          if(log_file[0] != '\0') {
2533              FILE* logFile = fopen(log_file, "a");
2534              assert(logFile != NULL);
2535              fprintf(logFile, "%s\n%u %s%s\n%s\n", "---DEBUG---", (unsigned)time(NULL),
    ↪  asctime(timeinfo), str, "---END DEBUG---");
2536              fclose(logFile);
2537          }
2538
2539          if(debug_file[0] != '\0') {
2540            //NOTE: asctime has a newline.
2541            FILE* debugFile = fopen(debug_file, "a");
2542            assert(debugFile);
2543            fprintf(debugFile, "%s\n%u %s%s\n%s\n", "---DEBUG---", (unsigned)time(NULL),
    ↪  asctime(timeinfo), str, "---END DEBUG---");
2544            fclose(debugFile);
2545          }
2546          printf("%s\n%u %s%s\n%s\n", "---DEBUG---", (unsigned)time(NULL), asctime(timeinfo),
    ↪  str, "---END DEBUG---");
2547        }
2548    #else
2549      void debug(char* str) {return; }
2550    #endif
2551  #endif
2552
2553
2554  #ifndef EVIL_NO_MAIN
2555    // Included by default
2556    #define Main int main(int __attribute__((unused)) argc, char __attribute__((unused))
        ↪  **argv
2557  #endif
2558  #ifdef EVIL_MALLOC
2559    // Excluded by default: stdlib is a big dependency.
2560
2561    // Provides a "safer" malloc pattern
2562
2563    #include <stdlib.h>
```

```
  #include <stdio.h>
  #define checked_malloc(object, object_type, buffer, fail_msg, exit_q) object_type object =
   ↪  malloc(buffer); if(!object){fprintf(stderr, "%s\n", fail_msg);
   ↪  if(exit_q){exit(EXIT_FAILURE);}}
#endif

#ifdef EVIL_MATH

  // This library does the heavy lifting.
  #include <math.h>
  // TODO: Make math.h functions use _Generic to
  // handle multiple types.
  #define add(_a, _b) (_a + _b)
  #define take(_a, _b) (_a - _b)
  #define multiply(_a, _b) (_a * _b)
  #define divide(_a, _b) (_a / _b)
#endif
#ifdef EVIL_PARSE_ARG

  #error "Not Yet Implemented"

  /*

  Needs to handle:

  Single letter flags:

  -a -b
  -ab

  Key word flags:

  --word=x
  --word x

  Setting boolean flags:
  (absence implies false)

  --bool-flag true
  --bool-flag false
  ---bool-flag


  */

#endif

#ifndef EVIL_NO_PROC
  // Included by default

  #define declare(_name, _ret, ...) _ret _name(__VA_ARGS__)
  #define proc(_name, _ret, ...) _ret _name(__VA_ARGS__){
```

```
#endif
#ifdef EVIL_RANDOM
  #include <stdlib.h>
  #include <time.h>

  #ifndef EVIL_NO_WARN
    #warning "randomseed and random(min, max) depend on C's rand. They are not cryptographically safe. They are not even good pseudorandom generators."
  #endif

  // Crappy C-standard stuff.
  #define randomseed() srand((unsigned int)time(NULL))
  #define random(min, max) (rand() % (max + 1 - min)) + min
  // TODO: Mersenne Twister (see http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/CODES/mt19937ar.c)
#endif
#ifndef EVIL_NO_SPECIFIER
  // Included by default.

  #define constant(_type, _name, _value) const _type _name = _value

  // Static vars are usually referred to as storage-class
  // specifiers in the C standard.
  #define storage(_type, _name, _value) static _type _name = _value
#endif
#ifdef EVIL_STRING

  // TODO: gc-backed cord lib
  #error "Not Yet Implemented"

  // Should also provide wrapper functions for any other modules
  // that use a C String.

#endif

#ifndef EVIL_NO_STRUCT
  // Imported by default

  #define BitField(_name, _type, _width) _type _name : _width
  #define Struct(_name) struct _name {
  #define Union(_name) union __name {
  #define Typedef typedef
#endif


#endif
```