

How to Use the Plan 9 Comeau C++ and Comeau C Compiler

An Informative Parody

Greg Comeau
comeau@comeaucomputing.com

Introduction

The Comeau compiler on Plan 9 is a wholly new program; in fact it was one of the last pieces of software written in 2012 for Plan 9. :) Programmers familiar with existing C++ and C compilers will find a number of similarities and differences in both the language the Plan 9 Comeau compiler accepts and in how the compiler is used.

The Comeau compiler is really a set of compilers, one for each architecture — currently Intel 386 — that accept various dialects of ISO C++ and ISO C and efficiently produce fairly good code for the target machine. The main interface for native code programming is the `como` command. There is a packaging of the Comeau compiler that accepts strict ISO C++ and ISO C for a POSIX environment via the `comoape` command, but this document focuses on the native Plan 9 environment via the `como` command, that in which all the system source and almost all the utilities are written. However, lots of the below discussion will apply to both high level Comeau compiler sets. For some additional information see <http://www.comeaucomputing.com/4.3.0/minor/plan9/como43101.html>.

Source

The core language accepted by the Comeau compilers is ISO C++ and ISO C in various flavors and dialects. For instance, Comeau C supports the core language of C99 (via `--c99`) and of C89 (via `--c`). In regards to C++, Comeau C++ supports the core language of C++03 (via `--c++` which is the default mode of the Comeau compilers). There is also a number of other different dialects, modes and settings available.

Note that library-wise, for C we use what is available and provided in the core Plan 9 libraries reflecting Standard C as the basis. For C++, we make `libcomo` available for obtaining access to `iostreams`, `STL`, etc.

As regards Plan 9, our goal is to support the core language specification of Standard C++ and Standard C, while also supporting Plan 9 programming. For instance, as regards native Plan 9 programming, there is a completely different structure for include files, as Plan 9 C programmers writing native mode applications already know.

Note that Plan 9 C compilers provided with Plan 9 make a number of modest extensions, and include a greatly simplified preprocessor. Neither Comeau C++ nor Comeau C currently provide those extensions or simplifications as such, since our goal is to provide support for the core language specification of Standard C++ and Standard C.

That said, the two opposing "goals" rarely need to bump heads per se. For instance, the provided Plan 9 C compilers by default diagnose so-called K&R "old" style function declarations. ISO C90 does not require this to be diagnosed. However, C99 and C++ do. Therefore, `como --c` does not by default issue a diagnostic for this ("lack of") construct, whereas `como --c99` and `como --c++` (the default `como` language setting) will. However, that said, as a quality of implementation concern, Comeau does allow some warnings to be emitted when desired and even turned into errors. In this case, a programmer could enforce the Plan 9 default by using `como --c 'diag_error=implicit_func_decl'`. So, for instance, if you were to have this program:

```

term% cat noproto.c
int main()
{
    float f = 99.99;
    /* Let's say undeclared_function() really take a char */
    undeclared_function(f); /* This then is just trouble waiting to happen */

    return 0;
}

```

The Plan 9 C compiler, Comeau C99, and Comeau C++ will all diagnose this, however, to obtain the diagnostic in Comeau C90 you would use:

```

term% como -A --c '--diag_error=implicit_func_decl' noproto.c
Comeau C/C++ 4.3.10.1 (Apr 21 2012 16:32:28) for Plan9
Copyright 1988-2012 Comeau Computing. All rights reserved.
MODE:strict errors [386-native] C90

noproto.c:5 error: function "undeclared_function" declared implicitly
    undeclared_function(f); /* This then is just trouble waiting to happen */
    ^

1 error detected in the compilation of "noproto.c".
term%

```

There are also other options. For instance when Comeau C99 is not in strict mode (enabled via the `-A` option) it issues a warning instead of an error. You can also have Comeau C90 issue a warning instead of an error by using `'--diag_warning=implicit_func_decl'`. Of course, the author of the code can make choices avoiding this type of issue altogether (in either the provided Plan 9 C compilers or the Comeau compilers) by using function prototypes, `#include`'ing appropriate header files, etc.

Note that the Plan 9 provided C compilers don't accept `#if` nor `##`. However, as it is often that code originally written for other systems uses these constructs, the Plan 9 `/bin/cpp` command accepts them, as does the Comeau Plan 9 compilers.

As of this writing, the Comeau compilers attempt to make use of the header files and libraries already provided by the Plan 9 provided C compilers. That includes many of the the machine-independent ones in `/sys/include`, and many of the machine dependent ones such as in `/o/include` where `o` is the appropriate machine, for instance for Intel 386 that would be `/386/include` and `/386/include/ape`. In many cases, we have needed to apply "some magic" in order for that to be successful both in and of itself, but also so that the headers can be used by Comeau C90, Comeau C99, and Comeau C++ without any need for commotion by the end-user programmer. That is certainly the intent, some work no doubt remains towards this goal.

Therefore, the intent is that you are free to use `nil`, `vlong`, `u32int`, etc. so long as you `#include` the correct headers, say `<u.h>`, etc. Of course this is also so long as what you are doing is appropriate for the language at hand. For instance, C++ would treat/dis/allow `nil` in different contexts than it is treated/dis/allowed in C (so for instance, in C++, you can't pass `nil` to `exits()` since `exits` take a `char *` argument).

As per Rob Pike's article *How to Use the Plan 9 C Compiler*, the suggestion for the provided Plan 9 C compilers is that "Every Plan 9 C program begins with:

```
#include <u.h>
```

because all the other installed header files use the `typedefs` declared in `<u.h>`." And then from there the usual is to provide respective Plan 9 include files that reflects a library, which often provides a broader declaration sweep than say ISO C calls for. For instance, instead of `<string.h>`, `<memory.h>`, etc you might

just request `<libc.h>`. The Comeau compilers for Plan 9 allow you to take this same approach if you wish. As an example:

```
term% cat ij.c
#include <u.h>
#include <libc.h>

int i = 99;
int j(-i);

int main()
{
    print("i=%d\n", i);
    print("j=%d\n", j);

    return 0;
}
term% como -A --long_long ij.c
Comeau C/C++ 4.3.10.1 (Apr 21 2012 16:32:28) for Plan9
Copyright 1988-2012 Comeau Computing. All rights reserved.
MODE:strict errors [386-native] C++ noC++0x_extensions

term% 8.out
i=99
j=-99
term%
```

Note the use of the `--long_long` command line option. This is needed in strict C++ mode because unlike C, C++ does not have a `long long` type, and so you'll need to enable this as a Comeau extension in some cases.

Also note that this test program uses `print` and not `printf`. The former is a Plan 9 routine while the latter is a Standard C (and C++) routine. If you were to want to use `printf` a Plan 9 programmer might code:

```
term% cat ijprintf.c
#include <u.h>
#include <libc.h>
#include <stdio.h>

int i = 99;
int j(-i);

int main()
{
    printf("i=%d\n", i);
    printf("j=%d\n", j);

    return 0;
}
term% como ijprintf.c
Comeau C/C++ 4.3.10.1 (Apr 21 2012 16:32:28) for Plan9
Copyright 1988-2012 Comeau Computing. All rights reserved.
MODE:non-strict warnings [386-native] C++ noC++0x_extensions

term% 8.out
i=99
j=-99
term%
```

This is being compiled as a C++ program, however, it is as a Plan 9 C programmer would expect it to be.

That said, because Standard C and Standard C++ do not have a requirement involving `u.h` etc, the Comeau compiler also allows for you to just use `stdio.h` as well. This allows code which did not originate as a native Plan 9 app to still have a more fighting chance of compiling without commotion and without the 100% requirement of using `comoape` the APE supporting Comeau compiler. It will be interesting to see how this negotiates the test of time. And similar to an earlier comment, this is currently the intent, and more work no doubt remains towards this goal.

Plan 9 C programs also differ from Standard C and Standard C++ in that normally in Plan 9 `main()` is declared as "returning" `void`. Also, normally it is expected that the program eventually calls the `exits` routine. By default the Comeau compilers diagnose this instead wanting to favor the `int main()/return 0;` type of sequence. In cases where you want to model the Plan 9 C compiler behavior, feel free to use the `'--diag_suppress=bad_return_type_on_main'` command line option. This becomes a bit of a misnomer since it's not "bad" in Plan 9. Also, note that the provided Plan 9 C compilers allow you to use the Standard C and Standard C++ type of sequence, however, we assume this yields a kind of Plan 9 undefined behavior for Plan 9 because there is no return value.

Process

The provided Plan 9 C compilers are named involving an alphanumeric prefixing character, let's call it `o` which represents a specific CPU architecture that is supported by Plan 9. This comes into play because that character is often prefixed to command names and files to establish a certain convention, for instance `oc` to invoke the compiler, `ol` to invoke the linker, and `o.out` to run the executable program image. In the case of the 386 `o` is assigned the value `8`. Therefore to use these the names become `8c`, `8l` and `8.out` respectively as well as others for instance `file.8` is expected to be an object file for the 386 that might have been compiled by `8c`.

Where possible, the Comeau compilers attempt to follow some of these convention as well. As of this writing, the Comeau compiler is available for the 386 and is invoked by running the `como` command. Linking is also done by invoking the `como` command -- do not use `8l`. Object files produced via `como` (currently for Intel 386 only) are named with the `.8` suffix and the default executable program image file is named `8.out`.

Here's a simple side-by-side comparison of command invocations using the Plan 9 provided 386 compiler and the current Comeau compiler:

Plan 9 example	Comeau example
<code>8c file.c</code>	<code>como -c file.c</code>
<code>8l file.8</code>	<code>como file.8</code>
<code>8.out</code>	<code>8.out</code>

Note that when using the Comeau compiler for APE programming use `comoape` instead of `como`, in which case avoid `.8` files since the naming convention for object files in that case is `.o`.

As we add more Plan 9 Comeau compilers, we expect to follow the command invocation conventions, so for instance, `5comoape` would invoke the ARM version of the Comeau compiler for APE programming, `qcomo` would invoke the Power PC version of the Comeau compiler for native programming, and so on.

If you are compiling with the Comeau compiler and using `mk` then you'll obviously need to make some changes or additions to accommodate Comeau, its command line options, etc., since now you have more choices than just the provided Plan 9 C compilers and linkers.

Heterogeneity

Although the Comeau compiler for Plan 9 is currently only available for Intel 386, it will use many of the same conventions of the actual provided Plan 9 C compilers. The intent is a similar composition. This will render things not only philosophically similar, but just makes things physically easier as well. Therefore, for instance, it assumes `$cputype` as well. In the case of Intel 386 this is `386` and so therefore in the case of this CPU architecture, it will also be making use of `/386` and its subdirectories when appropriate and where necessary.

Heterogeneity and `mk`

In addition to the semantics of `$cputype`, the provided Plan 9 C compilers can be controlled to create target binary program images for CPU architectures that are different from the host CPU architecture. This target is controlled via a second variable `$objtype`. However, as currently the Comeau C/C++ compiler is currently only available for Intel 386, for our use it will also be `386` as well. Since currently `$cputype` and `$objtype` will be the same for Comeau, they should not be modified for use with the Comeau compilers or when using the Comeau compilers with `mk`. To say this in a different way, there is no need for the end user programmer to change them just to invoke `como` (or `comoape`) itself.

Portability

Comeau C/C++ for Plan 9 provides for native Plan 9 programming via its `como` interface. Therefore, our intent/goal is to also make portable programming painless, and to strive for machine independence when necessary and possible. Hopefully our piggy-backing approach is able to achieve this goal and render Plan 9 development with Comeau an enjoyable and seamless as possible endeavor.

Of course, as with the provided Plan 9 C compilers, when the going gets tough, the tough get going, and so, just as in cases where you need to move away from native Plan 9 programming per se via say ANSI C and POSIX ala APE you'd use `pcc`, we also provide an alternate interface via `comoape`.

I/O

`man bio` provides descriptions for various library routines for buffered I/O, which can be interfaced via `<bio.h>`. There appears to be an allowance in the provided Plan 9 C compilers which briefly stated is that pointers to two different structs are compatible so long as the initial member sequence is identical. This creates an anomaly for both Standard C and Standard C++ when using the `bio` library. We believe we have created the basic infrastructure to satisfy this regarding Standard C++, though it still needs to be extended.

However with Standard C you will need to use the command line option `--svr4` in addition to the normal `--c`. Depending upon your needs that may not be the C mode you want to be in. As well, even in this mode, the Comeau C compilers will still generate a warning:

```
warning: argument of type "Biobuf *" is incompatible with parameter of type "Biobufhdr *"
```

This warning can be removed through the use of the `'--diag_suppress=incompatible_param'` command line option, though silencing all such incompatibilities whether pointer to struct or not.

Arguments

`man arg` provides descriptions of 4 macros provided in `<libc.h>` for native code apps dealing with command line arguments. This allows you to capture the current "option character" via `ARGC` or the current option string via `ARGF` where `case` statements are wrapped by a `ARGBEGIN/ARGEND` pair which establishes a loop and switch statement in order to interrogate the arguments. Machinery is in place whereas this should work transparently with the respective Comeau compilers.

Extensions

The provided Plan 9 C compilers offer a few extension to C90. One such extension is "struct displays" which allow for the dynamic creation of struct-based expressions. It is established by what looks like the initializer for a structure prefixed by a cast. This was the basis for Standard C99's "compound literals" (see <http://comeaucomputing.com/techtalk/c99/#compoundliterals>) which is supported by Comeau C99 (`-c99`).

The provided Plan 9 C compilers also support anonymous structs and anonymous unions. This features provides some syntactic sugar whereas the nested anonymous entity can have its members referenced in an expression without needing to prefix an outer name. The Comeau C++ compilers support anonymous unions as per Standard C++. Although the Comeau C compilers do support anonymous unions and structs on other platforms and dialects it is not currently supported by Comeau C for Plan 9. [Contact us](#) if you would like to have this added as a customization.

Another extension to C90 that the provided Plan 9 C compilers support is the ability to initialize specific array elements and to initialize specific struct elements through additional syntax shorthands. This was the basis for Standard C99's "designated initializers" (see <http://comeaucomputing.com/techtalk/c99/#di>) which is supported by Comeau C99 (`--c99`).

The provided Plan 9 C compilers also support `extern register`; the Comeau compilers do not currently support this capability.

The compile-time environment

The provided Plan 9 C compilers emit warnings in many contexts. Similarly, the Comeau C and Comeau C++ compilers emit various warnings which can be controlled in many different ways. Some warning can be controlled specifically, for instance, in C++ you can enable or disable a warning that is issued when programs compiled under the new for-init scoping rules would have had different behavior under the old rules by using `--for_init_diff_warning`.

You can also turn off all warnings via `--no_warnings`. You can also have the Comeau compilers emit diagnostics normally considered weaker than warnings via the `--remarks` option though sometimes this is noisier than one would prefer.

You can also have other warning silenced, for instance, a used and not set situation via `--no_used_before_set_warnings`.

The Plan 9 C compilers support leaving off the identifier name in the declaration of a parameter to a function in the case of unused parameters. This is supported in C++ mode of the Comeau compilers. The provided Plan 9 C compilers also support the `SET(identifiers)` and `USED(identifiers)` statements; there is no such statement in the Comeau compilers.

Debugging

The average Plan 9 C programmer probably prefers to use the `acid` debugger. Normally this is done via passing an argument to `acid` which is the process id number of the command which failed, as once a Plan 9 process dies, it's still left around so to speak. Programmers using Comeau C++ can also use `acid` to debug their programs, however note there is some similarities and some differences. Let's take an example:

```
1  #include <u.h>
2  #include <stdio.h>
```

```

3
4 void foo(void)
5 {
6     char *p = 0;
7
8     printf("in foo\n");
9     *p = 'x';
10    printf("out foo\n");
11 }
12
13 int main()
14 {
15     printf("hi\n");
16     printf("hello\n");
17     foo();
18     printf("goodbye\n");
19     return 0;
20 }

```

This program should fail once it goes to write the x character into where p points to, since it's a null pointer. So let's compile it:

```

term% como --c bug.c
Comeau C/C++ 4.3.10.1 (Apr 21 2012 16:32:28) for Plan9
Copyright 1988-2012 Comeau Computing. All rights reserved.
MODE:non-strict warnings [386-native] C90

```

and now run it:

```

term% 8.out
hi
hello
in foo
8.out 149: suicide: sys: trap: page fault pc=0x0000103c

```

The failed process id is 149, so let's debug it. We can now run acid on it:

```

term% acid 149
/proc/149/text:386 plan 9 executable
/sys/lib/acid/port
/sys/lib/acid/386
acid: stk()
foo()+0x1c /tmp/htu/bug.c:9
main()+0x22 /tmp/htu/bug.c:18
_main+0x31 /sys/src/libc/386/main9.s:16
acid:

```

This is similar to what one would expect with the provided Plan 9 C compilers. However, if we compile it as a C++ program instead of a C program we get:

```

term% como bug.c
Comeau C/C++ 4.3.10.1 (Apr 21 2012 16:32:28) for Plan9
Copyright 1988-2012 Comeau Computing. All rights reserved.
MODE:non-strict warnings [386-native] C++ noC++0x_extensions

```

```

term% 8.out
hi
hello
in foo
8.out 255: suicide: sys: trap: page fault pc=0x0000103c

```

```

term% acid 255
/proc/255/text:386 plan 9 executable
/sys/lib/acid/port
/sys/lib/acid/386
acid: stk()
foo__Fv()+0x1c bug.c:9
main()+0x27 bug.c:18
_main+0x31 /sys/src/libc/386/main9.s:16
acid: src(*PC)
/tmp/htu/bug.c:9
4     void foo(void)
5     {
6     char *p = 0;
7
8     printf("in foo\n");
>9    *p = 'x';
10    printf("out foo\n");
11    }
12
13    int main()
14    {
acid:

```

Note that we can display the line in error.

Note also that the difference is that function `foo` is now named `foo__Fv`, this is because of C++ name mangling. However, note that you can also do this:

```

term% acid 255 | decode43101.CC -u
/proc/255/text:386 plan 9 executable
/sys/lib/acid/port
/sys/lib/acid/386
stk()
acid: foo()()+0x1c bug.c:9
main()+0x27 bug.c:18
_main+0x31 /sys/src/libc/386/main9.s:16

```

with two caveats: the function name now prints with two ()'s, and, there is a buffering problem between `acid` and `decode43101.CC`, although at least the decoder will now make it easier to see which function it is in.

The above is a naive discussion. Debugging and `acid` can be fairly involved so we direct you to `man acid` as well as the *Acid Manual*. As well, note that there will be different interactions between `acid` and C++ features.

[Comeau Computing](http://www.comeaucomputing.com)

<http://www.comeaucomputing.com>

<http://www.comeaucomputing.com/4.3.0/minor/plan9/como43101.html>

support@comeaucomputing.com

Last updated April 12, 2012

(c) © 1992-2012 Comeau Computing. All Rights Reserved.