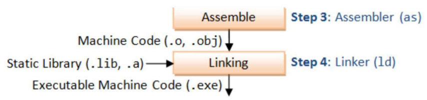

GCC compiles a C/C++ program into executable in 4 steps as shown in the above diagram. For example, a "`gcc -o hello.exe hello.c`" is carried out as follows:

1. Pre-processing: via the GNU C Preprocessor (`cpp.exe`), which includes the headers (`#include`) and expands the macros (`#define`).

   ```
   > cpp hello.c > hello.i
   ```

   The resultant intermediate file "`hello.i`" contains the expanded source code.

2. Compilation: The compiler compiles the pre-processed source code into assembly code for a specific processor.

   ```
   > gcc -S hello.i
   ```

   The `-S` option specifies to produce assembly code, instead of object code. The resultant assembly file is "`hello.s`".

3. Assembly: The assembler (`as.exe`) converts the assembly code into machine code in the object file "`hello.o`".

   ```
   > as -o hello.o hello.s
   ```

4. Linker: Finally, the linker (`ld.exe`) links the object code with the library code to produce an executable file "`hello.exe`".

   ```
   > ld -o hello.exe hello.o ...libraries...
   ```

Verbose Mode (-v)

You can see the detailed compilation process by enabling `-v` (verbose) option. For example,

```
> gcc -v hello.c -o hello.exe
```

Defining Macro (-D)

You can use the -D*name* option to define a macro, or -D*name*=*value* to define a macro with a value. The *value* should be enclosed in double quotes if it contains spaces.

1.5 Headers (.h), Static Libraries (.lib, .a) and Shared Library (.dll, .so)

Static Library vs. Shared Library

A library is a collection of pre-compiled object files that can be linked into your programs via the linker. Examples are the system functions such as `printf()` and `sqrt()`.

There are two types of external libraries: *static library* and *shared library*.

1. A static library has file extension of "`.a`" (archive file) in Unixes or "`.lib`" (library) in Windows. When your program is linked against a static library, the machine code of external functions used in your program is copied into the executable. A static library can be created via the *archive* program "`ar.exe`".
2. A shared library has file extension of "`.so`" (shared objects) in Unixes or "`.dll`" (dynamic link library) in Windows. When your program is linked against a shared library, only a small table is created in the executable. Before the executable starts running, the operating system loads the machine code needed for the external functions - a process known as *dynamic linking*. Dynamic linking makes executable files smaller and saves disk space, because one copy of a library can be shared between multiple programs. Furthermore, most operating systems allows one copy of a shared library in memory to be used by all running programs, thus, saving memory. The shared library codes can be upgraded without the need to recompile your program.

Because of the advantage of dynamic linking, GCC, by default, links to the shared library if it is available.

You can list the contents of a library via "nm *filename*".

You can list the contents of a library via "nm *filename*".

Searching for Header Files and Libraries (-I, -L and -l)

When compiling the program, the *compiler* needs the *header files* to compile the source codes; the *linker* needs the *libraries* to resolve external references from other object files or libraries. The compiler and linker will not find the headers/libraries unless you set the appropriate options, which is not obvious for first-time user.

For each of the headers used in your source (via `#include` directives), the compiler searches the so-called *include-paths* for these headers. The include-paths are specified via -I*dir* option (or environment variable `CPATH`). Since the header's filename is known (e.g., `iostream.h`, `stdio.h`), the compiler only needs the directories.

The linker searches the so-called *library-paths* for libraries needed to link the program into an executable. The library-path is specified via -L*dir* option (uppercase `'L'` followed by the directory path) (or environment variable `LIBRARY_PATH`). In addition, you also have to specify the library name. In Unixes, the library lib*xxx*.a is specified via -l*xxx* option (lowercase letter `'l'`, without the prefix "`lib`" and "`.a`" extension). In Windows, provide the full name such as `-lxxx.lib`. The linker needs to know both the directories as well as the library names. Hence, two options need to be specified.

Default Include-paths, Library-paths and Libraries

Try list the default include-paths in your system used by the "GNU C Preprocessor" via "`cpp -v`":

```
> cpp -v
......
#include "..." search starts here:
#include <...> search starts here:
 d:\mingw\bin\../lib/gcc/mingw32/4.6.2/include          // d:\mingw\lib\gcc\mingw32\4.6.2\include
 d:\mingw\bin\../lib/gcc/mingw32/4.6.2/../../../../include // d:\mingw\include
 d:\mingw\bin\../lib/gcc/mingw32/4.6.2/include-fixed    // d:\mingw\lib\gcc\mingw32\4.6.2\include-fixed
```

Try running the compilation in verbose mode (`-v`) to study the library-paths (`-L`) and libraries (`-l`) used in your system:

```
> gcc -v -o hello.exe hello.c
......
-Ld:/mingw/bin/../lib/gcc/mingw32/4.6.2                    // d:\mingw\lib\gcc\mingw32\4.6.2
-Ld:/mingw/bin/../lib/gcc                                  // d:\mingw\lib\gcc
-Ld:/mingw/bin/../lib/gcc/mingw32/4.6.2/../../../../mingw32/lib // d:\mingw\mingw32\lib
-Ld:/mingw/bin/../lib/gcc/mingw32/4.6.2/../../..           // d:\mingw\lib
-lmingw32     // libmingw32.a
-lgcc_eh      // libgcc_eh.a
-lgcc         // libgcc.a
-lmoldname
-lmingwex
-lmsvcrt
-ladvapi32
-lshell32
-luser32
-lkernel32
```

Eclipse CDT: In Eclipse CDT, you can set the include paths, library paths and libraries by right-click on the project ⇒ Properties ⇒ C/C++ General ⇒ Paths and Symbols ⇒ Under tabs "Includes", "Library Paths" and "Libraries". The settings are applicable to the selected project only.

1.6 GCC Environment Variables

GCC uses the following environment variables:

- `PATH`: For searching the executables and run-time shared libraries (`.dll`, `.so`).
- `CPATH`: For searching the *include-paths* for headers. It is searched after paths specified in -I<*dir*> options. `C_INCLUDE_PATH` and `CPLUS_INCLUDE_PATH` can be used to specify C and C++ headers if the particular language was indicated in pre-processing.
- `LIBRARY_PATH`: For searching *library-paths* for link libraries. It is searched after paths specified in -L<*dir*> options.

2.1 First Makefile By Example

2.1 First Makefile By Example

Let's begin with a simple example to build the Hello-world program (`hello.c`) into executable (`hello.exe`) via make utility.

```
// hello.c
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
```

Create the following file named "makefile" (without any file extension), which contains rules to build the executable, and save in the same directory as the source file. Use "tab" to indent the command (NOT spaces).

```
all: hello.exe

hello.exe: hello.o
	gcc -o hello.exe hello.o

hello.o: hello.c
	gcc -c hello.c

clean:
	rm hello.o hello.exe
```

Run the "`make`" utility as follows:

```
> make
gcc -c hello.c
gcc -o hello.exe hello.o
```

Running `make` without argument starts the target "`all`" in the `makefile`. A makefile consists of a set of rules. A rule consists of 3 parts: a target, a list of pre-requisites and a command, as follows:

```
target: pre-req-1 pre-req-2 ...
	command
```

The *target* and *pre-requisites* are separated by a colon (`:`). The *command* must be preceded by a tab (NOT spaces).

When `make` is asked to evaluate a rule, it begins by finding the files in the prerequisites. If any of the prerequisites has an associated rule, make attempts to update those first.

In the above example, the rule "`all`" has a pre-requisite "`hello.exe`". `make` cannot find the file "`hello.exe`", so it looks for a rule to create it. The rule "`hello.exe`" has a pre-requisite "`hello.o`". Again, it does not exist, so `make` looks for a rule to create it. The rule "`hello.o`" has a pre-requisite "`hello.c`". `make` checks that "`hello.c`" exists and it is newer than the target (which does not exist). It runs the command "`gcc -c hello.c`". The rule "`hello.exe`" then run its command "`gcc -o hello.exe hello.o`". Finally, the rule "`all`" does nothing.

More importantly, if the pre-requisite is not newer than than target, the command will not be run. In other words, the command will be run only if the target is out-dated compared with its pre-requisite. For example, if we re-run the make command:

```
> make
make: Nothing to be done for `all'.
```

You can also specify the target to be made in the `make` command. For example, the target "`clean`" removes the "`hello.o`" and "`hello.exe`". You can then run the `make` without target, which is the same as "`make all`".

```
> make clean
rm hello.o hello.exe

> make
gcc -c hello.c
gcc -o hello.exe hello.o
```


Try modifying the "`hello.c`" and run `make`.

NOTES:

- If the *command* is not preceded by a tab, you get an error message "makefile:4: *** missing separator. Stop."
- If there is no `makefile` in the current directory, you get an error message "make: *** No targets specified and no makefile found. Stop."
- The makefile can be named "`makefile`", "`Makefile`" or "`GNUMakefile`", without file extension.

2.2 More on Makefile

Comment & Continuation

A comment begins with a `#` and lasts till the end of the line. Long line can be broken and continued in several lines via a back-slash (`\`).

Syntax of Rules

A general syntax for the rules is:

```
target1 [target2 ...]: [pre-req-1 pre-req-2 ...]
	[command1
	 command2
	 ......]
```

The rules are usually organized in such as way the more general rules come first. The overall rule is often name "`all`", which is the default target for `make`.

Phony Targets (or Artificial Targets)

A target that does not represent a file is called a phony target. For example, the "`clean`" in the above example, which is just a label for a command. If the target is a file, it will be checked against its pre-requisite for out-of-date-ness. Phony target is always out-of-date and its command will be run. The standard phony targets are: `all`, `clean`, `install`.

Variables

A variable begins with a `$` and is enclosed within parentheses `(...)` or braces `{...}`. Single character variables do not need the parentheses. For example, `$(CC)`, `$(CC_FLAGS)`, `$@`, `$^`.

Automatic Variables

Automatic variables are set by make after a rule is matched. There include:

- `$@`: the target filename.
- `$*`: the target filename without the file extension.
- `$<`: the first prerequisite filename.
- `$^`: the filenames of all the prerequisites, separated by spaces, discard duplicates.
- `$+`: similar to `$^`, but includes duplicates.
- `$?`: the names of all prerequisites that are newer than the target, separated by spaces.

For example, we can rewrite the earlier makefile as:

```
all: hello.exe

# $@ matches the target; $< matches the first dependent
hello.exe: hello.o
	gcc -o $@ $<

hello.o: hello.c
	gcc -c $<

clean:
	rm hello.o hello.exe
```

Virtual Path - VPATH & vpath

You can use VPATH (uppercase) to specify the directory to search for dependencies and target files. For example,

```
# Search for dependencies and targets from "src" and "include" directories
# The directories are separated by space
VPATH = src include
```

You can also use `vpath` (lowercase) to be more precise about the file type and its search directory. For example,

```
# Search for .c files in "src" directory; .h files in "include" directory
# The pattern matching character '%' matches filename without the extension
vpath %.c src
vpath %.h include
```

Pattern Rules

A pattern rule, which uses pattern matching character `'%'` as the filename, can be applied to create a target, if there is no explicit rule. For example,

```
# Applicable for create .o object file.
# '%' matches filename.
# $< is the first pre-requisite
# $(COMPILE.c) consists of compiler name and compiler options
# $(OUTPUT_OPTIONS) could be -o $@
%.o: %.c
        $(COMPILE.c) $(OUTPUT_OPTION) $<

# Applicable for create executable (without extension) from object .o object file
# $^ matches all the pre-requisites (no duplicates)
%: %.o
$(LINK.o) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

Implicit Pattern Rules

Make comes with a huge set of implicit pattern rules. You can list all the rule via `--print-data-base` option.

2.3 A Sample Makefile

This sample makefile is extracted from Eclipse's "C/C++ Development Guide -Makefile".

```
# A sample Makefile
# This Makefile demonstrates and explains
# Make Macros, Macro Expansions,
# Rules, Targets, Dependencies, Commands, Goals
# Artificial Targets, Pattern Rule, Dependency Rule.

# Comments start with a # and go to the end of the line.

# Here is a simple Make Macro.
LINK_TARGET = test_me.exe

# Here is a Make Macro that uses the backslash to extend to multiple lines.
OBJS =  \
 Test1.o \
 Test2.o \
 Main.o

# Here is a Make Macro defined by two Macro Expansions.
# A Macro Expansion may be treated as a textual replacement of the Make Macro.
# Macro Expansions are introduced with $ and enclosed in (parentheses).
REBUILDABLES = $(OBJS) $(LINK_TARGET)

# Here is a simple Rule (used for "cleaning" your build environment).
# It has a Target named "clean" (left of the colon ":" on the first line),
# no Dependencies (right of the colon),
# and two Commands (indented by tabs on the lines that follow).
# The space before the colon is not required but added here for clarity.
```


http://coffeeghost.net

Quick Python Script Explanation for Programmers

```
import os

def main():
    print 'Hello world!'

    print "This is Alice's greeting."
    print 'This is Bob\'s greeting.'

    foo(5, 10)

    print '=' * 10
    print 'Current working directory is ' + os.getcwd()

    counter = 0
    counter += 1

    food = ['apples', 'oranges', 'cats']

    for i in food:
        print 'I like to eat ' + i

    print 'Count to ten:'
    for i in range(10):
        print i

def foo(param1, secondParam):
    res = param1 + secondParam

    print '%s plus %s is equal to %s' % (param1, secondParam, res)

    if res < 50:
        print 'foo'

    elif (res >= 50) and ((param1 == 42) or (secondParam == 24)):
        print 'bar'

    else:
        print 'moo'

    return res # This is a one-line comment.
    '''A multi-
line string, but can also be a multi-line comment.'''

if __name__ == '__main__':
    main()
```

Load other code modules.

Module name. This refers to "os.py"

The name "main" is just a convention, not a requirement. See the very bottom of this script.

Newline automatically added to print statements. Also, there are no semicolons at the end of the line.

No curly braces! Instead, the block starts when you add 4 spaces of indentation. (or any other amount, but 4 spaces is the standard)

I prefer single-quotes, but either is fine. Either way, you don't have to escape the other kind of quote inside the string.

Function call.

String replication. Evaluates to '=========='

String concatenation.

Call a function in the os module.

Variables MUST be instantiated first.

Lists can contain different data types in the same list, including other lists.

One-line block. When the indentation goes back down, the block has ended.

For loop. "i" takes on each value in the list "food" in order.

The range() function returns a list like [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] Don't forget the colon at the end!

Function definition. Don't forget the colon at the end.

String interpolation works basically the same way as it does in C.

The comparison operators are the same as C.

End of indentation, not a } brace, signifies the end of the block.

Boolean operators are words, not && and ||.

Colons come after def, for, while, if, elif, and else statements.

Comments.

Multi-line strings don't affect block indentation, though, only the indentation at the START of the statement or expression.

We put a call to main() at the bottom so that each def statement is executed by the time we call main(). This script's __name__ variable has the value '__main__' only when the script is run, not imported. With this check, the main() function won't run if another script imports this script.

Describe the pattern represented by the following regular expression, give an example of string that can match it:
"^\(?[2-9][0-9]{2}\)?(|-|\.)[0-9]{3}(|-|\.)[0-9]{4}"
This regex matches 10-digit phone numbers, with three kinds of delimiter: none, -, .
The first three digits(zone number) can be either with () or not. A zone number cannot start with 0 or 1.
ex: 2345678999, (234)5678999, 234-567-8999

1) How would you know the differnces between the the working copy of bar.c and bar.h, an their respective versions in a commit that was made just before the last commit (penultimate commit)? You cannot use commit hashes to refer to commits.
Ans: git status
    git diff
Explain the steps of dynamic linking:
Dynamic linking defers much of the linking process until a program starts running. It provides a variety of benefits that are hard to get otherwise:
Dynamically linked shared libraries are easier to create than static linked shared libraries.
Dynamically linked shared libraries are easier to update than static linked shared libraries.
The semantics of dynamically linked shared libraries can be much closer to those of unshared libraries.
Dynamic linking permits a program to load and unload routines at runtine, a facility that can otherwise be very difficult to provide.
There are a few disadvantages, of course. The runtime performance costs of dynamic linking are substantial compared to those of static linking, since a large part of the linking process has to be redone every time a program runs. Every dynamically linked symbol used in a program has to be looked up in a symbol table and resolved. (Windows DLLs mitigate this cost somewhat, as we describe below.) Dynamic libraries are also larger than static libraries, since the dynamic ones have to include symbol tables.

--start a linker---
The linker then initializes a chain of symbol tables with pointers to the program's symbol table and the linker's own symbol table. Conceptually, the program file and all of the libraries loaded into a process share a single symbol table. But rather than build a merged symbol table at runtime, the linker keeps a linked list of the symbol tables in each file. each file contains a hash table to speed symbol lookup, with a set of hash headers and a hash chain for each header. The linker can search for a symbol quickly by computing the symbol's hash value once, then running through apprpriate hash chain in each of the symbol tables in the list.
--find libraries---
Once it's found the file containing the library, the dynamic linker opens the file, and reads the ELF header to find the program header which in turn points to the file's segments including the dynamic segment. The linker allocates space for the library's text and data segments and maps them in, along with zeroed pages for the bss. From the library's dynamic segment, it adds the library's symbol table to the chain of symbol tables, and if the library requires further libraries not already loaded, adds any new libraries to the list to be loaded.
When this process terminates, all of the libraries have been mapped in, and the loader has a logical global symbol table consisting of the union of all of the symbol tables of the program and the mapped library.
Lazy:
All calls within the program or library to a particular routine are adjusted when the program or library is built to be calls to the routine's entry in the PLT. The first time the program or library calls a routine, the PLT entry calls the runtime linker to resolve the actual address of the routine. After that, the PLT entry jumps directly to the actual address, so after the first call, the cost of using the PLT is a single extra indirect jump at a procedure call, and nothing at a return.

real 0m4.866s: elapsed time as read from a wall clock
user 0m0.001s: the CPU time used by your process
sys 0m0.021s: the CPU time used by the system on behalf of your process

```
--- lib/utimens.c~    2005-11-02 04:16:12.000000000 -0800
+++ lib/utimens.c    2010-01-26 12:20:13.292737899 -0800
@@ -71,7 +71,7 @@ struct utimbuf
   Return 0 on success, -1 (setting errno) on failure.  */
 int
-futimens (int fd ATTRIBUTE_UNUSED,
+coreutils_futimens (int fd ATTRIBUTE_UNUSED,
                char const *file, struct timespec const timespec[2])
 {
  /* There's currently no interface to set file timestamps with
@@ -160,5 +160,5 @@ futimens (int fd ATTRIBUTE_UNUSED,
 int
 utimens (char const *file, struct timespec const timespec[2])
 {
-  return futimens (-1, file, timespec);
+  return coreutils_futimens (-1, file, timespec);      }
```