C2 language

Bas van den Berg

2014
# Table of contents

1. Intro
2. Design
3. Changes
4. Concept: types
5. Concept: multi-pass parsing
6. Concept: modules
7. Concept: build system
8. The C2 Project
The C programming language has been around for a long time and is still used a lot nowadays. The core of the language is very solid, but other aspects are showing their age. C2 attempts to modernize these parts, while keeping the feel of C. It should be seen as an *evolution* of C.
C2 Design goals

- Higher development speed
- Same/better speed of execution
- Better compilation times
- Integrated build system
- Stricter syntax (easier for tooling)
- Great tooling (formatting tool, graphical refactoring tool)
- C2 programs can use C libraries (and vice-versa)
- Should be easy to learn for C programmers (evolution)
- Should support avoiding common mistakes
C2 Non-goals

- higher-level features (garbage collection, classes, etc)
- completely new language
C improvement points

- Lots of typing (header/forward declarations)
- Build system separate from language
- Variable syntax complex

```c
char *(*(*foo [] [8])())[];
```
Changes

From C to C2

- No header files
- No forward declarations
- No includes necessary
- Integrated compiler option syntax
- Integrated Build system
- Compilation per target, not file
- Simplified type syntax
- Stricter error checking (uninitialized var usage is error)
- More built-in types (uint8, uint16, uint32, int8, int16, int32, ...)
- Some syntax cleanup
- ...

Bas van den Berg
C2 language 2014
Keyword changes

removed keywords:
- extern
- static
- typedef
- long
- short
- signed
- unsigned

new keywords:
- module
- import
- as
- public
- local
- type
- func
- nil
- elemsof

- int8
- int16
- int32
- int64
- uint8
- uint16
- uint32
- uint64
- float32
- float64
Hello World!

Spot the five differences...
Example - Base Types

```c2
module types;

public type Number int;

type PNum int**;

type IntArr int[20];

public type String const uint8*;

type DoubleBufPtr DoubleBuf*;
type DoubleBuf Buffer*[2];
```

All 'typedefs' are uniform..
Example - Function Types

Note: declaring an array/pointer to function types requires to steps.

```c2
module types;

type CallBack func int(int a, utils.Point* p);

type CBFunc func void (MyType* mt, ...);
CBFunc[10] callbacks;
```
Example - Struct Types

```c
module types;

type ChessBoard struct {
    int[8][8] board;
}

type Example struct {
    int n;
    union {
        int b;
        Point c;
    } choice;
    volatile uint32 count;
}
```
Feature - multi-part array initialization

```c2
module types;

type Element struct {
    const char[16] name;
    int value;
}

const Element[] elements;

elements += { "test1", 10 }
...

elements += { "test2", 20 }
...

elements += { "test3", 30 }
```

This is possible because recipe file and multi-pass parsing.
Example C2:

```c2
module example;

Number hundred = 100;

func Number add(Number a, Number b) {
    return a + b;
}

type Number int;
```

Ordering in a file is not relevant for parsing. Variables, functions and types can be specified in any order.
Package scopes cause less prefixes in names

```
module graphics;

class Buffer {
  ...
}

class void init() {
  ...
}

class void create(Buffer* buf) {
  ...
}
```

```
#ifndef GRAPHICS_H
#define GRAPHICS_H

typedef struct {
  ...
} Graphics_Buffer;

void graphics_init();

void graphics_create( Graphics_Buffer* buf);

#endif
```
Symbol accessibility

Only public symbols can be used outside the module. Non-public symbols can be used from any file within the same module. Non-public roughly translates to the C keyword *static*. 

```c2
application.c2

module gui;

import utils local;

Engine engine; // ok
Engine_priv priv; // not ok

my_utils.c2

module utils;

public type Engine struct {
    ...
}

type Engine_priv struct {
    ...
}
```
Opaque pointers

Use of pointers to non-public Types is allowed (but no de-referencing)
Multi-file module

No need to *import* your own module. Treat as if all code is in the same file.
### gui.c2

```c2
module gui;

import utils local;

utils.Buffer buf;

func void run()
{
    utils.log("ok");
    log("also ok");
}
```

### utils_buf.c2

```c2
module utils;

public type Buffer int[10];
```

### utils_log.c2

```c2
module utils;

public func void log(int8* msg)
{
    log("also ok");
}
```
Naming conflicts

Use statements are have a file scope, not module scope.
Use statement

Syntax is 'import long as short' to avoid too much typing, while making it clear where a symbol comes from.
Concept: modules

Use statement - example

```c2
module server;

import network as net;
import filesystem as fs;

func fs.File* getFile(net.URL url) {
    net.Socket sock = net.open(server);
    fs.File* file = net.get(sock, url);
    net.close(sock);
    return file;
}
```

⇒ Both modules have been aliased.
Use statement - example 2

```c2
module server;

import network as net local;
import filesystem as fs local;

func File* getFile(URL url) {
    Socket sock = net.open(server);
    File* file = get(sock, url);
    net.close(sock);
    return file;
}
```

⇒ Both modules have been aliased and imported locally.
Only often used modules are imported locally.
the c2 module

A special module called c2 can be used to get compile-information, build-time, (git/svn) version, build-flags etc. So no need to script some of your own.

```c2
module example;
import c2;
uint64 buildtime = c2.buildtime;
const char* version = c2.version;
const char*[[]] options = c2.options;
```
To build C2 projects, simply use

$ c2c

This searches the current and parent directories for the *recipe file* (recipe.txt). This means c2c can be called from any subdir in the project, which is handy when working in a subdir:

`drivers/networking/ethernet$ c2c`
Every C2 project has a recipe file in the root directory of the project. This file contains a list of all target that need to be build. For each target the recipe describes:

- name
- type - executable/library
- files - all required c2 files. This allows C2 to do better optimizations and error checking
- configuration - all ’defines’ used
- exports - which modules will be exported (visible as ELF object in the resulting file
example recipe file

recipe.txt

executable one
   example1/gui.c2
   example1/utils.c2
end

library mylib
   config NO_DEBUG WITH_FEATURE1 FEATURE2
   export mylib
   example2/mylib1.c2
   example2/mylib2.c2
end
The results of building are stored in the output directory. So make clean is simply removing of this directory.
Because of the language design, compiling C2 code requires a different process than compiling C. The basic process is as follows:

- parse all c2 files into ASTs
- check all ASTs
- generate IR code per module
- generate object code per module
- link all objects

So in effect, all files are parsed simultaneously.
Build process difference

- C: a new compiler is started for each .c file
- C2 finds a compile error in file x much faster
- C2 generates code per module, not file
- The generation (+ optimization) step takes much longer than the parse/analyse step, so the yellow blocks are really much bigger
Build process per file

- first parse the file to the AST
- extract the symbol table and add to the global table
- generate IR code from the AST and Global Symbol table
The scope of the C2 language is wider than the C language.

For example, there is no syntax format for specifying attributes in the C language. In C2, the syntax is specified, there are common attributes and compiler-builders can add custom attributes without disturbing others.

⇒ widening the language scope allows for huge improvements and ease of use.
The language makes several interesting tooling options possible. Some of these options are grouped in c2reto, the C2 Refactor Tool:

- Visualizing dependencies between functions/vars/types/files/modules
- Drag ’N Drop reordering of declarations in files
- Drag ’N Drop moving of declarations between files
- renaming, style formatting, etc
The C2 compiler is currently built on top of llvm and uses the clang Lexer/Pre-processor. The parser and semantic analyser are custom. Code is translated to an AST (Abstract Syntax Tree) that’s similar to clang’s (but much simpler). After generating and checking the AST’s, LLVM’s IR code is generated from the AST.
http://www.c2lang.org

http://github.com/c2lang/c2compiler