Department of Computer Science
UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

COS121 Lecture Notes: L11

Compile and make

22 August 2014

Contents

L11.1 Introduction

Computer science is not just about building computers or writing computer programs. It covers a wide range of topics which you will encounter while studying computer science. During your first year of study at UP, you will use the C++ programming language to experiment with practical applications within these topics. This entails writing, compiling and running programs written in C++. In the formal curriculum considerable time is devoted to teaching how to write programs. Writing programs is both an art and a skill. Here we focus on some of the scaffolding you need to exercise your programming skills. To become a successful programmer it is extremely important that you compile and run as many programs as possible. Do not accept what you are told in the textbook without questioning. Type the examples and compile and run them. Experiment with them by making small changes and observe if those changes have the impact that you expected. In this chapter we discuss the practical tricks and tools you need to compile programs. You need to master these tools to be able to experiment with the examples you have to study. You will also have to apply these tools when writing your own programs.

The `make` program was developed to automate some aspects of the compiling process. This chapter covers the aspects of `make` that are needed in your first year. It also paves the way to learning the advanced features of `make` you might need in your further studies.

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 L11.2. In Section L11.3 we discuss how the `make` program automates this process. The content and structure of makefile entries are explained. In Section L11.4 we discuss how macros and rules can be used to write generic makefile entries. Understanding this enables you to write compact versatile makefiles that can easily be adapted to describe other systems. Section L11.5.1 cites common errors that are made by novices when creating their own makefiles, while Section L11.5.2 provides a summary that can be used as reference when you have to apply this skill and need to refresh your memory on how to do it. Finally we conclude with Section L11.5.3 to introduce a few challenges to students who like to deepen their understanding through practical experience. The section also provides guidelines on how to further develop your compiling skills and find out more about how to put the `make` program into practical use for a variety of applications.

L11.2 Command line compiling and linking

L11.2.1 Compiling 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 instructions to compile and run programs. When writing a C++ program, the source code has to be compiled before it can be executed. When the code of 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.

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 the compilation of the program, 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. If your source code file is named **HelloWorld.cpp**, you can compile the program with the following command:

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

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

```
./HelloWorld.out
```

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

L11.2.2 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 a class needs to be compiled, it has to be combined into a single document before it can be compiled. 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 `#include` command at an appropriate position in the implementation file.

If a header file is named **Account.h**, it can be included in any .cpp file that uses the classes, functions or variables declared in the header file 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 header files inlcuded in it is also compiled. 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.out
```

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 previously 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 entire content of the header 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.

L11.2.3 Compiling multiple source files

When a system becomes larger, it makes sense to divide the source code into separate easily-manageable .cpp files. Even if a system consists of many files, it can be compiled issuing a single compile command.

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**, then 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.

L11.2.4 Compiling without linking

The single command that performs the three stages mentioned in Section L11.2.1 to create the executable program, can be separated into two commands. The one command is is issued to complete the compiling and assembly stages without performing the linking stage. The second command is issued to complete the linking stage. The first command should compile and assemble all .cpp files and store its result (the object code) in .o files. The second command should link the object code in these .o files to produce the executable program.

You can use the -c option with g++ to create the corresponding object (.o) file from a .cpp file. 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.

L11.2.5 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. If you would like to name the executable file of the above-mentioned system **GoRide.out**, you can issue the following command:

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

Assume 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, whereafter they are linked and a single executable file called **GoRide.out**[1] is created.

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

L11.2.6 Recompiling after a small change

If you have changed one of the source files in the above-mentioned system it is not neccesary to recompile all. Only the files that are changed as a result of changing a source file need to be recompiled. Considering how the system is designed, you will realise that **Tricycle.o** and **main.o** are not affected by changes in Bike.cpp.

If only Bike.cpp is changed, a complete rebuild of **GoRide.out** 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.out
```

Similarly, if only **Tricycle.h** was changed, the rebuild would require the following commands:

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

Here we have shown how we can save compile time in situations where a complete rebuild is not needed.

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.

You may realise that this saving of compile time came at the cost of the application of extra work done by the programmer. The programmer had to issue more commands. These commands also required some thinking to get them right. In Section L11.3 you will see how this compile time can be saved without this cost when this work is automated.

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

L11.2.7 Compiler flags

When compiling you can issue the compile command with options. A host of options, known as compiler flags, are available for most compilers. They are, however, not standardised for all compilers. Table 1 lists some useful flags that are available for the GCC compiler we use.

Table 1: Useful g++ flags

Flag	Usage
-o	To specify the output filename
-w	Disable all warning messages. Note that use of -w to suppress the compiler warnings is against our coding standards, because for better reliability one should take compiler warnings seriously, 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 shows 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 optimisation. You may also specify the level of optimisation required, for example -O1 or -O2.
-E	Displays the pre-processor output on 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 to reduce the compile time of projects that consist of multiple files.
-MM	Displays the makefile dependencies for the given source file(s) on the screen.

Any number of these flags can be inserted in the compile command. The following command will compile the `HelloWorld.cpp` program that was mentioned in Section L11.2.1 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.out
```

L11.2.8 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 capitalised or lowercase letter.

The following command is an example that shows how these substitutes can be used to write more generic commands. This command compiles all the .cpp files in the current directory that has a lower or uppercase **k** as the third character in the file name:

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

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

L11.3 Automating the build process

A program called **make** can be used to build systems automatically from given source code. Apart from automating the build process, this program was designed to only rebuild parts of the system of which the source code 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 usually called `Makefile` or `makefile`. If no path is specified, it is assumed that this file resides in the same directory as the source files.

The `make` program 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.

L11.3.1 Entries in a makefile

Each entry in the makefile uses the following format:

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

The word 'target' in this template is the filename of the file which will be created or updated when any of its source files are modified. The list of source files, represented by the words 'source files(s)' in the template, 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 may be listed. The command(s) given in the subsequent line(s) are executed in the given order. It is assumed that their execution will create the target file. The → in the above template represent a **tab character**. **Each command must be preceded by** →. The → is used by `make` to distinguish between commands and dependency rules.

L11.3.2 A makefile entry to compile and link a single file

When the code of a system is located in a single .cpp, technically there is no reason to *automate* the process with `make` and have a makefile to compile it. It can be compiled using a command line instruction as described in Section L11.2.1. It is, however, advisable to start using `make` as early as possible in your C++ programming career. You will have to start using it sooner or later and will find it easier to use it when it really becomes necessary if it has been part of the process all along.

A single file can be compiled using a single command line instruction. Consequently a makefile with single makefile entry is needed to compile the file when using `make`. In this case each of the three parts of the entry as described in Section L11.3.1 is trivial:

- The *target* is the name of the required executable that has to be created.
- The *source file* is the name of the file containing the source code.
- The *command* is the command line instruction needed to compile the program.

The makefile to compile the program in the example in Section L11.2.1 — while enabling most compiler warnings and preventing dynamic linking — consists of the following single entry:

```
MyFirstProgram: HelloWorld.cpp
→     g++ -Wall -static HelloWorld.cpp -o MyFirstProgram.out
```

L11.3.3 Makefile entries for selective compilation

If a system consists of multiple files, the `make` program should be applied. The input data to the `make` program (the makefile) serves a dual purpose. In the first place it specifies all the commands that need to be executed to build the system and ultimately create its executable as well as the order in which they should be executed. Secondly, it provides the information needed to apply selective compilation as described in Section L11.2.6.

When executed, 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. If **Bike.cpp**, in the example described in Section L11.2.5, 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 about when **Bike.o** needs to be updated, and what command should be executed to update **Bike.o**, are contained in the following makefile entry:

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

Similarly, the files on which **GoRide.out** depend, and the command to be issued when any of these files are updated are contained in the following makefile entry:

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

The first of the above entries 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 this command, **Bike.o** will become *newer* than all other files in the system. This will trigger `make` to update all targets that are dependent on `Bike.o` according to the entries in the makefile. Consequently the command of the second of the two above-mentioned makefile entries will be executed to create an updated version of `GoRide.out`.

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

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

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 order in which the commands are executed (if executed at all) is **not** determined by the order in which they are listed in the makefile. Usually the `make` program is invoked to create a single specified target, and the order of the dependency list of that target determines the order in which the required commands are executed. When the `make` program is invoked with the above data to create `GoRide.out`, it will start by inspecting the dependency list of `GoRide.out`. If one or more of the items in this dependency list of the current target are 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.

L11.3.4 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 recognise it and use it if you issue the following command at the command prompt:

```
make
```

It is actually required that you specify an entry point in the makefile which is the name of the target you wish to create. The default entry point is the top entry in the makefile. Thus you can avoid having to specify the target by making the instructions to create the final executable the top entry. You can use the `make` program to create any specific target by passing the required target as a command line parameter to the `make` program as the entry point. Type the following command to create **Tricycle.o** using the above-mentioned makefile:

```
make Tricycle.o
```

You are not required to give your makefile any specific name. You may 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. If you named your makefile **MyMakeData** you can specify that the `make` program must use this file with the following command:

```
make -f MyMakeData
```

L11.3.5 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 dependency 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. 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 Loan.h Date.h Book.h
        g++ -c Student.cpp
```

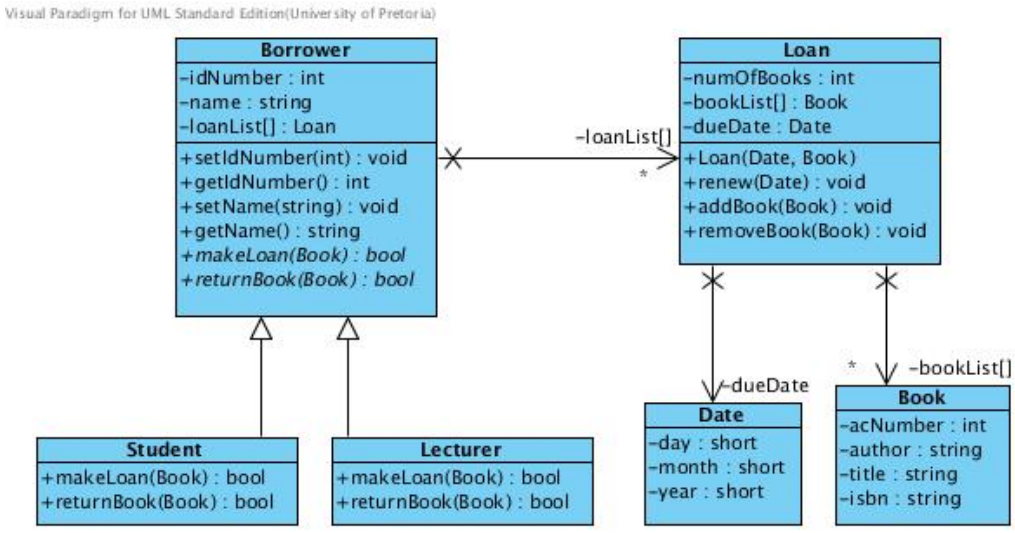

Figure 1: Borrower-Loan Class diagram

The dependency list of a given target is used to specify which files are needed to create the target. The dependency list is also used to specify the order in which the files should be considered. For this reason the order in which the files are listed in a dependency list is important.

It does not make sense to use wild card characters in dependency lists. The order is not determined if files are specified with wild card characters. Furthermore, the use of wild card characters is likely to defeat the purpose of the dependency list in terms of selective compilation because the wild cards might match files that need not be considered, resulting in some unnecessary compilation.

L11.3.6 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 interpreted as the continuation of the current line. The following entry is equivalent to the entry in Section L11.3.5.

```
Student.o: Student.cpp \
           Student.h \
           Borrower.h \
           Loan.h \
           Book.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 new line. 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.

L11.3.7 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. The link instruction will create them in the order of which they are listed. As a general rule, if `A` depends on `B`, then `B` should be listed before `A`. i.e. if `fileA.cpp` directly or indirectly includes `fileB.h`, then `fileB.o` should appear before `fileA.o` in the list of .o files in the link command.

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.out: 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.out
```

Note that `Book.o` and `Date.o` may appear in any order because thy are independent of any of the files of the system and also independent of one another. They both 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 as long as they are after all the other files.

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

L11.3.8 Adding custom commands

You can define your own commands that you frequently use in your makefile. Do this by specifying the command as a target without any dependencies. This is called a pseudo target. List the command(s) that you would like to execute when you invoke the custom command below the pseudo target. Precede each command by the usual tab character.

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 applied to delete all interim files and other redundant files from the current directory. It is sometimes useful to force a full compilation with the next issue of a `make` command. This may be needed when the system is recompiled on a different operating system or different version of the 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 with names ending with ~ are also included in the list of files that should be deleted. These are typical automatic backup files. The -f option supresses the error message that might be generated by the `rm` command, for example, when this 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. 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 a case you can force a full compilation by issuing the clean command. If you often find that you need to use this clean command, it is likely that some of your dependencies are not included. If all the file dependencies are correct in the makefile, it is unlikely that you need to use this command. You may only need it when you port to another operating system or other version of the operating system on which the system was previously compiled.

Custom commands are powerful tools. It not only enables you to code some frequently used actions like removing backups or making tarballs in your makefile, it also enables you to add additional steps to the compilation process. 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.out: Bike.o Tricycle.o main.o
  g++ Bike.o Tricycle.o main.o -o GoRide.out

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
```

L11.3.9 Comments

The following listing of our example makefile includes some comments:

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

# 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.out *.o *~ # deleting executable, .o's and backups
```

It should be clear that any text preceded by the # character is treated as comments and ignored by the `make` program. As shown in the last line of this example, code may appear on the same line as comments. As with any source, it is good programming practice to include comments in your makefile to explain the code to readers. You should, however, avoid redundancy. It is not a good idea to write comments about information that is already clear in the code.

L11.4 Reducing the size of a makefile

L11.4.1 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 macros. These features combined with the use of wild card characters (see Section L11.2.8) enable the programmer to write more concise makefiles.

L11.4.2 Macros

The `make` program allows you to use macros, which are similar to variables, to store string constants that can be substituted anywhere in the makefile. 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 is 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.out: $(OBJECTS)
  g++ $(OBJECTS) -o GoRide.out

# 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 should select macro names carefully. 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. Customary to our coding standards, we treat these macros similar to named constants in C++ programs by using ALL_CAPS when defining them.

L11.4.3 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 in the current dependency list, which are outdated.
`$<`	The source file of the current (single) dependency.

L11.4.4 Rules

The real power of makefiles is seen when rules are used. A new wild card character is introduced. It is the percentage sign (`%`). The meaning of `%` is similar to the meaning of the `*` wild card character, but not exactly. It behaves differently when expanding. When `*` is used in a template, all files that match the `*`-pattern are included in a single structure that adheres to the template. On the contrary, when `%` is used in a template, a separate structure that adheres to the template is created for each instance that matched the `%`-pattern.

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. 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, then 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. We also include two custom commands that are independent of any other files (they have no dependency lists). One is to remove redundant files and the other is to execute the program.

```
CC = g++
CFLAGS = -Wall -Werror
LFLAGS = -static
TARGET = GoRide.out
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 specifies that each .o file is dependant on its corresponding .cpp file, we no longer need to include the .cpp files in the list of dependencies. We only need to list all the .h files that are directly or indirectly included in each of the .cpp files. The compilation 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 applicable to a specific system. One only has to update the dependencies and macros defined at the beginning of the makefile.

L11.5 A final note

L11.5.1 Common errors

Syntax errors

- 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` program 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.

Errors in dependency lists

- 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.

Mistakes in ordering

- 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.
- Having an occurrence of an item after its creation rule.

  It is important that the rule to create any item in any given dependency list is listed after all its occurrences in dependency lists.

L11.5.2 Survival kit

Example makefile for a system consisting of a single cpp file

Type the following makefile instruction in a text editor and save it as a file called `makefile`

```
<executable-filename> : <cpp-filename>
→ g++ -static <cpp-filename> -o <executable-filename>
```

NOTE: The → is the tab character. Example:

```
Task3:  Task3.cpp
→g++ -static Task3.cpp -o Task3
```

Example makefile for a system consisting of multiple files

Create a makefile consisting of $n + 1$ makefile instructions where n is the number of cpp files in your system.

For each of the cpp files in your system include a compile instruction in the makefile to compile it to an object file. This instruction should comply with the following template. Note that the h-files that should be listed are all the h-files that are directly or indirectly included in the specific cpp file:

```
<object-file>:  <cpp-file> <h-file1> <h-file2>...<h-filen >
→g++ -c <cpp-file> -o <object-file>
```

Example:

```
InternalTools.o:  InternalTools.cpp InternalData.h InternalTools.h
→g++ -c InternalTools.cpp -o InternalTools.o
```

Include a link instruction in the makefile to link all the object files to an executable. This link instruction should be the first instruction in the makefile, i.e. it should be inserted at the top. This instruction should comply with the following template:

```
<executable-file>:  <o-file1> <o-file2>...<o-filen >
→g++ -static <o-file1> <o-file2>...<o-filen > -o <executable-file>
```

It is important to list the o-files in the link instruction in an order that takes the dependency of these files into account. If `FileA.cpp` is dependent on `FileX.h`, then `FileX.o` should be listed before (to the left of) `FileA.o` in the link instruction.

Example:

```
ZombieApocalypse:  InternalData.o InternalTools.o Main.o
→g++ -static InternalData.o InternalTools.o Main.o -o ZombieApocalypse
```

L11.5.3 Put compiling and makefiles in your pocket

The above survival summary will be enough to get you through most of your assignments. You are, however, strongly advised to work through this entire document and read more about the GNU compiler and makefiles in manuals and other resources that you can buy in bookshops or find on the internet. You can access the manuals of the `make` program and the GCC compiler by typing the following commands in the console[2];

```
man make
man gcc
```

[2]Press **Q** to quit the manual

When studying an example, you should verify that you understand the purpose and consequence of every command and every line of code. The best way to do this is to replicate the example in practice. Experiment with the example in Section L11.2.3. This can be done by creating files named **Bike.cpp** and **Tricycle.cpp** as well as an h-file file named **Wheel.h**. The h-file should contain the prototypes of a few functions while the cpp-files should contain the implementations of these functions. **Tricycle.cpp** should also contain a `main` function that calls the functions defined in the .h file. Now you can see for yourself how this system is compiled with the given command.

You can also deepen your understanding of how makefiles work by applying the knowledge to other examples and doing more exercises. 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 and use a custom rule to `tar` the .cpp, .h files and the makefile of your system.
2. Write and use a custom command that will print all .cpp files that have changed since the last build.
3. Write and use a makefile that updates a macro called `O_FILES` by applying a rule to a macro called `SRC_FILES`. This rule should set `O_FILES` to be the same as `SRC_FILES` but with all the .cpp suffixes replaced with .o suffixes [3].
4. Write and use a makefile that use a rule to generate the dependency list of the .cpp files that are listed in the definition of a macro called `SRC_FILES`. The generated dependency list must be included in the makefile [4].

If a makefile uses the rule you specified in Exercise 3 it is easy to update the makefile for a project when you add a new class to your project. When adding its .cpp filename to the definition of `SRC_FILES` in the makefile, its object file will automatically be added to `O_FILES`. This means that adding a file to your project requires only adding its dependencies and changing one line in your makefile. If the makefile also contain a rule to dynamically generate the required dependencies — like you have done in Exercise 4 — it becomes even easier to update the makefile of a system when the files in the system is changed. It requires editing only one line in your makefile namely the list of .cpp source files. Adapting this makefile to an entire different project is now as simple as replacing the definition of `SRC_FILES`.

Now that you have discovered how `make` can be applied to automate a wide variety of tasks, you should be able to use its capabilities to manage the build process of the systems you may encounter in your future studies and future career successfully.

[3] You will have to find out about macro modification

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