

# The CMod Manual

## 1 What is CMod?

CMod is a novel tool that provides a sound module system for C. CMod works by enforcing a set of four rules that are based on principles of modular reasoning and on current programming practice. CMod's rules flesh out the convention that `.h` header files are module interfaces and `.c` source files are module implementations. Although this convention is well-known, existing explanations of it are incomplete, omitting important subtleties needed for soundness. In contrast, we have proven formally that CMod's rules enforce both information hiding and type-safe linking.

To use CMod, the programmer develops and builds their software as usual, redirecting the compiler and linker to CMod's wrappers. CMod has been applied to large gamut of open source programs and has revealed subtle bugs in coding. Violations to CMod's rules revealed more than a thousand information hiding errors, dozens of typing errors, and hundreds of cases that, although not currently bugs, make programming mistakes more likely as the code evolves. At the same time, programs generally adhere to the assumptions underlying CMod's rules, and so we could fix rule violations with a modest effort. CMod can effectively support modular programming in C: it soundly enforces type-safe linking and information hiding while being largely compatible with existing practice.

To get your hands dirty right away, we talk about how you can download, compile and run CMod (Sections 2 and 3). CMod points out locations in your code which are potential sources of errors. To understand the errors you need to have at least a cursory idea of what CMod checks in programs for which we provide an overview (Section 4). After you develop a basic intuition about the design of CMod you can appreciate the violations that it points out. Then we use an example codebase (`vsftpd`) to describe the results of running CMod over it (Section 5).

## 2 Compiling CMod

Download the latest tarball from <http://www.cs.umd.edu/projects/CMod/> and untar into your favorite directory. Let us call this directory `<DIR>`. When you go into `<DIR>` and do `make` it will fail and give you a neat little error message saying that you do not have GCC sources. A copy of the sources has been included in the distribution and you should find `gcc-core-3.3.tar.gz` in the `gcc-patch/` directory. Go to the section "Setting Up GCC" for instructions to setup the GCC for CMod to compile against. That section will tell you how to setup two directories, `<DIRGCC>/gcc-3.3/` and `<DIRGCC>/gcc-build`. From this point onwards we will assume that you have GCC sources setup as required. Now do the following:

1. For bash do `export GCCDIR=<DIRGCC>/gcc-3.3/` and do `export GCCBUILD=<DIRGCC>/gcc-build/`. Alternatively, you can edit `gcc-patch/Makefile` if you do not want to repeat this every time you compile CMod (which should hopefully be only once).
2. Compile (using 'make' inside the `src-ocaml/` directory) to get the CMod binaries in the `<DIR>/bin/` directory.

## 3 Running CMod

This section assumes that you have compiled CMod as mentioned in the “Compiling CMod” section and that the CMod binaries are available in `<DIR>/bin`. To incorporate CMod into your build process simply put the binaries at the head of your shell path. i.e:

- For bash do `export PATH=<DIR>/bin/:$PATH`
- For sh do `setenv PATH <DIR>/bin/:$PATH`

It is *essential* for the CMod binaries be before other search locations. This is because CMod assumes that the compiler (`gcc`) and linkers (`ld` and `ar`) appear in standard locations and provides wrapper scripts for these.

This is all the changes you need to make to incorporate CMod into your build process. Compile your project normally using your `Makefile` script. The CMod wrappers will dump out some extra information in the midst of the normal compile/link instructions. At the end of the `make` process, CMod prints out a summary of the rule violations (as a four tuple, the details of which will be more apparent after Section 4) encountered for the entire build.

## 4 Overview of CMod’s design

CMod sanitizes modular programming for C. It enforces, in C, the guarantees that modularity provides. C’s module system is informal and has no specification. It is an ad-hoc set of style and coding conventions that have evolved over time and are adhered to by most programmers. Surprisingly, these conventions come very close to approximating what a well designed module system looks like. But because C did not start off having one, it consequently does not enforce one either. This is where CMod comes in. It formalizes using four rules (the details of why they are sound is left out in this instruction manual) what it takes to push C’s idioms of modularity into being provably correct.

The informal and well-known notion of a module system in C is that of treating `.h` header files as interfaces and `.c` source files are corresponding implementations. Textual inclusion of the header files serves to correlate the interface with the implementation. This is illustrated in Figure 1. `bitmap.h` is the interface file for a bitmap module which names a structure and provides declarations for accessor functions. Correspondingly, there is an implementation, `bitmap.c` that provides the definitions for the symbols exported. On the other hand, there is a client of the module, `main.c` that uses the facilities provided by including the module’s interface (the header file) and then using the provided functions and structures.

### 4.1 Modular information-hiding and type-safety

We now talk about the two aspects of a modular design, information-hiding and type-safe linking and how they can be violated in C.

**Type-Safe Linking** Type-Safe linking is one of the cornerstones of any module system. A module system allows users to write separate fragments of a program in separate files and allows separate compilation. What is expected is that when the compiled fragments are put together the mechanisms provided by the module system will enforce type-safety for symbols that are referenced across modules. If this is ensured then the fact that the fragments were compiled separately will have no consequence in terms of type soundness of the linking process.

**Information Hiding** Information hiding is another very important aspect of a language that facilitates modularity. Information hiding states that only the portions of a module that are exported are accessible from outside of the module by code in other modules. This allows a module developer to constantly improve and change the implementation of the module as long as the resulting code still conforms to the module interface.

### bitmap.h

```
struct BM;
void init(struct BM **);
void set(struct BM *, int);
```

### bitmap.c

```
#include "bitmap.h"

struct BM { int data; };
void init(struct BM **map) { ... }
void set(struct BM *map, int bit) { ... }
void private(void) { ... }
```

### main.c

```
#include "bitmap.h"

int main(void) {
    struct BM *bitmap;
    init(&bitmap);
    set(bitmap, 1);
    ...
}
```

Figure 1: Basic C Modules

**Breaking modularity in C** Due to C’s loosely designed idioms concerning modularity, the lack of an enforcement mechanism, does not guarantee type-safe linking. For the example that was shown in Figure ?? we show how modularity properties can be violated in Figure 2. The following violations are illustrated (which *notably* will not yield a warning even under `-Wall` for standard C compilers):

1. The interface `bitmap.h` is not included in the implementation `bitmap.c`
2. Because of the lack of connection between the interface and implementation they differ in the type of `init` violating type-safety.
3. The client (`main.c`) is allowed to violate information hiding by externing a module local symbol `private(void)`.
4. The client (`main.c`) is allowed to define its version of other module’s types e.g. `struct BM` in this case, and consequently allowed to error because of violating type abstraction.

As a consequence of all these violations that are not flagged by the compilers, C’s idiomatic notion of modularity is broken without explicit support from tools such as CMod. CMod rules are designed to be backward compatible with existing C codebases while at the same time providing formal guarantees about type-safety and information hiding. We describe the rules next.

## 4.2 The rules

We provide an informal description of the CMod rules that guarantee type-safe linking. Detailed understanding of these is not required for using CMod but will certainly help in understanding the violations that CMod will report about your code. The formal definitions, proof of correctness, design philosophy and reasoning behind the rules can be found in the papers [1, 2].

CMod concerns itself with proper (type-safe) use of symbols and proper use of types across the files. This corresponds to the first two rules. Additionally, to make sense of these it has to reason about a sane preprocessing environment which is provided if the last two rules are adhered to.

**Definition 1 (Rule 1: Shared Headers)** *Whenever one file links to a symbol defined by another file, both must include a header that contains the declaration of that symbol.*

**Definition 2 (Rule 2: Type Ownership)** *Each type definition in the linked program must be owned by exactly one source or header.*

### bitmap.h

```
struct BM;
void init(struct BM **);
void set(struct BM *, int);
```

### bitmap.c

```
/* bitmap.h not included */

struct BM { int data; };

/* inconsistent declaration */
void init(struct BM *map) { ... }
void set(struct BM *map, int bit) { ... }
void private(void) { ... }
```

### main.c

```
#include "bitmap.h"

/* bad symbol import */
extern void private(void);

/* violating type abstr. */
struct BM { int *data; };

int main(void) {
    struct BM *bitmap;
    init(&bitmap);
    set(bitmap, 1);
    private();
    bitmap->data = ...;
    ...
}
```

Figure 2: Violations of type-safety and information hiding (in C’s idiomatic notion of modules) that are not flagged by the compiler/linker.

**Definition 3 (Rule 3: Proper Inclusion)** *Header files that act as interfaces must not depend on each other, must ignore duplicate inclusions, and must avoid inclusion cycles.*

**Definition 4 (Rule 4: Consistent Environment)** *All files linked together must be compiled in a consistent preprocessor environment.*

## 4.3 Enforcing the rules: The CMod tool

The rules of CMod have been implemented as tool which wraps the standard compiler and linker with a processing stage that enforces the rules and then calls the compiler or linker as appropriate. The architecture of the wrappers is illustrated in Figure 3.

The compiler wrapper, `cwrap`, takes a source file and preprocesses it to find dependency information and stores it for the linking phase. The linker wrapper, `lwrap`, takes the dependency information extracted during the compile phase along with the object file and uses that to check that the program conforms to the rules and emits any violations as warnings.

The details of each of the individual components are omitted for simplification of the document but we direct the interested reader to our technical papers [1, 2] for the details.

## 5 Example: Using CMod over vsftpd-2.0.3

`vsftpd` is a “Very Secure” FTP Daemon written with robustness and security in mind. We use it as an example case because it illustrates the fact that even the most carefully written code can have suspicious practices creep in.

Download and untar the `vsftpd` sources which can be found at <http://vsftpd.beasts.org>. To run CMod over `vsftpd` include CMod into your build path as mentioned in the previous section. Change to the `vsftpd` source directory and compile normally by doing `make`.

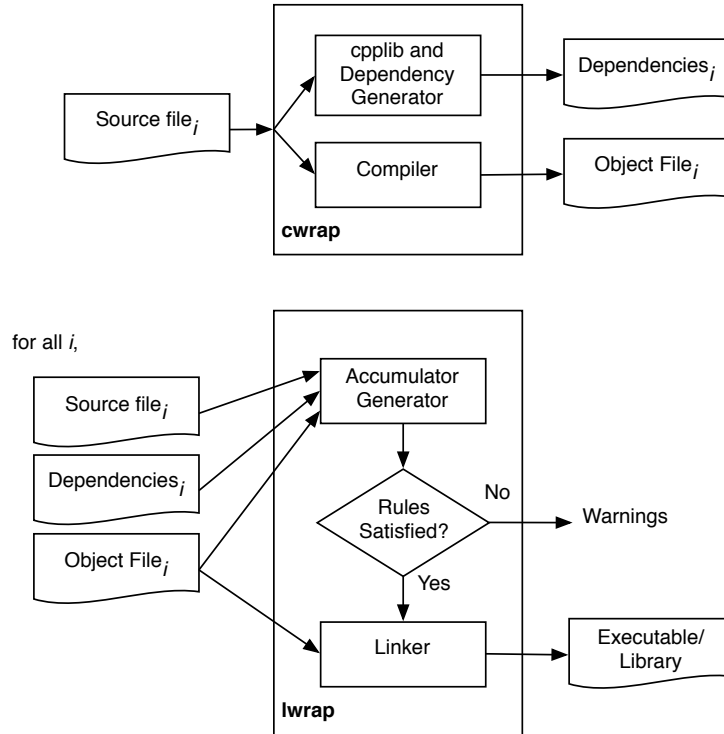


Figure 3: CMod architecture

## 5.1 Interpreting the violations

CMod prints a summary count of the rule violations as a four tuple at the end of the build. The details of the violations are accumulated in separate files so not to clutter the build output on the console. In this section we describe how to interpret the details of the individual violations.

Run `all-rules.sh` in the directory that you compiled the project. This dumps out a summary of the rule violations. For `vsftpd` it should look something like the output shown in Figure 4. We now discuss each of the individual sections in the output dump:

- The table for `RULE 1` warns of potential type-safety errors across modules. In this case they are all due to one module `oneprocess`. Each line on the `RULE 1` section of the dump follows the following format:

< Owning module >    < Client module >    < symbol >

where

- “Owning module” refers to the module defining the symbol
- “Client module” refers to the client module using the symbol
- “symbol” is the errant symbol that appears in both files

In this particular case, what we diagnosed was the the exporting module `oneprocess.c` does not include its own interface file `oneprocess.h`. While the clients, `main.c`, `prelogin.c`, `postlogin.c` and `ftpdataio.c` include the interface, the compiler has no way of correlating that with the module that actually defines those symbols and therefore has no way of type-checking that they are actually defined with the required type. This illustrates that disconnect between the implementation and interface that is allowed by C. CMod points it out and by fixing such errors the developers can ensure that their projects are not susceptible to errors creeping in as the project matures.

```

===== RULE 1 =====
oneprocess.o    main.o          vsf_one_process_start
oneprocess.o    prelogin.o     vsf_one_process_login
oneprocess.o    postlogin.o    vsf_one_process_chown_upload
oneprocess.o    ftpdataio.o   vsf_one_process_get_priv_data_sock
===== RULE 2 =====
===== RULE 3 =====
filesize.h:session.h
filesize.h:str.h
filesize.h:sysdeputil.h
filesize.h:sysutil.h
filestr.c:str.h
session.h:str.h
str.c:str.h
strlist.c:strlist.h
sysutil.c:sysutil.h
===== RULE 4 =====
You should have resolved for the compile to go through.

```

Figure 4: Summary of rule violations for vsftpd.

- They were no RULE 2 violations for vsftpd but if there are then they are dumped out in the following format:

```

< typedef|struct >    < type-name >    < fileA >    < location >
< typedef|struct >    < type-name >    < fileB >    < location >

```

where each pair of lines indicates a definition of a type that appears in two places (and hence the pair of lines). Each line of the pair indicates one location in “fileA” or “fileB” where a conflicting definition of the type “type-name” appears which is either a “typedef” or “struct”.

- Recall that RULE 3 violations are ones in which one header file “depends” on another and so there can be problems when you switch the order in the places they are included. A file `fileA` depends on another `fileB` if `fileA` `#define/#undefs` a CPP symbol that is used (by being the guard in an `#ifdef` or being expanded as a macro etc) by `fileB` or vice versa. Such dependencies are dangerous as switching the order of inclusion causes the files to preprocess differently. CMod points out these order dependencies by listing pairs of files which depend on each other in the following format:

```
< fileA >:< fileB >
```

For vsftpd there are 9 violations of this form as shown in the output. The summary output does not include the details of all the CPP symbols that caused the violations. There are located separately in files named `object.cmod.inc`, where `object` is the name of the linked object. In the case of vsftpd our final linked object is vsftpd and so the file is named `vsftpd.cmod.inc`. Each line in a `.cmod.inc` file is of the following format:

```

< id >< macro symbol > in < source file >    :
                                < headerA >:< lnoA >  === < headerB >:< lnoB >

```

where `macro symbol` is the name of the macro which causes the dependency, `source file` is the `.c` file which includes two headers `headerA` and `headerB` that are dependent. The corresponding line numbers in the headers indicate the locations where the macro symbol is defined (correspondingly used in the other). on each other.

- **RULE 4** violations are handled a little differently. If you were to read the technical papers you will realize that rule 4 violations (the ones that involve the preprocessing to happen in a consistent environment) are critical to the soundness of the tool. In particular, all other rules do not make sense unless there are no rule 4 violations. Therefore we do not allow the compile to proceed if there are any rule 4 violations (which are rare). So the compile aborts as soon as a rule 4 violations is encountered. Usually this happens when some `fileA.c` is compiled with `-DXXX` while you forget to include `-DXXX` on the command line for `fileB.c` and you link `fileA.o` and `fileB.o` together. This can cause all sorts of problems while preprocessing. We have not found reasonable instance amongst the million lines of open source code that we have examined where this difference in preprocessing environment is essential to the project. So we would be surprised if your project required it.

## 6 Helper scripts

We now describe some scripts that are packaged alongwith CMod which facilitate the task of analyzing your source code.

1. `all-rules.sh`: As mentioned earlier, this script accumulates all the rule violations and prints out a summary for the same.
2. `common-hdrs.sh`: This computes a list of headers shared by two source files `fileA.c` and `fileB.c`. i.e. headers that are included by both files either through a direct `#include` or transitively by inclusion through some other header. For example, in the case of the Rule 1 violations in `vsftpd`, `oneprocess.c` and `main.c` did not share a header that declared `oneprocess`'s symbols. Given the symbol it is easy to find out which header files declare it. But it is not easy to figure out if one or both the source files are actually including the relevant headers. If you run `common-hdrs.sh oneprocess.D main.D` (notice that we use the `.D` files instead of the `.c`, because the `.D` contains the CMod meta data) it outputs:

```
/tmp/vsftpd-2.0.3/filesize.h
/tmp/vsftpd-2.0.3/session.h
/tmp/vsftpd-2.0.3/str.h
/tmp/vsftpd-2.0.3/sysdeputil.h
/tmp/vsftpd-2.0.3/tunables.h
/tmp/vsftpd-2.0.3/utility.h
```

indicating that the interface file `oneprocess.h` is not a shared include. Then you can look inside `oneprocess.D` and the last line lists all the includes and it is evident that it fails to include its own interface.

3. `list-exes.sh`: Lists all the linked executables linked through `ld` and `ar` (for libraries).
4. `list-hdrs.sh`: Lists all the headers that were preprocessed i.e. all headers that were included through some source file during the compile.
5. `list-srcs.sh`: Lists all source files that were processed during the build.
6. `list-incs.sh`: Lists the output data file for **RULE 3** violations.
7. `list-syms.sh`: Lists the output data file for **RULE 1** violations.
8. `prj-stats.sh`: Computes statistics for the project. Lists LoC, number of source files, number of header files, number of final linked targets.
9. `cleanall-cmod.sh`: Removes all temporary and CMod output files from the build directory. Use it to clean up after doing the projects native `make clean`.

## References

- [1] S. Srivastava, M. Hicks, and J. S. Foster. Modular information hiding and type safety for C. In *Proceedings of the ACM Workshop on Types in Language Design and Implementation (TLDI)*, pages 3–14, January 2007.
- [2] S. Srivastava, M. Hicks, J. S. Foster, and P. Jenkins. Modular information hiding and type safe linking for C, June 2007. Submitted to IEEE Transactions on Software Engineering. Full version of TLDI 07 paper.

## A Setting Up GCC Sources

CMod requires GCC sources to build and link against because it delegates preprocessing tasks to the CPP library. The source distribution of CMod packages the appropriate tarball of the GCC sources in the `gcc-patch/` directory. We now describe the steps to setup the sources:

- Choose a directory where you will keep the GCC sources. Let us call this directory `<DIRGCC>`. Chdir to `<DIRGCC>` and untar the `gcc-core-3.3.tar.gz` tarball there creating `<DIRGCC>/gcc-3.3/`.
- In `<DIRGCC>` again, create another directory called `gcc-build`.
- Chdir into `<DIRGCC>/gcc-build/` and run `<DIRGCC>/gcc-3.3/configure`. Notice the difference in directories locations. This sets up `gcc-build` as your source directory.