

C++ For C Coders 8

Build Process

Data Structures
C++ for C Coders

한동대학교 김영섭 교수
idebtor@gmail.com

build process
compile & link
static library
make & Makefile

Build Process

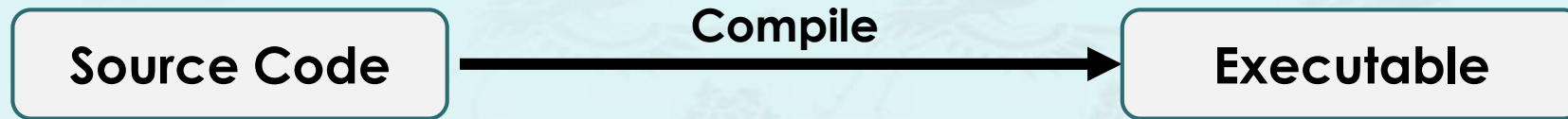
Build Process

1. Compile
2. Link
3. Build a static library
4. Make and Makefile

Build Process

The term build process here refers to the steps starting with source code (**a set of .cpp and .h files**) ending with an executable file representing your program.

- A simplistic view of the build process



```
$ g++ sort.cpp print_list.cpp bubble.cpp quicksort.cpp -I../include -o sort
```

- A simple but realistic view of the build process



```
$ g++ -c sort.cpp print_list.cpp -I../include
```

Compile

```
$ g++ sort.o print_list.o -L../lib -lsort -o sort
```

Link

or

```
$ g++ -c sort.cpp print_list.cpp -I../include -L../lib -lsort -o sort
```

(Assume that you have **libsort.a** in **../lib** folder.)

Compile & Link

Building an executable for a program consists of two major stages:

- **Compile stage (.cpp, .h → .o)**
 - **Syntax** checked for correctness.
 - Variables and function calls checked to insure that correct declarations were made and that they match.
 - It **doesn't match function definitions** to their calls at this point.
 - Translation into **object code**. It is not an executable.
- **Linking stage (.o, .a → .exe)**
 - Links the object code into an executable.
 - May involve one or more object code files.
 - **Function calls** are matched up with their definitions, and the compiler checks to make sure it has one, and only one, definition for every function.
 - The end result of linking is usually an executable.

Compiling options

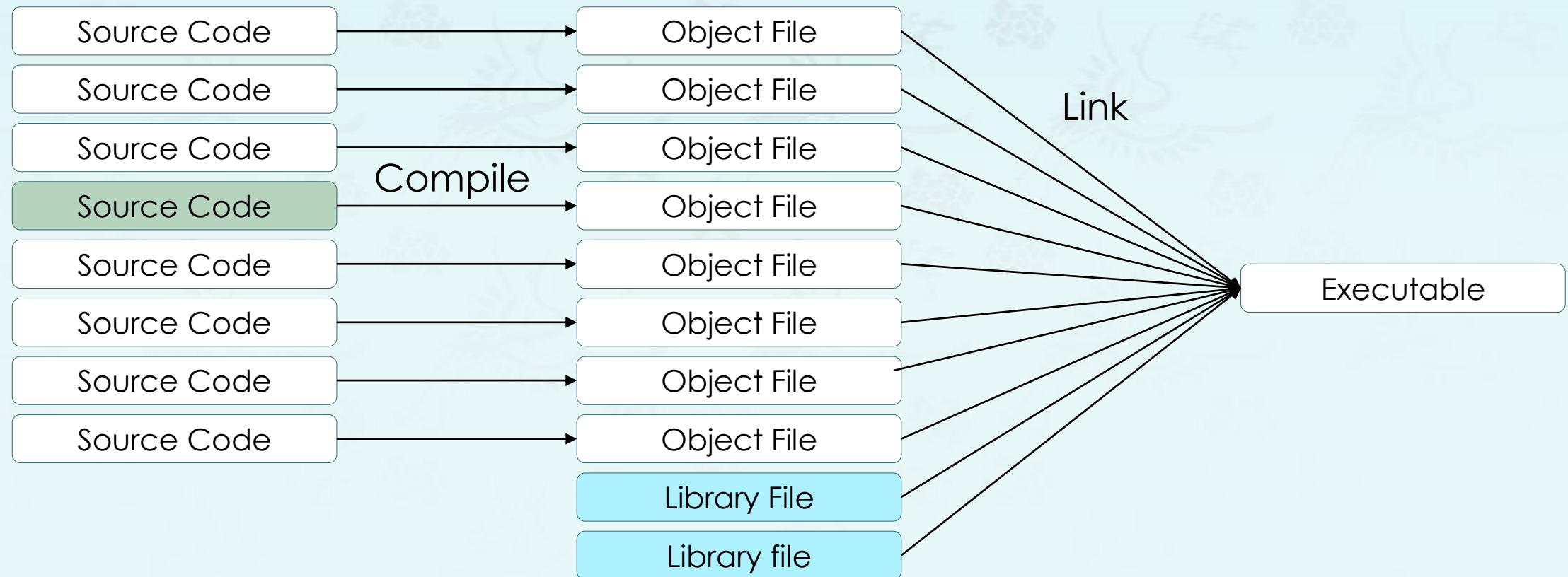
- **-g** - turn on debugging (so GDB gives more friendly output)
- **-Wall** - turns on most warnings
- **-O** or **-O2** - turn on optimizations
- **-o <name>** - name of the output file
- **-c** - output an object file (**.o**)
- **-I<include path>** - specify an **include** directory
- **-L<library path>** - specify a **lib** directory
- **-l<library>** - link with library **lib<library>.a**

- Use **-Ldir** option such that linker looks for library files in **dir** folder.
Use **-llibrary** such that linker searches the library named **library**.

Build Process

Addressing the build process efficiency:

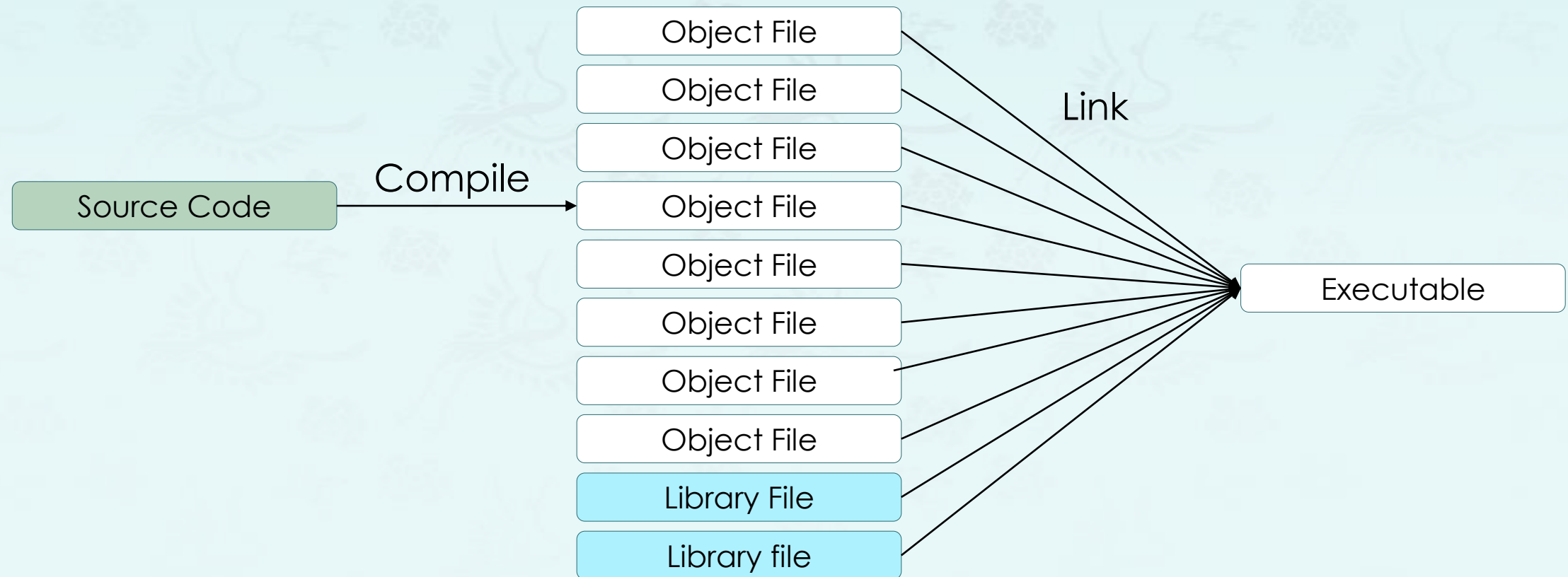
- In this case we highlight the situation where ONE of those files has changed since the previous build. In building the naive brute-force way, ALL source code is recompiled, even those source code files that have not changed.



Build Process

Addressing the build process efficiency:

- Only one source file has changed since the last build. This is the only file re-compiled on the following build. The build otherwise uses the unchanged object files from the previous build, shortening the overall build time.



Creating a Library Archive

An archive in C/C++ is a file that bundles a set of object files into a single file.

- This file always follows the naming convention of starting with **lib** and ending with **.a**, e.g., `libsort.a` and `libnowic.a` in our library examples.
- An archive file can be build from object files using the **ar** command:

```
$ ar cr libsort.a bubble.o insertion.o quicksort.o selection.o
```

- **ar** Options
 - c: Create an archive file
 - r: Insert the files member... into archive (with replacement).
 - s: Write an object-file index into the archive, change is made to the archive
 - t: Display contents of archive (show the list of .o files, use `nm ~.o` to see functions in ~.o)

Creating a Library Archive

- **ar** Examples:
 - `g++ -c nowic.cpp -I../include` // produces `nowic.o`
 - `ar crs libnowic.a nowic.o` // produces `libnowic.a` that includes `nowic.o`
 - `ar` // list all the options available
 - `ar t libnowic.a` // list `~.o` files archived
 - `ar x libnowic.a` // extract `~.o` files archived
 - `nm nowic.o` // list the actual function names in `.o` file
- You may refer to **`/nowic/UsingStaticLib.md`**.

The make utility

- Building a program from its source files can be a complicated and time-consuming operation. The commands are too long to be typed in manually every time. However, a straightforward shell script is seldom used for compiling a program, because it's too time-consuming to recompile all modules when only one of them has changed.
- However, it's too error-prone to allow a human to tell the computer which files need to be recompiled. Forgetting to recompile a file can mean hours of frustrating debugging. A reliable automatic tool is necessary for determining exactly which modules need recompilation.
- A standard tool for solving exactly this problem is called **make**. It relies either on its own built-in knowledge, or on a file called a **Makefile** that contains a detailed recipe for building the program.

references

- <http://nuclear.mutantstargoat.com/articles/make/>
- <https://skandhurkat.com/post/makefile-dependencies/>

The make utility

- You may need to install some packages.
(Install it as **admin privilege**. 관리자모드로 설치하십시오)

```
$ pacman -S base-devel          #install the build toolchain  
$ pacman -Syu                   #update msys2
```

The make utility – a simple example of Makefile

- **Basic syntax for Makefile:**

```
target: dependencies
<tab>system command(s)
```

- **Example:**

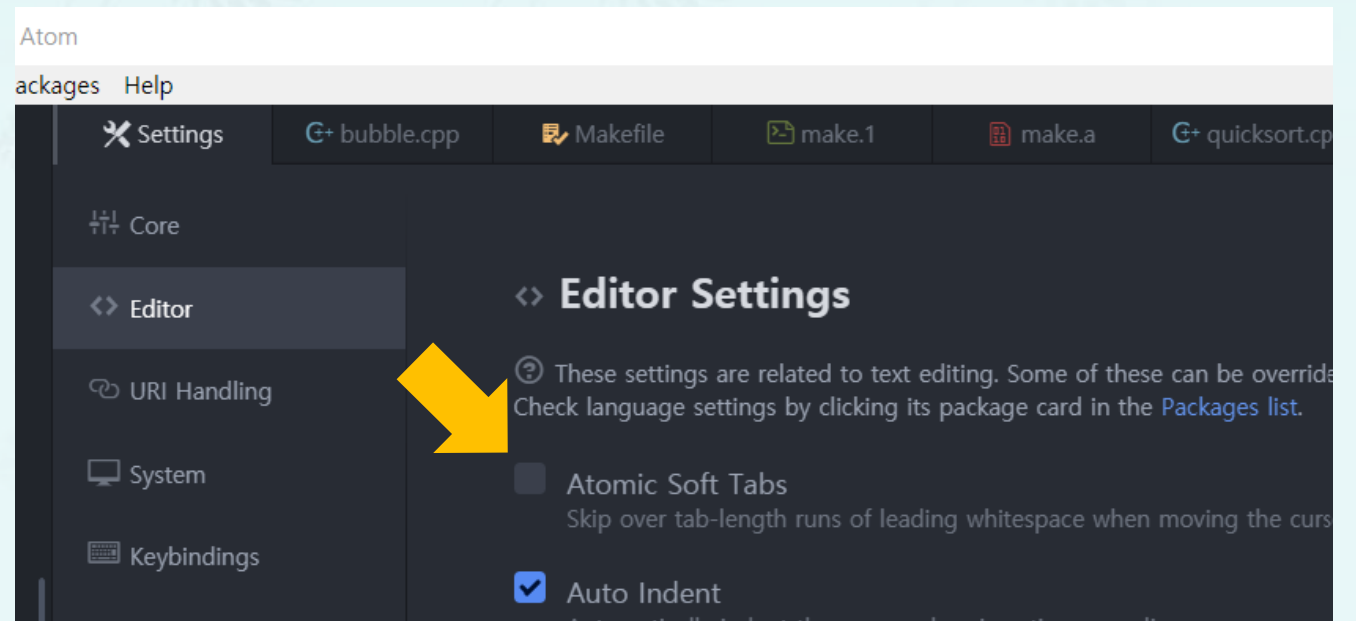
Source files: quicksort.cpp,
 print_list.cpp
Executable: qsort.exe

- **Makefile:**

```
qsort: quicksort.o print_list.o
    g++ quicksort.o print_list.o -o qsort

quicksort.o: quicksort.cpp
    g++ -c quicksort.cpp
print_list.o: print_list.cpp
    g++ -c print_list.cpp
clean:
<tab>rm -f *.o qsort.exe qsort
```

- Use a hard <tab>
- Restart atom after turning on Hard Tabs



The make utility – a simple example of Makefile

- **Basic syntax for Makefile:**

```
target: dependencies
<tab>system command(s)
```

- **Example:**

Source files: quicksort.cpp,
 print_list.cpp
Executable: qsort.exe

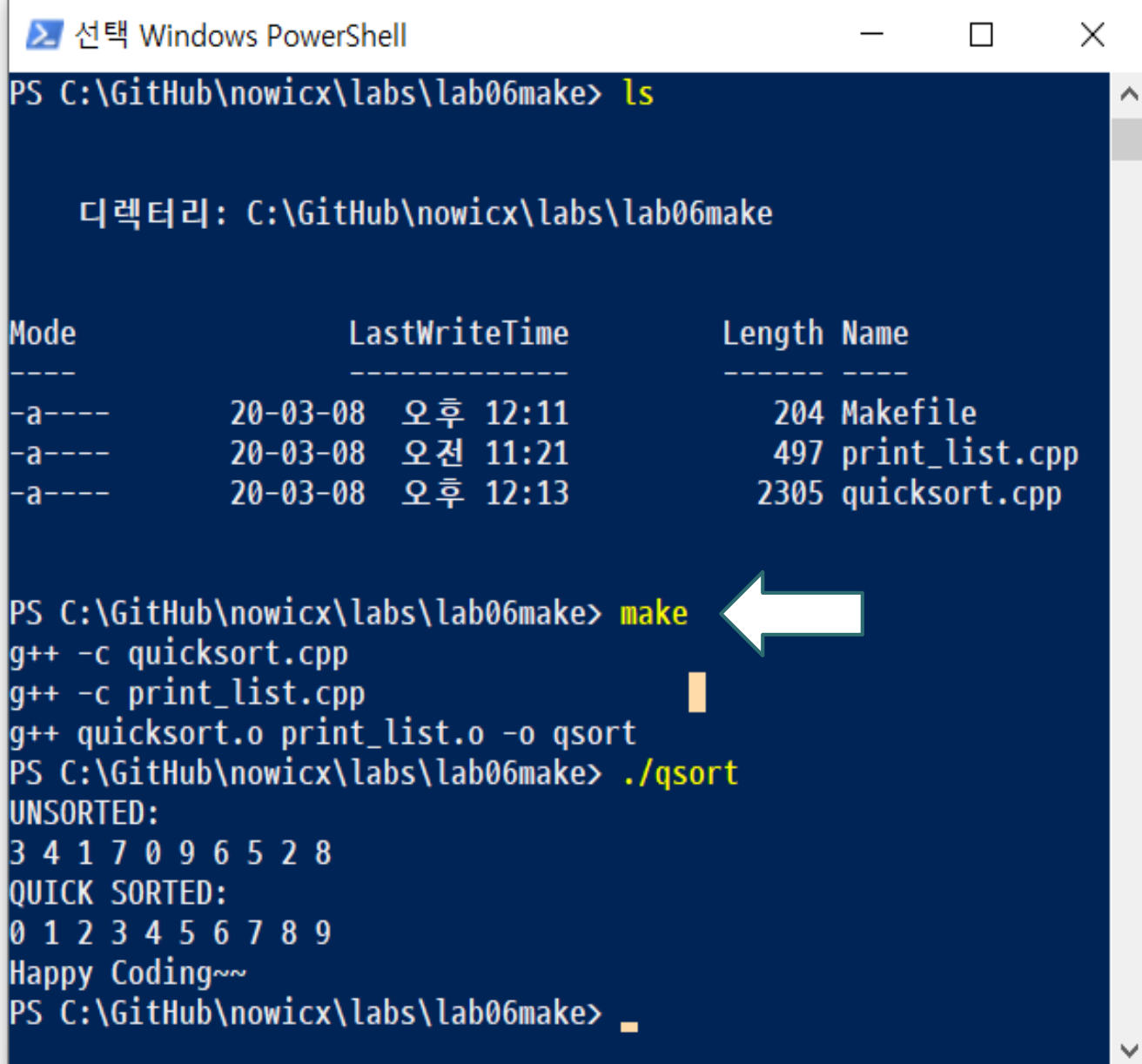
- **Makefile:**

```
qsort: quicksort.o print_list.o
      g++ quicksort.o print_list.o -o qsort
```

```
quicksort.o: quicksort.cpp
      g++ -c quicksort.cpp
print_list.o: print_list.cpp
      g++ -c print_list.cpp
```

```
clean:
```

```
<tab>rm -f *.o qsort.exe qsort
```



The screenshot shows a Windows PowerShell window titled "선택 Windows PowerShell". The current directory is "C:\GitHub\nowicx\labs\lab06make". The user enters the command "ls", and the output shows a directory listing with columns for Mode, LastWriteTime, Length, and Name. The files listed are Makefile (204 bytes), print_list.cpp (497 bytes), and quicksort.cpp (2305 bytes). The user then enters the command "make", and the output shows the compilation of quicksort.cpp and print_list.cpp, followed by the linking of the object files into the executable qsort. The user then enters the command "./qsort", and the output shows the execution of the program, which prints "UNSORTED:" followed by the numbers 3 4 1 7 0 9 6 5 2 8, and "QUICK SORTED:" followed by the numbers 0 1 2 3 4 5 6 7 8 9. The user then enters the command "Happy Coding~~", and the output shows the prompt "PS C:\GitHub\nowicx\labs\lab06make>". A white arrow points to the "make" command in the terminal.

```
PS C:\GitHub\nowicx\labs\lab06make> ls

다렉터리: C:\GitHub\nowicx\labs\lab06make

Mode                LastWriteTime         Length Name
----                -
-a----          20-03-08 오후 12:11             204 Makefile
-a----          20-03-08 오전 11:21             497 print_list.cpp
-a----          20-03-08 오후 12:13            2305 quicksort.cpp

PS C:\GitHub\nowicx\labs\lab06make> make
g++ -c quicksort.cpp
g++ -c print_list.cpp
g++ quicksort.o print_list.o -o qsort
PS C:\GitHub\nowicx\labs\lab06make> ./qsort
UNSORTED:
3 4 1 7 0 9 6 5 2 8
QUICK SORTED:
0 1 2 3 4 5 6 7 8 9
Happy Coding~~
PS C:\GitHub\nowicx\labs\lab06make> .
```

The make utility – a cruel example of Makefile

```
sortx: sortx.o print_list.o bubble.o insertion.o quicksort.o selection.o
    g++ -o sortx sortx.o print_list.o bubble.o insertion.o quicksort.o selection.o
sortx.o: sortx.cpp
    g++ -c sortx.cpp -I../../include
print_list.o: print_list.cpp
    g++ -c print_list.cpp
bubble.o: bubble.cpp
    g++ -c bubble.cpp
insertion.o: insertion.cpp
    g++ -c insertion.cpp
quicksort.o: quicksort.cpp
    g++ -c quicksort.cpp
selection.o: selection.cpp
    g++ -c selection.cpp
clean:
    rm -f *.o
cleanx:
    rm -f *.o sortx.exe sortx
```

Using Rules

```
INCDIR = ../../include
```

```
SRCS = sortx.cpp print_list.cpp bubble.cpp ...
```

```
OBJS = $(SRCS:.cpp=.o)
```

```
TARGET = sortx
```

```
$(TARGET): $(OBJS)
```

```
    g++ -I$(INCDIR) $(SRCS) -o $(TARGET)
```


The make utility – a typical Makefile

```
CC = g++
CCFLAGS = -Wall -std=c++11
LDFLAGS = -L$(LIBDIR) -lsort -lnowic -lm
LIBDIR = ../lib
INCDIR = ../include
SRCS = $(wildcard *.cpp)
OBJS = $(SRCS:.cpp=.o)
TARGET = sortx
TARGET: $(OBJS)
    $(CC) $(CCFLAGS) -I$(INCDIR) -o $@ $^ $(LDFLAGS)

.PHONY: clean
clean:
    rm -f $(OBJS) $(TARGET)
```

\$@ - refers to the target
\$^ - refers to all dependencies
\$< - refers to the first dependency
% - make a pattern that we want to watch
in both the target and the dependency

The make utility – make.1

```
# make.1 - incomplete makefile without automatic dependencies
CC = g++
CCFLAGS = -Wall -std=c++11
LDFLAGS = -L$(LIBDIR)
LIBDIR = ../../lib
INCDIR = ../../include
SRCS = sortx.cpp print_list.cpp bubble.cpp insertion.cpp quicksort.cpp selection.cpp
OBJS = $(SRCS:.cpp=.o)
TARGET = sortx
%.o: %.cpp
    $(CC) -c -I$(INCDIR) -o $@ $< $(CCFLAGS)
$(TARGET): $(OBJS)
<tab> $(CC) -o $@ $^ $(LDFLAGS)
.PHONY:clean cleanx
clean:
    rm -f $(OBJS)
cleanx:
    rm -f $(OBJS) $(TARGET).exe $(TARGET)
```

The following dependency is unchecked.
sortx.cpp depends on include/sort.h

```
$ make -f make.1
$ make clean -f make.1
$ make cleanx -f make.1
```

The make utility – make.2

```
# make.2 - make using auto dependencies
CC = g++
CCFLAGS = -Wall -std=c++11
LDFLAGS = -L$(LIBDIR)
LIBDIR = ../../lib
INCDIR = ../../include

SRCS = sortx.cpp print_list.cpp \
      bubble.cpp insertion.cpp \
      quicksort.cpp selection.cpp
OBJS = $(SRCS:.cpp=.o)
DEPS = $(SRCS:.cpp=.d)
TARGET = sortx
# make target (executable)
$(TARGET): $(OBJS)
    $(CC) -o $(TARGET) $(OBJS) $(LDFLAGS)
```

```
# compile & automatic dependency generation
%.o: %.cpp
    <tab>$(CC) -c $(CCFLAGS) -I$(INCDIR) $< -o $@
    <tab>$(CC) -I$(INCDIR) -MM -MF $*.d $<
    -include $(DEPS)

.PHONY: all debug clean cleanx
all: $(TARGET)
debug: CCFLAGS += -DDEBUG -g
debug: all
clean:
    rm -f $(OBJS) $(DEPS)
cleanx:
    rm -f $(OBJS) $(DEPS) $(TARGET).exe $(TARGET)
```

```
$ make -f make.2
$ make debug -f make.2
$ make clean -f make.2
$ make cleanx -f make.2
```

Build a static library – libsort.a

```
~/include>      sort.h
~/lib>          libsort.a
~/labs/lab06    sort.cpp print_list.cpp bubble.cpp insertion.cpp ...

~/labs/lab06$  g++ -c bubble.cpp
~/labs/lab06$  g++ -c insertion.cpp
~/labs/lab06$  g++ -c quicksort.cpp
~/labs/lab06$  g++ -c selection.cpp
~/labs/lab06$  ar cru libsort.a bubble.cpp insertion.cpp quicksort.cpp selection.cpp
~/labs/lab06$  ar t libsort.a
~/labs/lab06$  cp libsort.a ../../lib
```

Build an executable using a static lib

```
~/include>          sort.h
~/lib>              libsort.a
~/labs/lab06        sort.cpp print_list.cpp

~/labs/lab06> g++ sort.cpp print_list.cpp -I../../include -L../../lib -lsort -o sort
```

Build a static library – make.3

```
# make.3 - build a static library
CC = g++
CCFLAGS = -Wall -std=c++11
SRCS = bubble.cpp insertion.cpp quicksort.cpp selection.cpp
OBJS = $(SRCS:.cpp=.o)
DEPS = $(SRCS:.cpp=.d)
TARGET = libsort.a
$(TARGET): $(OBJS)
    ar cru $@ $^
    ranlib $@
$(OBJS): %.o: %.cpp
    $(CC) -c $(CCFLAGS) $< -o $@
    $(CC) -MM -MF $*.d $<
-include $(DEPS)
clean:
    rm -f $(OBJS) $(DEPS) $(TARGET)
```

\$@ - refers to the target
\$^ - refers to all dependencies
\$< - refers to the first dependency
\$* - wildcard (or any number of characters)
% - make a pattern that we want to watch
in both the target and the dependency

Build an executable using a static lib – make.4

```
# make.4 - makefile using a static lib
CC = g++
CC = g++
CCFLAGS = -Wall -std=c++11
LDFLAGS = -L$(LIBDIR) -lsort
LIBDIR = ../../lib
INCDIR = ../../include

SRCS = sortx.cpp print_list.cpp
OBJS = $(SRCS:.cpp=.o)

TARGET = sortx
# make target (executable)
$(TARGET): $(OBJS)
    $(CC) -o $(TARGET) $(OBJS) $(LDFLAGS)
```

```
# compile & automatic dependency generation
$(OBJS): %.o: %.cpp
    $(CC) -c $(CCFLAGS) -I$(INCDIR) $< -o $@
    $(CC) -I$(INCDIR) -MM -MF $*.d $<
-include $(SRCS:.cpp=.d)

.PHONY: all debug clean cleanx
all: $(TARGET)
debug: CCFLAGS += -DDEBUG -g
debug: all
clean:
    rm -f $(OBJS) *.d
cleanx:
    rm -f $(OBJS) *.d $(TARGET).exe $(TARGET)
```

```
$ make -f make.4
$ make debug -f make.4
$ make clean -f make.4
$ make cleanx -f make.4
```

C++ For C Coders 8

Build Process

Data Structures
C++ for C Coders

한동대학교 김영섭 교수
idebtor@gmail.com

build process
compile & link
static library
make & Makefile