Appendix A1- Makefiles

Copyright ©2015 by Linda Marshall and Vreda Pieterse. All rights reserved.

Contents

A1.1 Introduction

Makefiles are data files that can be used by the make program to intelligently compile and link a system consisting of multiple C++ source files. It is the responsibility of the designer of the system to specify the detail needed by the make program to successfully compile and link the system. In order to be able to write your own makefiles, you need to understand how systems are compiled and linked without the aid of the make program. This is discussed in Section A1.2. In Section A1.3 we discuss how the make program automates this process. The content and structure of makefile entries are explained. In Section A1.4 we discuss how macros and rules can be used to write generic makefile entries. Understanding this enables you to write smaller and versatile makefiles that can easily be adapted to describe other systems. Finally we conclude with two small sections to mention common errors that are made by novices when creating there own makefiles and to introduce a few challenges to students who like to deepen their understanding through practical experience.

A1.2 Command line compiling and linking

A1.2.1 Compiling of a single C++ file

We assume that you use a text editor such as SciTE to write C++ programs and use the console terminal program of a linux operating system to type command-line instructions to compile and run these programs. When writing a C++ program, the source code has to be compiled before it can be executed. When the whole system is included in a single .cpp file, the process is trivial. You simply have to invoke the GCC compiler with the g++ command and specify the .cpp file name as input file. For example if your source code file is named **HelloWorld.cpp**, you can compile the program with the following command:

```
g++ HelloWorld.cpp
```

This command will create the executable file with the default name (a.out). This program can then be executed with the following command:

```
./a.out
```

Although only a single command is issued to perform this task, the following three steps are automatically executed to obtain the final executable program:

1. Compiler stage: All C++ language code in the .cpp file is converted into a lower-level language called Assembly language;
2. Assembler stage: The assembly language code made by the previous stage is then converted into object code which are fragments of code which the computer understands directly. An object code file is normally stored with .o as its file extension.
3. Linker stage: The final stage in compiling a program involves linking the object code to code libraries which contain certain "built-in" functions. This stage produces an executable program, which is named a.out by default.

It is possible to specify the output file name by using the **-o** option followed by the name of the output file. For example if your source code file is named **HelloWorld.cpp**, you can compile the program with the following command:

```
g++ HelloWorld.cpp -o HelloWorld
```

This command will create the executable file with the specified name **HelloWorld**. This program can then be executed with the following command:

```
./HelloWorld
```

Note that you can call the executable whatever you want. It need not have the same name as the .cpp file.

A1.2.2 Header files

When writing code in terms of classes (using the object oriented programming paradigm), it is customary to save the code of a given class in two files:

1. Header file: This file commonly contains forward declarations (prototypes / signatures) of classes, member functions, instance variables, and other identifiers. A header file is normally saved with .h as its file extension.

2. Implementation : This file contains the complete definitions of the member functions that are listed in the header file. The implementation file is normally saved with .cpp as its file extension.

When the code for such class needs to be compiled, it has to be made into a single document to be compiled by the compiler. This is done by using a compiler directive to insert the header file in the .cpp file before compiling the combined content of the two files. This is specified by using the a compiler **#include** command at an appropriate position in the implementation file. For example if a header file is named **Account.h**, it can be included in any .cpp file that make use of the classes, functions or variables declared in it by simply adding the following line of code in the .cpp file:

```
#include "Account.h"
```

When a .cpp file is compiled, the code in all the the header files inlcuded in it is also compiled. For example if a file named **Account.cpp** includes the file **Account.h** with the above compiler directive, the constructed file that is compiled by the following command consists of the code contained in both these files:

```
g++ Account.cpp -o Account
```

Multiple header files can be included in a single .cpp file. A single header file may also be included in multiple .cpp files.

To avoid multiple declarations when a given header file is included in multiple .cpp files in the same compilation, it is customary to guard the content of header files as a defined block (with a unique name). This guard compiles the inclusion only on condition that the defined block was not perviously encountered during the current compilation. The following illustrates how the content of a header file can be guarded:

```
#ifndef H_GUARD
#define H_GUARD

/* The forward declarations of classes, functions,
 variables, and other identifiers comes here */

#endif
```

The content of the header file is given a unique name with the command `#define H_GUARD`. To simplify naming, this name is usually chosen to correlate with the header file name.

The content of the file is placed in a conditional block starting with `#ifndef H_GUARD` and ending with `#endif`. This conditional statement specifies that the body of this statement should only be considered if the given name is not defined. Because the specified name is defined inside this block, the content of the block will be compiled only the first time it is encountered during the compilation process.

A1.2.3 Compiling multiple source files

When your program becomes very large, it makes sense to divide your source code into separate easily-manageable .cpp files. It can be compiled issuing a single command. For example, if a system is made up of two .cpp files named **Bike.cpp** and **Tricycle.cpp** respectively and a single common .h file named **Wheel.h**. The command to compile all, assuming **Wheel.h** is properly included in both .cpp files, is as follows:

```
g++ Bike.cpp Tricycle.cpp
```

Note that the first two steps taken in compiling the files are identical to the previous procedure for a single .cpp file. However, the two compiled files are linked together at the Linker stage to create one executable program. Because the name of the output file was not specified, the output file will be named **a.out** in this case.

A1.2.4 Compiling without linking

The steps taken in creating the executable program can be divided into steps, separating the compiling process from the linking process. Firstly all .cpp files can be compiled and stored as .o files and in a final step the .o files can be linked to create the executable program.

You can use the -c option with g++ to create the corresponding object (.o) file from a .cpp file. For example, the command to compile both the above mentioned .cpp files, without linking them is the following:

```
g++ -c Bike.cpp Tricycle.cpp
```

When executing this command, the compiler will stop after the assembler stage. The object code is written to disk in two .o files corresponding with the .cpp files.

A1.2.5 Linking compiled code

The compiler can use either .cpp or .o files when issued (*without* the -c option) to create the required executable file. The following command can thus be issued to link the compiled code that was created in the previous section:

```
g++ Bike.o Tricycle.o
```

If you would like to name the executable file something else than **a.out**, you can specify it with the -o option. For example if you would like to name the executable file of the above mentioned system **GoRide**, you can issue the following command:

```
g++ Bike.o Tricycle.o -o GoRide
```

Suppose you have written a system consisting of the following files

C++ file	**Includes**
Bike.cpp	Wheel.h, Bike.h
Tricycle.cpp	Wheel.h, Tricycle.h
main.cpp	Bike.h, Tricycle.h

When issuing the following commands, the intermediate .o files for this system will be created on disk, wherafter they are linked and a single executable file called **GoRide**[1] is created.

```
g++ -c Bike.cpp Tricycle.cpp main.cpp
g++ Bike.o Tricycle.o main.o -o GoRide
```

A1.2.6 Re-compiling after a small change

If you have changed one of the source files in the above mentioned system it is not neccesary to re-compile all. Only the files that are changed as a result of changing a source file need to be recompiled. For example, suppose only the Bike.cpp was changed. Knowing how the system is designed, you will realise that **Tricycle.o** and **main.o** are not affected by this change. A complete re-build of **GoRide** incorporating the change can thus be achieved by issuing the following commands:

```
g++ -c Bike.cpp
g++ Bike.o Tricycle.o main.o -o GoRide
```

Similarly if you have changed **Tricycle.h** the re-build would require the following commands:

```
g++ -c Tricycle.cpp main.cpp
g++ Bike.o Tricycle.o main.o -o GoRide
```

We have to admit that in this small system it would not make a big difference if you omit the creation of one or two .o files. However, when developing very large systems containing scores, or even hundreds, of files, it often saves a lot of time if only the appropriate files are updated.

[1]Note that the names given to files are arbitrary and should be chosen to be descriptive of your system

A1.2.7 Compiler flags

When compiling you can issue the compile command with options. One of the options (`-o`) was used in the example in Section A1.2.1 to specify the output file. A host of other options, known as compiler flags, are available for most compilers. However, they are not standardised for all compilers. Table 1 lists some useful flags that are available for the GCC compiler that is used on campus.

Table 1: Useful g++ flags

Flag	**Usage**
-o	To specify the output filename
-w	Disable all warning messages. Note that the the use of -w to suppress the compiler warnings is against our coding standards, because for better reliability one should take compiler warnings serious, and not ignore them as if they don't exist!
-Wall	Enable most compiler warnings
-Werror	Treat compiler warnings as errors Note that the use of this flag really show that you are serious about compiler warnings because you actually want to turn them into errors.
-pedantic	Issue all the warnings demanded by ISO C++; reject all programs that use language extensions supported by the GCC compiler that is not part of the ISO specification of the language.
-pedantic-errors	Like `-pedantic`, except that errors are produced rather than warnings.
-g	turns on debugging. This makes your code ready to run under gdb.
-O	turns on optimization, you may also specify levels (-O2).
-E	outputs the preproccessor output to the screen (stdout).
-static	On systems that support dynamic linking, this prevents dynamic linking with the shared libraries
-c	compiles the given source down to an object file. This is used projects that have multiple files to reduce compile time.
-MM	outputs the Makefile dependancies for the source file(s) listed.

Any number of these flags can be inserted in the compile command. For example, the following command will compile the `HelloWorld.cpp` program that was mentioned in Section A1.2.1 to create an executable file called `MyWorld`. Furthermore, it will treat all warnings as errors and also include additional debugging information in this executable.

```
g++ -Werror -g HelloWorld.cpp -o MyWorld
```

A1.2.8 Wild card characters

When specifying file names (or paths), the asterisk character ***** can be used as a substitute for zero or more unspecified characters. Similarly the question mark **?** can be used as a substitute for one unspecified character. Ranges of characters enclosed in square brackets ([and]) serve as a substitute for any of the characters in the specified ranges; for example, [A-Za-z] substitutes any single capitalized or lowercase letter. The following command is an example that shows how these substitues can be used to write more generic commands. This command compiles all the .cpp files in the current directory that has a lower or upercase **k** as the third character in the filename:

```
g++ ??[kK]*.cpp
```

By using naming conventions, for example starting a specific subset of filenames with a common character, can enable you to refer to these files in terms of a wild card pattern.

A1.3 Automating the build process

A program called **make** can be used to build source codes automatically. Apart from automating the build process, this program was designed to only build source code that has been changed since the last build. The make program gets the inter-dependency of the source files in the system from a text file called **Makefile** or **makefile**. If no path is specified it is assumed that this file resides in the same directory as the source files.

Make checks the modification times of the files, and whenever a file becomes *newer* than something that depends on it, it runs the build script accordingly.

A1.3.1 Entries in a makefile

Each entry in the Makefile uses the following format:

target: source file(s)
→ *command*
...

A target given in such instruction is a file which will be created or updated when any of its source files are modified. The list of source files is also called the dependency list of the entry. The dependency list is a list of file names separated by spaces. One or more commands can be listed. The command(s) given in the subsequent line(s) are executed in order to create the target file. The → in the above format represent a **tab character**. **Each command must be preceded by** →. The → is used by **make** to distinguish between commands and dependency rules.

The **make** program uses the entries in the makefile to determine which command(s) should be issued to update the system. The commands that are executed are determined by the files that have changed since the last build. For example, if **Bike.cpp**, in the example described in Section A1.2.5, is changed it becomes newer than **Bike.o** that depends on it. The make program must then issue a command to create a new **Bike.o**.

The information needed by **make** to know when **Bike.o** needs to be updated, and what command should be executed to update **Bike.o** is listed as follows:

```
Bike.o:  Bike.cpp Bike.h Wheel.h
     g++ -c Bike.cpp
```

The above entry states that whenever **Bike.o** is *older* than **Bike.cpp, Bike.h** or **Wheel.h**, the command `g++ -c Bike.cpp` should be issued.

As a result of execution of the above mentioned command, **Bike.o** then becomes *newer* than **GoRide**, that in turn depends on **Bike.o**. Once again the **make** program must issue a command to create a new version of **GoRide**. The files on which **GoRide** depend, and the command to be issued when any of these files are updated is listed in the makefile as follows:

```
GoRide:  Bike.o Tricycle.o main.o
     g++ Bike.o Tricycle.o main.o -o GoRide
```

The following is a complete listing of an example makefile for the above mentioned system:

```
GoRide: Bike.o Tricycle.o main.o
     g++ Bike.o Tricycle.o main.o -o GoRide

Bike.o: Bike.cpp Bike.h Wheel.h
     g++ -c Bike.cpp

Tricycle.o: Tricycle.cpp Tricycle.h Wheel.h
     g++ -c Tricyle.cpp

main.o: main.cpp Bike.h Tricycle.h
     g++ -c main.cpp
```

When the make program reads this data, it will attempt to create only one specified target at a time. If one or more of the items in the dependency list of the current target is missing or outdated, it will find the instructions to build them in subsequent entries and execute them in the order they are mentioned in the dependency list. It is important that the rule to create any item in any given dependency list is listed after all its occurrences in dependency lists.

A1.3.2 Using the makefile with make

Once you have created your makefile and your corresponding source files, you are ready to use make. If you have named your makefile either Makefile or makefile, make will recognize it and use it if you issue the following command at the command prompt:

```
make
```

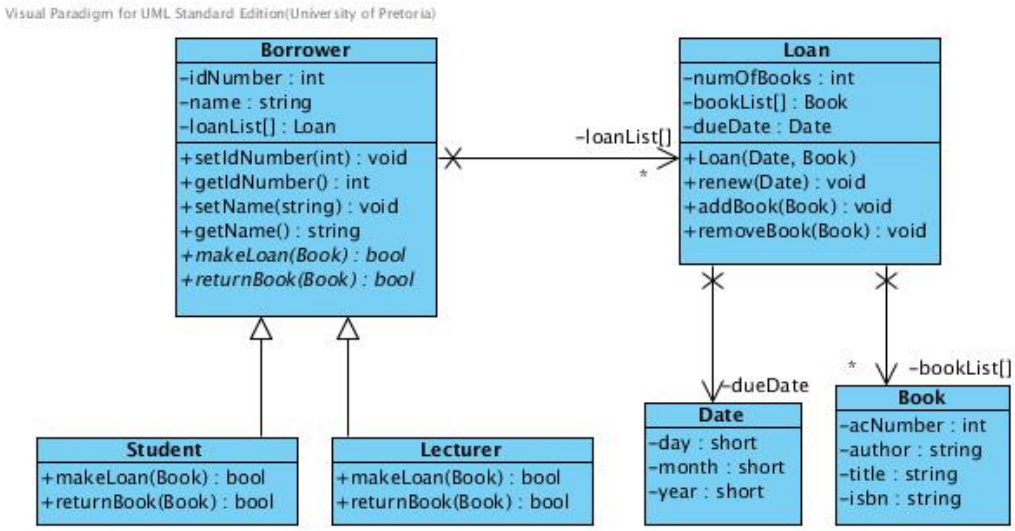

Figure 1: Borrower-Loan Class diagram

The default entry point is the top entry in the makefile. If the target of the top entry in your makefile is the final executable you wish to create you do not have to specify any parameters when issuing the make command. You can also use the make program to create a specific target by passing the required target as a command line parameter to the make program as the entry point. For example if you would like to create **Tricycle.o** using the above makefile with make you can type the following command:

```
make Tricycle.o
```

If you do not wish to call your makefile one of these names, you can give it any name you desire. If you do not use the default name, you have to specify the name of the makefile by using the -f option when invoking the make program. For example if you named your makefile **MyMakeData** you can specify that the make program must use this file with the following command:

```
make -f MyMakeData
```

A1.3.3 Dependency lists

To create the .o files of a system that contains a large number of files, it is important to include all the files that each .cpp depends on, in its dependancy list. A particular .cpp file depends on its own .h file as well as all the .h files that are directly or indirectly included in the .cpp file. For example if you are given the system that contains the classes shown in the UML class diagram of figure 1, and the C++ definition of each of the classes in the diagram is stored in an .h file with the same name as the class, then the `makefile` entry to create `Student.o` should be:

```
Student.o: Student.cpp Student.h Borrower.h Book.h Loan.h Date.h
        g++ -c Student.cpp
```

It does not make sense to use wild card characters in dependency lists. It will defeat the purpose of the dependency list.

A1.3.4 Continuation Line

Sometimes entries in a makefile tend to become very wide and consequently difficult to read and maintain. You are advised to break long lines into more lines. A backslash ("\") at the end of a line indicates that the next line should be interepreted as the continuation of the current line. For example is the following entry is equivalent to the one above.

```
Student.o: Student.cpp \
           Student.h  \
           Borrower.h \
           Book.h \
           Loan.h \
           Date.h
    g++ -c Student.cpp
```

It is important that the backslash that indicates that the current line continues on the next line is followed immediately by a newline. The continuation line may start at the margin, or may be indented (even with tabs). It is more readable if you use indentation on continuation lines.

A1.3.5 Linking order

When having to link a number of .o files, it is important to list them in the correct order for the link command. They are always created in the order of which they are listed. For example if fileA.cpp directly or indirectly includes fileB.h, then fileB.o should appear before fileA.o in the list of .o files to link.

For example, if we assume that a C++ file containing the `main` function, called `LibSys.cpp`, use the classes shown in the UML class diagram of Figure 1, the following makefile entry will link the .o files to create an executable file called `LibSys`:

```
LibSys: Book.o Date.o Loan.o Borrower.o Student.o Lecturer.o
        g++ Book.o Date.o Loan.o Borrower.o \
                Student.o Lecturer.o LibSys.cpp -o LibSys
```

Note that `Book.o` and `Date.o` may appear in any order because thy are independent of one another. However, both of them have to be listed before Loan.o because Loan.o depends on them. Likewise, Loan.o should be before Borrower.o, which should in turn be before Student.o and Lecturer.o. The last two objects are independent of one another and may therefore be listed in any order.

Because the order in which files are named in a link command is signifiant, using wild card characters in a linking instruction may lead to unwanted results.

A1.3.6 Adding custom commands

You can define any other command-line commands that you frequently use in your makefile. Do this by specifying it as a target without any dependencies and listing the command(s) that you would like to execute when you invoke the custum command below it, preceded by the usual tab charactes.

The following shows the format for such custom command:

target:
→ *command*
→ *command*
...

One of the most common custom commands that are found in makefiles is the **clean** command. This command is used to issue a command to delete all interim files and other redundant files the current directory. This is sometimes useful to force a full compilation with the next issue of a make command, for example when the system needs to be recompiled on a different operating system.

On linux a file is deleted (removed) with the **rm** command. The following is a typical definition of a custom command named **clean**:

```
clean:
     rm -f *.o *~
```

This command uses * as a *wild card* character to specify that all files with .o extension should be deleted. In the same way `*~` indicates that all files extension ending with `~` are also included in the list of files to be deleted. These are typical automatic backup files. The -f option supresses the error message that might be generated, for example when the command is issued while no such file exists.

Custom commands are executed by passing the required command as a command line parameter to the make program. For example if you would like to execute the above **clean** command, you can type the following command:

```
make clean
```

Sometimes when you recompile after a change you may see the message that states that your target is up to date, while you know it is not. In such case you can force a full compilation by issuing the clean command. If you often find that you need to use this "clean" option, it is likely that some of your dependancies are not included. When not porting to a differnt operating the use of "clean" defeats the purpose of the makefile. If all the file dependencies are correct in the makefile, you should never need the "clean" unless you port to another operating system.

Custom commands are powerful tools. It not only enables you to code some frequently used actions like removing backups or making tarballs to your makefile, it also enables you to add additional steps to the compilation process. For example if you have an application called **dingamaging** that takes a **.dmg** file to generates a **.cpp** file, the step to generate the .cpp file can also be included in the makefile.

Back to the Bicycle example. Suppose you were using this fictitious application to develop the code. The source code will then be in two files called `Bike.dmg` `Tricycle.dmg` respectively. You will need to apply **dingamaging** to these files to create `Bike.cpp` and `Tricycle.cpp`. The following is a complete listing of an example makefile for this system if it were developed with the help of a third party application called **dingamaging**:

```
GoRide: Bike.o Tricycle.o main.o
  g++ Bike.o Tricycle.o main.o -o GoRide

Bike.o: Bike.cpp Bike.h Wheel.h
  g++ -c Bike.cpp

Tricycle.o: Tricycle.cpp Tricycle.h Wheel.h
  g++ -c Tricyle.cpp

main.o: main.cpp Bike.h Tricycle.h
  g++ -c main.cpp

Bike.cpp: Bike.dmg
  dingamaging Bike.dmg

Tricycle.cpp: Tricycle.dmg
  dingamaging Tricyle.dmg
```

A1.3.7 Comments

As with any source, the 'source' of the makefile can, and should, also be commented to explain it to possible readers. Any text preceded by the # character is ignored by the make program. Therefore, you can include complete comment lines by starting a line with #, or comments to the right of a statement in the makefile. The following listing of our example makefile include some comments:

```
# Linking the object code of the complete system:
GoRide: Bike.o Tricycle.o main.o
  g++ Bike.o Tricycle.o main.o -o GoRide

# Commands for partial compilation of c++ source files:
Bike.o: Bike.cpp Bike.h Wheel.h
  g++ -c Bike.cpp

Tricycle.o: Tricycle.cpp Tricycle.h Wheel.h
  g++ -c Tricyle.cpp

main.o: main.cpp Bike.h Tricycle.h
  g++ -c main.cpp

# Custom command:
clean:
  rm -f GoRide *.o *~ # deleting executable, .o's and backups
```

A1.4 Reducing the size of a makefile

A1.4.1 Introduction

The make program has many features. Two of the powerful features offered are the ability to define macros and the ability to specify generic rules by using some built-in macro's. These features combined with the use of wild card charactres (see Section A1.2.8) enables the programmer to write more concise makefiles.

A1.4.2 Macros

The make program allows you to use macros, which are similar to variables, to store names of files. For example you can define a name for the list of object files as follows:

```
OBJECTS = Bike.o Tricycle.o main.o
```

The make program automatically expand a macro when it runs. Whenever the macro name appears within round brackets and preceded by a dollar sign, it will be replaced by its defined content. The following is a listing of our sample Makefile again, using the above mentioned macro.

```
OBJECTS = Bike.o Tricycle.o main.o

# Linking the object code of the complete system:
GoRide: $(OBJECTS)
  g++ $(OBJECTS) -o GoRide

# Commands for partial compilation of c++ source files:
Bike.o: Bike.cpp Bike.h Wheel.h
  g++ -c Bike.cpp

Tricycle.o: Tricycle.cpp Tricycle.h Wheel.h
  g++ -c Tricyle.cpp

main.o: main.cpp Bike.h Tricycle.h
  g++ -c main.cpp
```

Note that the name that is given to a macro is chosen, like any other variable name, by the programmer. You may name it whatever you like. However, when selecting macro names you should apply the same rules as you would for variables in your programs. Most importantly they should be descriptive of what they represent. Cutomary to our coding standards, we treat these macros similar to named constants in C++ programs by using ALL_CAPS when defining them.

A1.4.3 Special macros

In addition to those macros which you can create yourself, there are a few built-in macros which are used internally by the make program. Some of them are listed below:

`CC`	Contains the current compiler. Defaults to cc
`CFLAGS`	Special options which are added to the built-in compile rule
`$@`	Full name of the current target.
`$?`	A list of files for current dependency which are out-of-date.
`$<`	The source file of the current (single) dependency.

A1.4.4 Rules

The real power of makefiles is seen when rules are used. A new `%` wild card character is introduced. The meaning of `%` is similar to `*`. However, it has different behaviour when expanding. Instead of putting all files that match the pattern in place in the single command, a separate command is created for each instance.

The following is an example of a rule that specifies that any .o file in the current folder depends on its corresponding .cpp file and can be created by compiling the .cpp file with the -c flag:

```
%.o:  %.cpp
      g++ $< -Wall -c -o $@
```

This compiling is specified to be done with warnings enabled. `$<` expands to the first dependency (a .cpp source file). `$@` expands to the target (the corresponding .o file). This single rule can be used instead of a specific rule for each of the object files in the system.

Rules need not be for compiling only. For example, if we have a program called **dingamaging** that takes a **.dmg** file and automatically generates a **.cpp** file, we can automate the generation of .cpp files from .dmg files by adding the following rule to the makefile:

```
%.cpp:  %.dmg
      dingamaging $< -o $@
```

If we have added the above mentioned rule to our makefile and have a file named Bike.dmg in the current folder that is newer than the Bike.cpp file in the folder. The make program will generate a new Bike.cpp file, which in turn will trigger the recompilation of subsequent files depending on Bike.cpp.

In our final version of our example makefile below, we have expanded our use of macros. We redefine some of the pre-defined macros, for example CC and CFLAGS to contain detail specific to our needs. We also define a different flag set to be used when linking with a macro called LFLAGS. This way we exclude the flags that do not make sense when compiling from the CFLAGS list. We also include two custom commands that are independent of any other files (they have no dependancy lists). These are respectively to remove redundant files and to execute the program.

```
CC = g++
CFLAGS = -Wall -Werror
LFLAGS = -static
TARGET = GoRide
OBJECTS = Bike.o Tricycle.o main.o

# Linking all the object code:
all: $(OBJECTS)
  $(CC) $(LFLAGS) $(OBJECTS) -o $(TARGET)

# Dependencies:
Bike.o: Bike.h Wheel.h
Tricycle.o: Tricycle.h Wheel.h
main.o: Bike.h Tricycle.h

# Compilation rule:
%.o:  %.cpp
  $(CC) $< $(CFLAGS) -c -o $@

# Custom commands:
clean:
  rm -f $(TARGET) $(OBJS) *~ # deleting executable, .o's and backups

run:
  ./$(TARGET) # executing the program
```

The detail needed in the dependency list was reduced by the presence of the compilation rule. Because the rule specify that each .o file is dependant on its corresponding .cpp file, we no longer need to specify the .cpp files in the list of dependancies. We only need to list all the .h files that is included in the .cpp file to ensure proper partial recompilation if one of the .h files are changed. The rule also specifies how the .o files can be created. Therefore, this detail is not needed where the dependencies are listed. The make program will know *when* to create a specific .o file through the dependency list without the command, and will know *how* to create it through the specified compilation rule.

This generic makefile can now easily be modified to be applicibale to a specific system. One only have to update the macros defined at the beginning of the makefile and ofcourse the dependencies.

A1.5 Common errors in makefiles

A1.5.1 Syntax

- Failing to put a TAB at the beginning of commands. This causes the commands not to run.
- To put a TAB at the beginning of a blank line. This causes the make utility to complain that there is a "blank" command.

- Not hitting return just after the backslash, should you choose to use it. If the character directly before a newline is not the backslash the text on the next line is not interpreted as being a continuation of the previous line. It would be the same as not having the continuation character at all.

A1.5.2 Dependencies

- Not including all dependencies.

  To create an .o file by compiling the corresponding .cpp file, the .o file is dependent on the .cpp file **as well as** all the header files that are either directly or indirectly included in the .cpp file.

A1.5.3 Order of linking

- Listing the files in a link command in a wrong order.

  If Aaa.cpp directly or indirectly includes Bbb.h, then Bbb.o should appear before Aaa.o in a compile or link command that includes these .o files in its source file list.

A1.6 Challenges

The following are practical assignments you can do to convince yourself that you understand the work. These are listed in order of increasing difficulty.

1. Write a custom rule to `tar` the .cpp, .h files and the Makefile in the current folder.
2. Write a custom command that will print all .cpp files that have changed since the last build.
3. Write a makefile that use a rule to generate the dependency list of the .cpp files in the current folder and include it automatically in the makefile[2]

A1.7 Further Reading

`http://www.cs.duke.edu/courses/cps108/doc/makefileinfo/html/Makefiles.html#toc1`

[2]You will have to find out about including files in a makefile