1 Summary

- Makefiles

2 Make and Makefiles

- With single-file programs, compilation is a breeze:

  ```
g++-5 -std=c++14 change.cc -o change
  ```

- However, when we have a project across multiple files, compilation may become a pain to type out. Surely there is a better way to compile a project without typing all .cc files.

- We’ve told you that you should use separate compilation which looks something like

  ```
g++-5 -std=c++14 -c main.cc
g++-5 -std=c++14 -c list.cc
...
g++-5 -std=c++14 main.o list.o ... -o myprogram
  ```

- When we do this, we only have to recompile the modules that change. This means less time compiling but more time remembering what we have recently compiled.

- Surely there must be a better way to keep track of changes. This is a bigger issue when we would constantly be recompiling everything when we don’t have to.

- **make** can help here. It allows us to specify the dependencies of targets (the files produced by the build process) and the command for producing each target in a Makefile. **make** will automate the building process and avoid unnecessary compilation by keeping track of changed files based on last modified time \(^1\).

  A Makefile will look something like:

\(^1\)i.e. if a target is newer than its dependencies, then there is no need to rebuild this target
# example1/Makefile

# means myprogram depends on these dependencies
# if all of the dependencies are up to date, then execute
# the build command below
myprogram: main.o list.o iter.o node.o
    # specifies how to build main
    g++-5 -std=c++14 main.o list.o iter.o node.o -o myprogram

list.o: list.cc list.h node.h
    g++-5 -std=c++14 -c list.cc

node.o: node.cc node.h list.h
    g++-5 -std=c++14 -c node.cc

iter.o: iter.cc list.h node.h
    g++-5 -std=c++14 -c iter.cc

main.o: main.cc list.h
    g++-5 -std=c++14 -c main.cc

• The whitespaces before the build command (in this case, g++ ...) MUST be a tab.

• On the command line, run make. This will build our project.

• If list.cc changes, what happens?
  – compile list.cc
  – relink myprogram

• What happens when we execute the command make?
  – Builds first target in our Makefile, in this case myprogram.
  – What does myprogram depend on?
    * main.o list.o iter.o node.o
  – If list.cc changes:
    * list.cc is newer (timestamp) than list.o; rebuilds list.o
    * list.o is newer (timestamp) than myprogram; rebuilds myprogram

• Tip: We can build specific targets using make:

    make node.o

• Common practice: put a clean target at the end of a makefile to remove all binaries

---

2The description found in https://www.gnu.org/software/make/manual/html_node/Phony-Targets.html
# example2/Makefile

clean:
   rm *.o main

# clean is a ‘phony target’: it is not name of a file but
# a recipe to be executed when an explicit request is made

.PHONY: clean

• To do a full rebuild:

   make clean && make

• While Makefile can make our compilation process easier, writing out all of the dependencies
  and individual compilation commands can be time consuming.

• Conveniently, we can generalize a Makefile with variables.

# example3/Makefile

CXX = g++-5 #compiler name
CXXFLAGS = -std=c++14 -Wall #options to pass
OBJECTS = main.o list.o iter.o node.o
EXEC = myprogram

${EXEC}: ${OBJECTS}
   ${CXX} ${OBJECTS} -o ${EXEC} ${CXXFLAGS}

list.o: list.cc list.h node.h
   ${CXX} ${CXXFLAGS} -c list.cc

... 

• Shortcut: For any rule of the form x.o: x.cc a.h b.h [more .h files], we can leave out
  the build command. make will guess that it is ${CXX} ${CXXFLAGS} -c list.cc -o list.o

# example4/Makefile

CXX = g++-5 #compiler name
CXXFLAGS = -std=c++14 -Wall #options to pass
OBJECTS = main.o list.o iter.o node.o
EXEC = myprogram

# This one is not in the generic form, so have to list it

${EXEC}: ${OBJECTS}
   ${CXX} ${CXXFLAGS} ${OBJECTS} -o ${EXEC}

list.o: list.cc list.h node.h

node.o: node.cc node.h list.h

...
• Issue: how to track dependencies and updating them as they change
  
  – `g++` can help. `g++14 -MMD -c list.cc` will create `list.o list.d`.
  – What will `list.d` contain?
    
    `list.o: list.cc list.h node.h`

• Looking at this `.d` file, we can see it is exactly what we need in our `Makefile`. We just need to include all `.d` files in our `Makefile`. This means our `Makefile` will look like

```
# example5/Makefile
CXX = g++-5
CXXFLAGS = -std=c++14 -Wall -MMD
OBJECTS = main.o list.o iter.o node.o
DEPENDS = ${OBJECTS:.o=.d}
EXEC = myprogram

${EXEC}: ${OBJECTS}
  ${CXX} ${CXXFLAGS} ${OBJECTS} -o ${EXEC}

-include ${DEPENDS}

.PHONY: clean

clean:
  rm ${OBJECTS} ${EXEC} ${DEPENDS}
```

• This is the final version of our `Makefile`. Altering the variables of this `Makefile`, we can use this exact `Makefile` for basically any program we want to create.