`make` and Makefiles

References

GNU Make is the authoritative reference for `make` (that was installed along with GNU C and C++ compilers).

Why `make`?

Suppose your task is to declare and define function `check_distinct` that takes an object of type `std::vector<int>` and returns `true` if the container elements are distinct, and `false` if two or more elements store the same value. Further suppose you declare the function in header file `q.h`:

```cpp
#ifndef Q_H // header guard
#define Q_H

#include <vector> // for class std::vector

namespace hlp2 { // this defines new scope called hlp2

bool check_distinct(std::vector<int> cont);

} // end of namespace hlp2

#endif // end of header guard
```

and provide the definition in source file `q.cpp`:

```cpp
#include "q.h"

namespace hlp2 { // this defines new scope called hlp2

bool check_distinct(std::vector<int> cont) {
  std::vector<int>::iterator first{cont.begin()}, last{cont.end()};
  while (first != last) {
    std::vector<int>::iterator first2 {first+1};
    while (first2 != last) {
      if (*first2 == *first) {
        return false; // found two elements with same value
      }
      ++first2;
    }
    ++first;
  }
  return true; // all values in container are distinct
}

} // end namespace hlp2
```

The instructor must author a driver source file `qdriver.cpp` and design a variety of test cases to test your definition of `check_distinct`. A little bit of thought from the instructor leads to three test cases with the first test case involving zero values, the second involving a group of `int` values that are all distinct, and the third involving a group of `int` values with one or more values duplicated. Each test case of `int` values must be contained in an object of type `vector<int>` and passed to `check_distinct`. Rather than hard-coding these values in `qdriver.cpp`, a more flexible approach involves storing each set of `int` values in a file and providing the filename as a command-line parameter to the program. An implementation of `qdriver.cpp` looks like this:

```
#include <iostream> // std::cout
#include <fstream>  // std::ifstream
#include <vector>   // std::vector<std::string>
#include <iterator> // std::istream_iterator<std::string>
#include "q.h"      // check_distinct

int main(int argc, char *argv[]) {
  // sanity test to make sure program usage is correct
  if (argc < 3) {
    std::cout << "Usage: ./q.out input-text-file output-text-file\n";
    return 0;
  }

  // sanity test to make sure input file can be opened for reads
  std::ifstream ifs {argv[1]};
  if (!ifs) {
    std::cout << "Unable to open input file " << argv[1] << std::endl;
    return 0;
  }
  std::istream_iterator<int> is{ifs}; // is points to first int in file
  std::istream_iterator<int> eos; // eos now points to end-of-file
  std::vector<int> values(is, eos); // fill values with ints from file
  ifs.close(); // return file resource back to system ...
  std::ofstream ofs {argv[2]}; // open output file
  if (!ofs) {
    std::cout << "Unable to open output file " << argv[2] << std::endl;
    return 0;
  }
  ofs << "Elements are " << (hlp2::check_distinct(values) ? "" : "not")
      << " distinct.\n";
  ofs.close();
}
```

An executable program `q.out` is generated by compiling and linking the source files:

```
g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror q.cpp qdriver.cpp -o q.out
```

As more and more files are added, the `g++` command becomes lengthier and lengthier to type. This is cumbersome, error prone, and unnecessarily repetitive. Doing things repeatedly is what computers are good at but not humans. Another inconvenience is that `g++` will require a lengthier time to build the program since every source file must be recompiled every time even if only a single source or header file is modified. This is acceptable for a small number of files, but becomes a serious problem for large projects consisting of tens or hundreds or thousands of source files. Such recompilations may occur hundreds of times every day causing substantial delays as programmers wait for the executable program to be built. Fortunately, it is possible to avoid unnecessary compilations by compiling individual source files separately to create intermediate object files:

```
$ g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror -c q.cpp
$ g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror -c qdriver.cpp
```

The individual object files and the standard C++ library object files are combined together by a linker to create an executable program:

```
$ g++ -std=c++17 -Wall -Wextra -Werror q.o qdriver.o -o q.out
```

Although the advantages of separate compilation are compelling, it can also be messy. Every time a source file, say `q.cpp`, is updated, typing two commands - one to compile `q.cpp` and the other to link the object files to create an executable - is still repetitive and cumbersome:

```
$ g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror -c q.cpp
$ g++ -std=c++17 -Wall -Wextra -Werror q.o qdriver.o -o q.out
```

Another major problem with separate compilation is that programmers will have to remember *dependencies* between different files. A *dependency* describes the relationship of one file to another in terms of a programming environment. For example, given source file `q.cpp`, object module file `q.o` can be produced by the command

```
$ g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror -c q.cpp
```

which says "compile file `q.cpp` but don't link it; instead, place the object code for `q.cpp` in `q.o`." This means *target* file `q.o` *depends upon* file `q.cpp`, because changes to `q.cpp` will require that it be recompiled in order to produce a new `q.o`. Similarly, if file `who.cpp` exists, then program `who.out` can be created from `who.cpp` (assuming the entire program is in `who.cpp`) by the command

```
$ g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror who.cpp -o who.out
```

which says "compile file `who.cpp`, link it with the standard C++ library, and place the program in executable file `who.out`." Here file `who.out` depends on file `who.cpp`, and any time `who.cpp` is changed, `who.out` has to be remade by recompiling `who.cpp`.

In large programming projects consisting of tens or hundreds or thousands of source and header files, it is exceedingly difficult to keep track of the dependencies required to create an executable from them. For example, if source file `xyz.cpp` includes header file `a.h` which in turn includes another header file `b.h`, and if `b.h` is updated, then `xyz.cpp` must be recompiled even though neither `xyz.cpp` nor `a.h` were altered. To solve this problem, a program called *make* is used. The version of *make* provided by GCC is coincidentally called `make`. `make` is a facility for automating the maintenance and generation of executables from source files. `make` uses a *makefile* that specifies the dependencies between files and the commands that will bring all files up to date and build an executable from these up to date files. `make` gathers modification times of files from the operating system (for example, the `stat` function in Linux) to determine if dependent files have been modified more recently than the target file and if that is the case it will execute commands specified in the *makefile* to create the latest version of the program.

So, to summarize, `make` uses dependencies specified in a *makefile* to produce files. Dependencies are simply the relationship of one file to another based on the file name (for example, `q.out` is dependent upon `q.cpp` and `qdriver.cpp`). If the files that the specified target (say `q.out`) depends on (`q.cpp` and `qdriver.cpp`) have more recent modification times than the target, `make` issues the commands specified in the *makefile* necessary to recreate the target; otherwise, `make` produces the message *'target' is up to date*.

How `make` works

To create an executable program from a set of source and header files, *makefile* must contain the following information:

- the name of source and header files comprising the program
- the interdependencies between these files
- the commands that are required to create the executable

A simple *makefile* consists of *rules* with each *rule* consisting of three parts: a *target*, a list of *prerequisite*s, and a *command* (or a *recipe*). A typical rule has the form:

```
target : prerequiste-1 prerequisite-2 ...
  command-1
  command-2
  ...
```

`target` is the name of the file to be created or an action to be performed by `make`. `prerequisite-1`, `prerequisite-2`, and so on represent the files that will be used as input to create `target`. If any of the prerequisites have more recent modification times than `target`, then `make` will create `target` by executing commands `command-1`, `command-2`, and so on. `make` will terminate and shutdown if any command is unsuccessful.

The most common error with editing a *makefile* is programmers forgetting to put a horizontal tab at the beginning of a command line, and instead place space characters there. Here's what happens if line 12 in a *makefile* is prefixed with space characters rather than a tab:

```
makefile:12: *** missing separator.  Stop.

```

> ***Note that every command (or recipe) in a makefile must be preceded by a tab and not spaces!!!***

Version 1

Here's an example:

```
1 # makefile version 1
2 q.out : qdriver.cpp q.cpp
3   g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror qdriver.cpp q.cpp -o
  q.out
```

Line 1 is a comment. In a *makefile*, anything after a `#` is treated as a comment and ignored.

> ***Comments may be placed in a makefile by putting a `#` in the first column. The `#` (in the first column) followed by the rest of the line are ignored.***

Line 2 says that target `q.out` must be remade (or made if it doesn't exist) if any of the prerequisite files (`qdriver.cpp` and `q.cpp`) have been modified more recently than the target. Before checking modification times of prerequisite files, `make` will look for rules that start with each prerequisite file. If such a rule is found, `make` will make the prerequisite file if any of its prerequisites are newer. After checking that all prerequisite files are up to date and remaking any that are not, `make` brings `q.out` up to date. In the above example, there are no rules that start with prerequisite files `qdriver.cpp` and `q.cpp`,

Line 3 provides the command that `make` should use to remake target `q.out`. Recall that `make` doesn't execute the command if `q.out` has a modification time more recent than `qdriver.cpp` and `q.cpp`. Otherwise, `make` will bring `q.out` up to date by calling `g++` with the usual and required GCC options to compile source files `q.cpp` and `qdriver.cpp` and link the newly created object files with standard C++ library.

Use the code previously described in this section to author source and header files `q.h`, `q.cpp`, and `qdriver.cpp` - we'll use these files in our `make` examples. When the `make` command is executed, it looks for a *makefile* in the current directory named `makefile` or `Makefile` (in that order) and reads it, if found. Create a *makefile* called `makefile` and add the rule described above. Then run the `make` command and observe `make`'s behavior:

```
$ make
g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror qdriver.cpp q.cpp -o
q.out
```

> ***You don't have to specify makefile as either*** `makefile` ***or*** `Makefile` ***- you can give the makefile any name you wish. You can tell*** `make` ***to use any file by adding option*** `-f` ***followed by the makefile's name.***

Notice that you've immediately solved the problem of having to type the cumbersome and lengthy `g++` command. Run the `make` command again:

```
$ make
make: 'q.out' is up to date.
```

By looking at modification times of source files and target file `q.out`, `make` is smart enough to detect that recompilation and relinking are unnecessary.

Now, change only source file `qdriver.cpp` by adding an innocuous comment and run the `make` command:

```
$ make
g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror qdriver.cpp q.cpp -o
q.out
```

Since the modification time of the target precedes the modification time of prerequisite file `qdriver.cpp`, `make` will run the `g++` command to bring `q.out` up to date.

Version 2

The previous `makefile` is inefficient; even though only prerequisite file `qdriver.cpp` has been modified, both prerequisite files are recompiled with a subsequent relinking of the object files. In order to be more efficient, let's amend the rule so that target `q.out` is now dependent on object files `qdriver.o` and `q.o` and further add two new rules to create targets `qdriver.o` and `q.o`:

```
 1  # makefile version 2
 2
 3  q.out : qdriver.o q.o
 4    g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror qdriver.o q.o -o q.out
 5
 6  qdriver.o : qdriver.cpp
 7    g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror -c qdriver.o qdriver.cpp
 8
 9  q.o : q.cpp
10    g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror -c q.o q.cpp
```

Lines 2, 5, and 8 are blanks. `make` discards empty lines - therefore, use blank lines to space out your rules so that they're easier to read by humans.

> ***Blank lines are discarded by*** `make`.

Delete older versions of object files and executable program and then run `make`:

```
$ rm -f *.o *.out
$ make
g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror qdriver.cpp -c qdriver.o
g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror q.cpp -c q.o
g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror qdriver.o q.o -o q.out
```

Let's look at how `make` first creates the dependencies and then the actual target. When you type the `make` command, `make` will execute the first rule that it finds which happens to be on line 3. This line describes the rule: target `q.out` is dependent on object files `qdriver.o` and `q.o` and must be made if it doesn't exist or remade if any of the prerequisite files (`qdriver.o` and `q.o`) have been modified more recently than the target. Before checking modification times of prerequisite files, `make` will look for rules that start with each prerequisite file. A rule exists for `qdriver.o` on line 6 and an other rule exists for `q.o` on line 9. Since `qdriver.o` doesn't exist, `make` will use the rule on lines 6 to create `qdriver.o` from prerequisite file `qdriver.cpp` by running the `g++` command on line 7. Similarly, `make` will use the rule on line 9 to create `q.o` from its prerequisite file `q.cpp` by running the `g++` command on line 10. Since target `q.out` doesn't exist, it will use the newly created prerequisites `qdriver.o` and `q.o` to build `q.out` using the command on line 4. So, now `make` has brought target `q.out` up to date.

Without making any changes to source files `qdriver.cpp` and `q.cpp`, run `make` again:

```
$ make
make: 'q.out' is up to date.
```

Since modification times of prerequisites `qdriver.o` and `q.o` are more recent than source files `qdriver.cpp` and `q.cpp`, and since the modification time of target `q.out` is more recent than its prerequisites, `make` does nothing.

Now, amend `q.cpp` by adding an innocuous comment and run `make`:

```
$ make
g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror q.cpp -c q.o
g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror qdriver.o q.o -o q.out
```

We've achieved efficiency since `make` is now able to deduce that only prerequisite file `q.o` (and not prerequisite file `qdriver.o`) must be made up to date since the modification time of `q.cpp` is more recent than `q.o`. This further means that target `q.out` must be made up to date using the first rule on line 3.

Version 3

Suppose you amend header file `q.h` by adding an innocuous comment and run `make`:

```
$ make
make: 'q.out' is up to date.
```

`make`'s response to changes in `q.h` is incorrect. Object file `qdriver.o` is a combination of both `qdriver.cpp` and `q.h` since inclusion of a header file causes the header file to be read into the source file at that point. Therefore both `qdriver.o` and `q.out` depend upon `qdriver.cpp` and `q.h`, since a change to either file will require recompiling and relinking. Likewise, `q.o` and `q.out` depend on both `q.cpp` and `q.h`. `make` doesn't have the smarts to handle this all by itself. It doesn't know what's included in a program, so it has to be told. This is where `makefile` comes in handy: you can specify to `make` via the `makefile` that included files are prerequisites to the program. This can be done by adding the following dependency lines to `makefile`:

```
# makefile version 3

q.out : qdriver.o q.o
  g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror qdriver.o q.o -o q.out

qdriver.o : qdriver.cpp q.h
  g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror -c qdriver.o qdriver.cpp

q.o : q.cpp q.h
  g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror -c q.o q.cpp
```

Remove previous object files and executable, run `make`, and observe `make`'s behavior:

```
$ rm -f *.o *.out
$ make
g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror qdriver.cpp -c qdriver.o
g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror q.cpp -c q.o
g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror qdriver.o q.o -o q.out
```

Next, add an innocuous comment to `q.h` and run `make`:

```
$ make
g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror qdriver.cpp -c qdriver.o
g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror q.cpp -c q.o
g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror qdriver.o q.o -o q.out
```

`q.h` is a prerequisite for both `qdriver.o` and `q.o` and has a more recent modification time than the object files. Therefore, `make` must make the target on line 6 up to date, followed by the target on line 9 followed by the target on line 3.

`make` variables

`make` allows you to assign strings to variables and later recall their contents. `make` variables are sometimes called *macros*. A `make` variable is assigned a value by using the variable name on the left-hand side of an equal sign `=`: $variable = value$. $variable$ may consist of any character except those with special meaning to `make`, e.g., `#`, `:`, `;`, `=`, `?`, `@`, blank, tab, newline. In general, you should not use any characters other than alphanumerics. Also, the convention is to specify variables using uppercase letters.

$value$ may be any string of characters up to a newline that isn't preceded by a `\` (continuation). Spaces around the `=` are optional.

You access `make` variables by enclosing them in parentheses and preceding them with a dollar sign. This causes the contents of the variable to be substituted at that point. Consider the following example:

```
EXEC = q.out
$(EXEC) : qdriver.o q.o
  g++ -std=c++17 -pedantic-errors -Wall -Wextra -Werror qdriver.o q.o -o $(EXEC)
```

Here we have a `make` variable `EXEC` that is assigned value `q.out`. The value of variable `EXEC` is accessed by either `$(EXEC)` or `${EXEC}`. `make` will replace every occurrence of either `$(EXEC)` or `${EXEC}` in `makefile` with `q.out`.

Why are variables useful? A general principle in software design is to use variables to express some common things. If changes are needed later, these modifications can be made in only one place. For example, if output file name is to be changed from `q.out` to `q`, only a single line of `makefile` needs to be updated. By using a variable, we ensure that the change is consistent throughout the entire `makefile`.

`make` has certain pre-defined variables. For C++ programmers, the variables that come into play are `CXX` and `CXX_FLAGS`. `CXX` is the default C++ compiler command used by `make` and is set to `g++`. `CXX_FLAGS` is the variable specifying the options provided to the C++ compiler and is normally set to the empty string.

Version 4

We can add more variables to limit changes to `makefile` plus three new rules called `all`, `clean`, and `rebuild`:

```
# makefile version 4

```

```
 3 # name of C++ compiler
 4 CXX       = g++
 5 # options to C++ compiler
 6 CXX_FLAGS = -std=c++17 -pedantic-errors -Wall -Wextra -Werror
 7 # flag to linker to make it link with math library
 8 LDLIBS    = -lm
 9 # list of object files
10 OBJS      = qdriver.o q.o
11 # name of executable program
12 EXEC      = q.out
13 
14 # by convention the default target (the target that is built when
15 # writing only make on the command line) should be called all and
16 # it should be the first target in a makefile
17 all : $(EXEC)
18 
19 # however, the problem that arises with the previous rule is that make
20 # will think all is the name of the target file that should be created
21 # so, we tell make that all is not a file name
22 .PHONY : all
23 
24 # this rule says that target $(EXEC) will be built if prerequisite
25 # files $(OBJS) have changed more recently than $(EXEC)
26 # text $(EXEC) will be replaced with value q.out
27 # text $(OBJS) will be substituted with list of prerequisites in line 10
28 # line 31 says to build $(EXEC) using command $(CXX) with
29 # options $(CXX_FLAGS) specified on line 6
30 $(EXEC) : $(OBJS)
31 	$(CXX) $(CXX_FLAGS) $(OBJS) -o $(EXEC) $(LDLIBS)
32 
33 # target qdriver.o depends on both qdriver.cpp and q.h
34 # and is created with command $(CXX) given the options $(CXX_FLAGS)
35 qdriver.o : qdriver.cpp q.h
36 	$(CXX) $(CXX_FLAGS) -c qdriver.cpp -o qdriver.o
37 
38 # target q.o depends on both q.cpp and q.h
39 # and is created with command $(CXX) given the options $(CXX_FLAGS)
40 q.o : q.cpp q.h
41 	$(CXX) $(CXX_FLAGS) -c q.cpp -o q.o
42 
43 # says that clean is not the name of a target file but simply the name for
44 # a recipe to be executed when an explicit request is made
45 .PHONY : clean
46 # clean is a target with no prerequisites;
47 # typing the command in the shell: make clean
48 # will only execute the command which is to delete the object files
49 clean :
50 	rm -f $(OBJS) $(EXEC)
51 
52 # says that rebuild is not the name of a target file but simply the name
53 # for a recipe to be executed when an explicit request is made
54 .PHONY : rebuild
55 # rebuild is for starting over by removing cleaning up previous builds
56 # and starting a new one
57 rebuild :
58 	$(MAKE) clean
59 	$(MAKE)
```

Running command `make` in the Linux shell will invoke the first rule which is to create target `$(EXEC)` or `q.out`. By convention, the name for the first rule that does the main work of building an executable program is given the name `all` or `default`. Line 17 specifies target `all` with a prerequisite `$(EXEC)`. Line 22 tells `make` that it should not confuse the rule called `all` with a file called `all` (if such a file exists) by using target name `.PHONY`.

Phony target `clean` on line 49 is different from the other targets; it has no prerequisites. If the following command is issued in the shell:

```
$ make clean
rm -f qdriver.o q.o q.out
$
```

then `make` will execute only the command on line 50 in rule `clean` and then exit. This target is useful for starting over from scratch because it removes object files and the executable program created by previous runs of `make`.

Line 57 adds a phony target `rebuild` that recursively calls `make` to rebuild the executable program:

```
$ make rebuild
```

The `rebuild` target essentially starts over by first recursively running `make` with target `clean` followed by the `make` command to rebuild `q.out`.

Summary

You now have a perfectly good *makefile* suitable for labs and programming assignments. It can be enhanced with additional features so that it becomes suitable for small and medium-sized software projects. These features and enhancements will not be covered in this document. Let's summarize some important details about `make` and *makefiles*.

If you name your *makefile* `makefile`, then you don't have to tell `make` which file to use. This is the recommended way of doing it. To build the project, just type:

```
$ make
```

If the file is named something else, say `gcc.mk`, you should provide `make` your *makefile* like this:

```
$ make -f gcc.mk
```

By default, `make` executes the first target (also called `all`), which should be the ultimate goal. Usually, that means linking the final project.

If you want to execute a different target, you must specify that target name. For example, to execute only target `clean`, do this:

```
$ make clean
```

The value assigned to a variable in the *makefile* can be overridden by specifying a new value in the command line. For example, to compile with `clang++` and not default compiler `g++`, do this:

```
$ make clean
$ make CXX=clang++
```

The `make clean` command is required to avoid sending the linker object files created by different compilers.

Testing `check_distinct`

It is important to understand a few *truths* about testing programs. It is possible to test a program and demonstrate that the program is incorrect. It is almost impossible to test a program and demonstrate that the program is correct. If a test fails, then we know that the program is wrong. If a program passes a test, what do we know about the program? Not much.

This may seem puzzling. If a program passes many tests, it must be correct, right? Not really. In a way, this is like the theory of black swans. Even though you may have seen many thousands of white swans, you can't say much about whether black swans exist or not. In contrast, if you see one black swan, you can say that black swans exist. Likewise, passing many thousands of tests doesn't tell us whether a program is correct. In contrast, if the program fails one test, we know that the program is incorrect.

All of this means that testing is extremely hard and important. Passing many tests gives you some confidence, but no guarantee. A non-trivial program can have many test cases. Even though testing is imperfect, testing is still useful in developing programs. So, let's think about how to test function `check_distinct`.

Once executable program `q.out` has been created by running `make` (using `makefile` - version 4), you must test your definition of function `check_distinct` by building test cases. You need a test case to consider the situation where the container has zero values and test cases to consider the situation where the container has values that are distinct or not. This means that you need at least three test cases: an empty file `empty.txt` that will test for the situation where the container has zero values; a file `distinct.txt` with distinct values so that there are no duplicates; and a file `duplicate.txt` with duplicate values.

Creating an empty file is easy:

```
$ touch empty.txt
```

The second and third files can be created by hand. Alternatively, test cases can be developed using an online random number generator and saving the random values to a file `duplicate.txt`. You can create a file `distinct.txt` containing only distinct values by using the Linux command `sort` with option `-u`:

```
$ sort -u duplicate.txt > distinct.txt
```

Once we've created the necessary data for the three test cases, we run the executable program for each test case by providing the appropriate input and output file names as command-line parameters to the program. For example:

```
$ ./q.out empty.txt yourout-empty.txt
```

The output generated by the program is written to text file `yourout-empty.txt`. Suppose file `out-empty.txt` contains the correct program output for the test case specified in `empty.txt`. To compare your output with the expected output, you would use the `diff` command:

```
$ diff -y --strip-trailing-cr --suppress-common-lines yourout-empty.txt out-empty.txt
```

The `diff` options are listed here. If `diff` is not silent, then one or more of your function definitions is incorrect and will require further work.

Adding tests to *makefile*

As we've seen, `make` can reduce the amount of repetitive typing. It can do more. With `make` we can create tasks with dependent tasks, and only the required tasks are rerun when files are modified. We can add rules to execute testing tasks in our `makefile`. Consider the following dependence rule:

```
1  .PHONY : test-empty
2  test-empty : $(EXEC)
3    ./$(EXEC) empty.txt yourout-empty.txt
4    diff -y --strip-trailing-cr --suppress-common-lines yourout-empty.txt out-empty.txt
```

To execute the commands associated with this recipe, we type:

```
$ make test-empty
```

This dependence rule follows the same phony rules mentioned earlier. Because `test-empty` is not a file, its modification time can never be later than the time of `$(EXEC)` which is the executable program created by `make`. As a result, command `./q.out` on line 3 and command `diff` on line 4 will always be executed. Before executing these two commands, `make` checks the dependence on prerequisite `q.out`. `make` finds this rule in the `makefile`

```
$(EXEC) : $(OBJS)
  $(CXX) $(CXX_FLAGS) $(OBJS) -o $(EXEC) $(LDLIBS)
```

and `make` compares the modification times of `q.out` with those of object files `qdriver.o` and `q.o`. Recursively, `make` will look at rules for building `qdriver.o` and `q.o`

```
qdriver.o : qdriver.cpp q.h
  $(CXX) $(CXX_FLAGS) -c qdriver.cpp -o qdriver.o

# target q.o depends on both q.cpp and q.h
# and is created with command $(CXX) given the options $(CXX_FLAGS)
q.o : q.cpp q.h
  $(CXX) $(CXX_FLAGS) -c q.cpp -o q.o
```

and `make` compares the modification times of `qdriver.o` with those of `qdriver.cpp` and `q.h` and the modification times of `q.o` with those of `q.cpp` and `q.h`. Each object file will be made up to date if it is older than the corresponding `.cpp` and `.h` files. If one or both object files are made up to date, then the program will bring `q.out` up to date.

Because of these dependencies, when you type

```
$ make test-empty
```

the executable program `q.out` will be built if either `.cpp` or the `.h` file has changed since the last time `make` was invoked before the actual test itself is executed.

More rules can be added to the `makefile` for running the remaining two cases:

```
.PHONY : test-distinct
test-distinct : $(EXEC)
	./$(EXEC) distinct.txt yourout-distinct.txt
	diff -y --strip-trailing-cr --suppress-common-lines yourout-distinct.txt out-distinct.txt

.PHONY : test-duplicate
test-duplicate : $(EXEC)
	./$(EXEC) duplicate.txt yourout-duplicate.txt
	diff -y --strip-trailing-cr --suppress-common-lines yourout-duplicate.txt out-duplicate.txt
```

To test each case, type

```
$ make test-empty
$ make test-distinct
$ make test-duplicate
```

Another rule can be used to test all test cases at once:

```
.PHONY : test-all
test-all : test-empty test-distinct test-duplicate
```

So now we've taken a lot of repetitive typing in the Linux shell and replaced it with target calls to `make` which will take care of keeping track of modification times of dependent files and bringing only those files that are not up to date current. That is, `make` and *makefile* automate the work flow involved in the coding and testing of programming projects.