

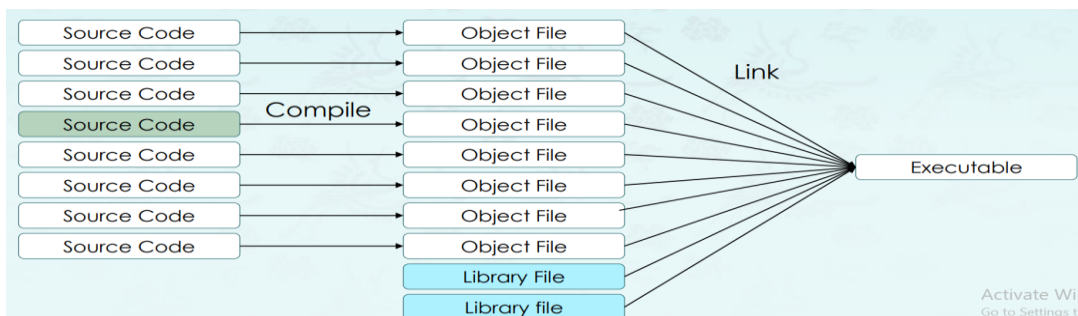
Getting Started with Visual Studio Code

VSCoDe Step3. Makefile 사용 설명서

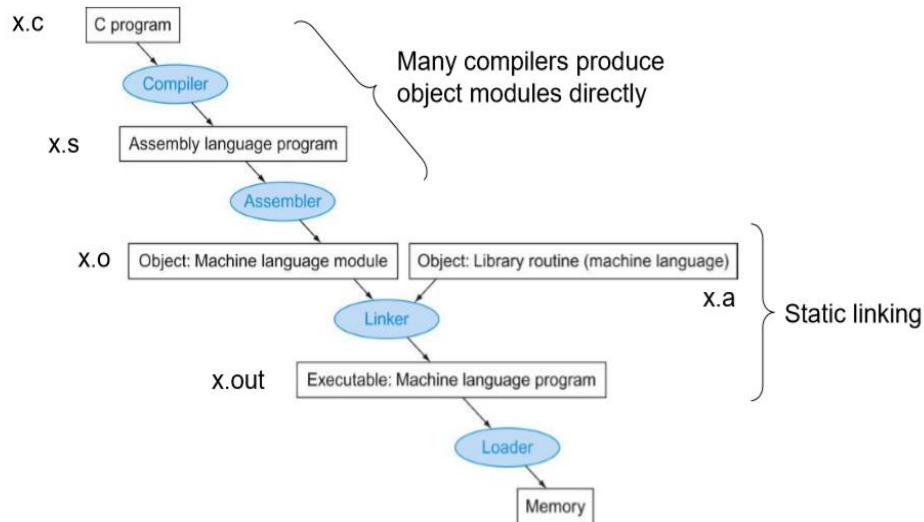
1. Build Process
2. Makefile 개념 설명
3. Makefile 환경 설정
4. Makefile 작성
5. Makefile을 통한 파일 빌드 및 제거

1. Build Process

- **Build process**란 간단히 말해 소스 코드 파일인 **.cpp**와 헤더 파일인 **.h**가 합쳐져 실행 파일인 **.exe**로 변화되는 과정을 의미합니다.
- **C++ 유형의 파일이 생성되는 방법은 아래와 같습니다.**
 - 1) C++ 프로그램은 가장 먼저 **전처리 장치(preprocessor)**에 전송됩니다. 전처리 장치를 통해 C++ 프로그램의 빌드 조건에 맞게 컴퓨터 처리가 이루어지도록 준비하며, 컴퓨터가 C++ 프로그램 언어를 처리할 수 있도록 소스 코드를 확장시켜줍니다.
 - 2) 확장된 소스 코드는 **컴파일러(compiler)**로 전송됩니다. 컴파일러를 통해 소스 코드를 어셈블리 코드로 변형시켜 주며, 이때에 **파일 확장명은 .s가 됩니다.**
 - 3) 어셈블리 코드는 **어셈블러(assembler)**에 전송되어 오브젝트 파일로 변형시켜주며, 이때에 **파일 확장명은 .o가 됩니다.** 오브젝트 파일은 기계어(machine language)로써 CPU가 직접 해독하고 실행할 수 있는 비트 단위의 컴퓨터 언어가 사용되어 컴퓨터가 코드를 쉽게 읽을 수 있게 도와줍니다. **현재 대부분 컴퓨터 언어의 컴파일러는 어셈블리 코드를 따로 생성하지 않고 바로 오브젝트 파일을 생성합니다.**
 - 4) 생성된 오브젝트 파일은 **링커(linker)**로 보내지며, 이때에 미리 준비된 또 다른 오브젝트 파일인 **라이브러리(.a)**와 함께 **정적 링킹(static linking)**이 되어 합쳐집니다. **라이브러리는 여러 개의 오브젝트 파일을 하나의 파일에 저장한 파일입니다.** 링커를 통해 여러 개의 오브젝트를 파일(cpp & h & a)을 하나로 합쳐 하나의 실행 파일을 생성합니다. 각 소스 코드마다 하나의 오브젝트 파일을 생성하며, 생성된 여러 개의 오브젝트 파일은 하나로 합쳐집니다. 이때에 **파일 확장명은 .exe가 되며, 컴퓨터가 읽고 실행 가능해지게 됩니다.**
- ✓ **정적 라이브러리(정적 링킹)**를 사용한다면 링커가 프로그램이 필요로 하는 부분만을 라이브러리에서 찾아 실행 파일에 바로 복사합니다. 하지만, 실행 파일 내에서 라이브러리 코드가 저장되기 때문에 많은 양의 메모리를 사용한다는 단점이 있습니다.



- 5) 실행 파일(exe)을 실행시킨다면 실행 파일은 **적재기(loader)**로 전송되며, 전송된 실행 파일은 컴퓨터의 외부 기억 장치에서 판독한 후에 프로그램을 실행시켜 결과 값을 사용자가 확인할 수 있도록 합니다.

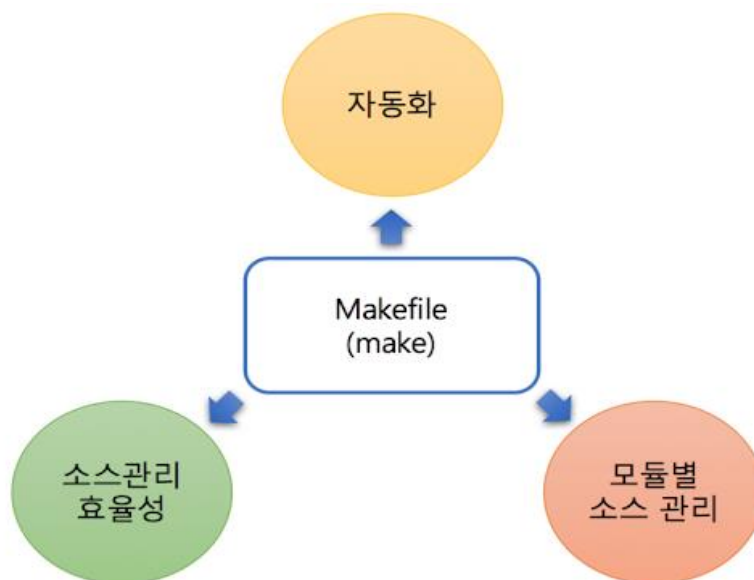


(C++ 프로그램 Build Process)

- Build process에서 가장 중요한 단계로는 소스 코드와 헤더 파일을 오브젝트 파일로 변환시켜주는 **“컴파일 단계 (.cpp & .h → .o)”**와 오브젝트 파일과 라이브러리 파일을 실행 파일로 변환시켜주는 **“링킹 단계(.o & .a → .exe)”**가 가장 중요합니다. 이때에 **Makefile**은 중요한 두 단계(컴파일과 링킹 단계)를 한번에 간편하게 실행할 수 있도록 도와주는 역할을 해주는 툴입니다.

2. Makefile 개념 설명

- 소스 코드를 컴파일하여 프로그램을 만드는 것은 복잡하고 시간이 많이 소요되는 작업입니다. 컴파일을 위한 명령어가 복잡하며, 소스 코드가 변경될 때마다 매번 수동으로 업데이트를 해주어야 하기에 번거롭습니다. 하나의 소스 코드만 변경되어도 다른 소스 코드와 함께 다시 컴파일을 해야 하기에 작업중인 프로그램 또는 프로젝트의 크기가 클수록 컴파일을 위한 시간이 많이 소요될 수 있습니다. **Makefile**은 이러한 불편함을 없애고 시간 소요를 최소화 시킬 수 있는 간편한 기술 파일입니다.
- 여러 파일 간의 종속 관계를 파악하여 **Makefile**에 기술된 대로 컴파일러에 직접 명령하여 명령이 순차적으로 진행될 수 있게 도와줍니다. 각 파일에 대한 반복적 명령의 자동화로 인한 시간 절약, 프로그램 종속 구조의 빠른 파악, 많은 양의 파일 관리 용이, 그리고 단순 반복 작업 및 재작성 최소화와 같은 장점들이 많은 유용한 툴입니다.
- **Makefile**을 작성한 후에는 Command line에 **"make" 또는 "make clean"**만을 입력함으로써 간편하게 **파일을 빌드 또는 제거**할 수 있습니다. 소스 코드에 변경이 있을 때마다 **"make"** 명령어를 사용함으로써 빠르게 프로그램을 업데이트할 수 있습니다.

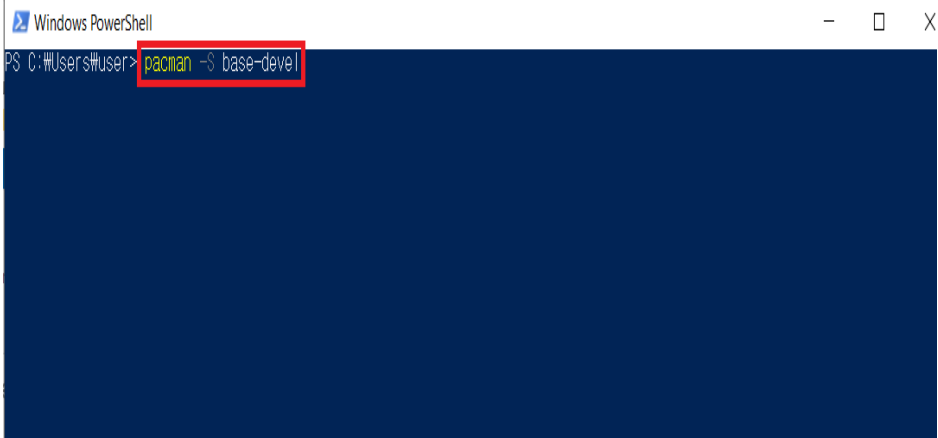


(Makefile 장점)

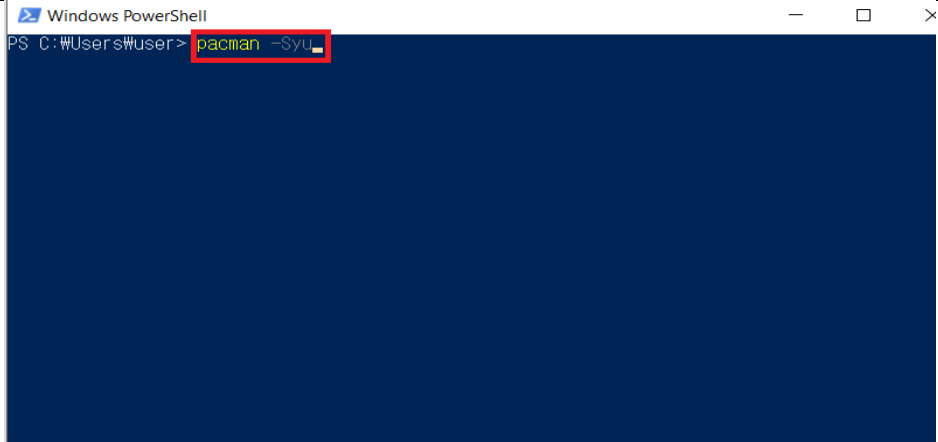
3. Makefile 환경 설정

- Makefile의 사용에 앞서 먼저 필요한 패키지들이 설치되어 있어야 합니다. (VSCode Step1의 MSYS2 업데이트 방법 참조) 콘솔 창에 아래의 명령어를 입력해줍니다.

```
pacman -S base-devel
```



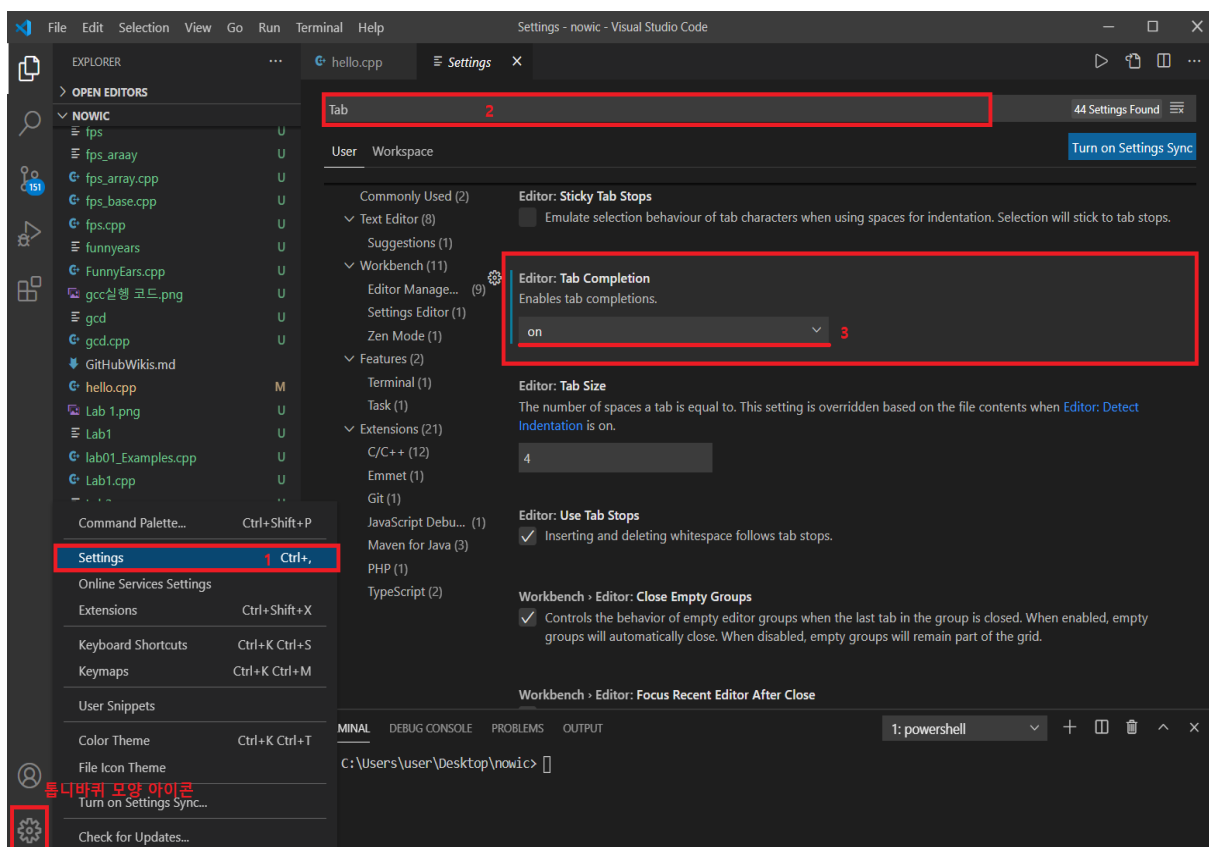
```
pacman -Syu
```



- “설치를 진행하시겠습니까? [Y/n]”라는 질문이 콘솔 창에 나오면 Y를 입력한 후에 [Enter]을 눌러줍니다. 설치 진행 중에 나오는 경고 메시지는 모두 무시하고 Y와 [Enter]을 입력하여 설치를 진행하면 됩니다.

- VSCode를 이용해 Makefile을 작성하기 위해서는 [Tab]에 대한 환경 설정을 해주어야 합니다. Makefile은 오류가 쉽게 날 수 있는 파일이기에 작성할 때에 항상 **스펠링과 띄어쓰기 사용을 주의해야 합니다. VSCode에서 기존의 Soft Tab을 Hard Tab으로 변경해주어야 합니다.**

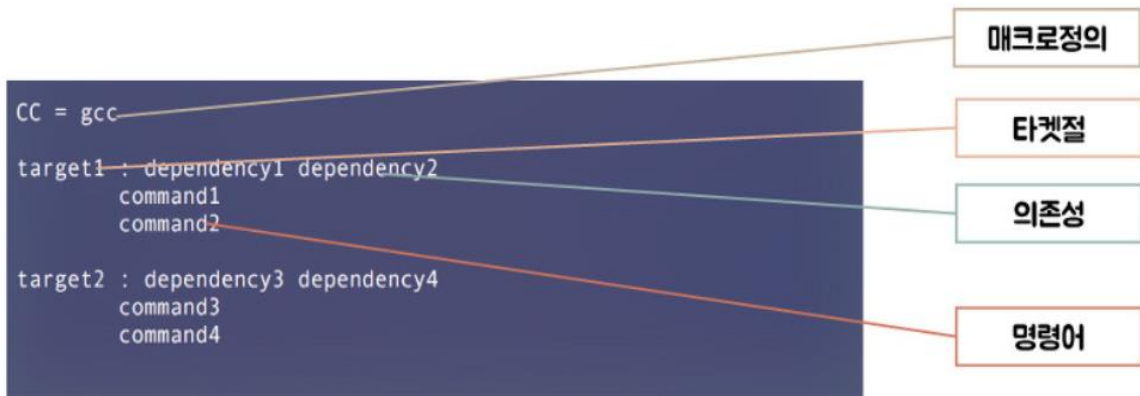
- 1) VSCode 좌측 하단의 톱니바퀴 모양 아이콘(Manage)을 클릭해준 후에 [설정] 또는 [Settings]를 선택해줍니다.
- 2) 검색 창에 “Tab”을 검색해줍니다.
- 3) “Editor: Tab Completion”을 “on”으로 변경해줍니다. (디폴트 값: off)
- 4) 환경 설정이 모두 완료되었다면 VSCode을 종료했다 다시 실행시켜줍니다.



- 만약 “makefile:2: *** missing separator. Stop”이라는 오류가 발생한다면 탭 설정이 잘못 되었거나 변경된 설정이 저장되지 않은 것입니다. Tab Completion을 off로 지정한 후에 다시 on으로 지정하고 VSCode를 종료했다 다시 실행해주어 오류를 해결할 수 있습니다.

4. Makefile 작성

- Makefile의 기본 구조는 아래와 같습니다. 이때에 **탭에서 가장 많은 오류가 발생하기에 탭에 대한 환경 설정을 확실히 해주어야 합니다.**



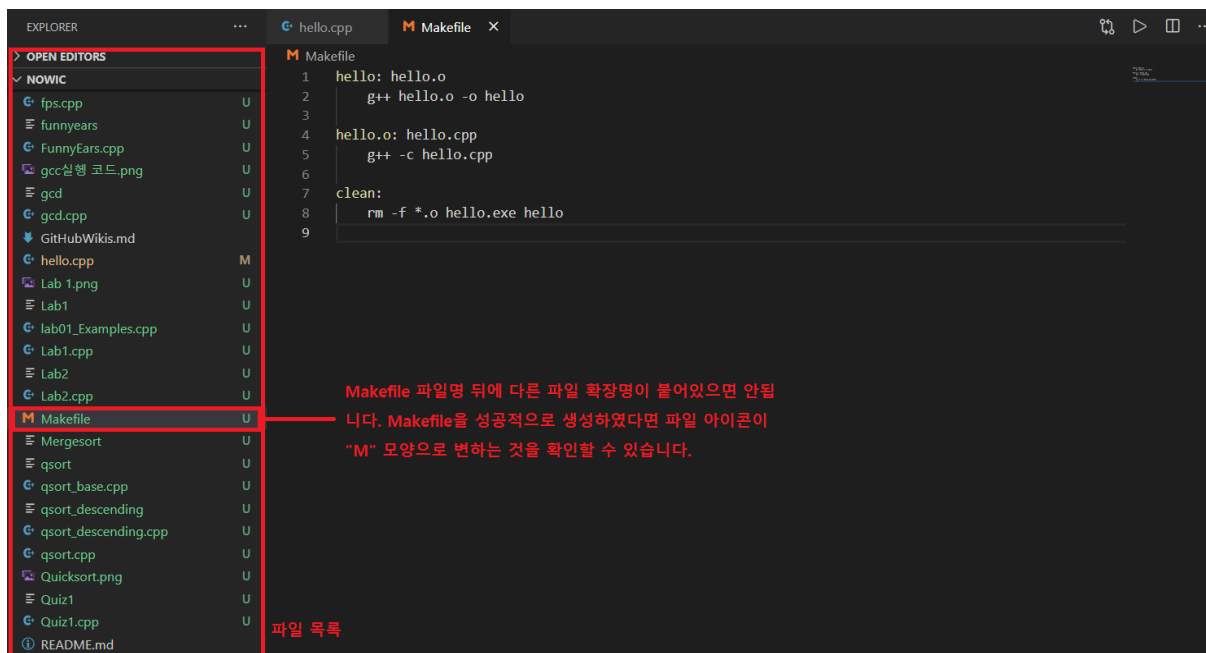
- **매크로(Macro):** 소스 코드를 단순화 시키기 위한 방법
- **타겟절(Target):** 목적 파일이라도 불리며, 명령어가 수행되어 나온 결과를 저장할 파일
- **의존성(Dependency):** 목적 파일을 생성하기 위해 필요한 재료
- **명령어(Command):** 실행되어야 할 명령어들

- 이때에 아래와 같이 작성 규칙을 지켜야 합니다. 이때에 **탭을 가장 조심해야 합니다.**

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

- 기존에 사용한 "hello.cpp"에 대한 Makefile을 생성합니다.

- 1) VSCode 상단 옵션의 [File]을 클릭한 후에 [New File]을 선택한 후에 새로운 파일을 생성해줍니다. 또는 **Ctrl + N** 단축키로 새로운 파일을 생성할 수도 있습니다. 생성할 파일명을 "**Makefile**"로 해줍니다. 이때에 **주의해야 할 점은 생성한 Makefile에 다른 파일 확장명이 붙어서는 안 되는 것입니다.** 생성한 파일 뒤에 확장명이 붙었다면 VSCode 좌측의 파일 목록에서 "**Makefile.[파일 확장명]**"을 선택하여 우클릭 한 후에 **[Rename]** 또는 **F2**를 눌러 파일 뒤의 파일 확장명을 지워줍니다. Makefile을 성공적으로 생성하였다면 파일 아이콘이 "**M**" 모양으로 변하는 것을 확인할 수 있습니다.



- 2) 생성한 Makefile에 아래의 코드를 입력해줍니다. **[Tab]**을 **주의하여** 입력해줍니다.

```

hello: hello.o
[Tab] g++ hello.o -o hello

hello.o: hello.cpp
[Tab] g++ -c hello.cpp

clean:
[Tab] rm -f *.o hello.exe hello

```

(hello.cpp 파일을 위한 Makefile)

"g++"은 cpp(C++) 파일을 위한 기본 명령어입니다. "-o" 옵션은 생성될 실행 파일 이름을 지정하는 옵션이며, "-c" 옵션은 오브젝트 파일을 생성하는 옵션입니다. 이때 -o 옵션이 필수는 아니지만, 실행 파일 생성시에 -o 옵션을 넣지 않으면 모든 실행 파일이 **a.out** 이라는 파일명을 가지게 되므로 여러 개의 실행 파일을 생성해야 할 때 효율적인 옵션입니다. 추가적으로 오브젝트 파일의 개수가 두 개 이상일 때에 파일들의 순서는 상관이 없습니다.

clean은 Makefile을 통해 생성된 오브젝트 파일과 실행 파일을 제거하기 위함입니다. "**rm**"은 "remove"의 줄임 말로써 파일 제거를 위한 명령어이며, 파일 확장명이 **".o"**인 모든 파일들과 생성한 실행 파일(**hello.exe**)을 제거해줍니다.

- 만약 여러 개의 소스 코드 또는 오브젝트 파일을 컴파일해야 할 때에는 hello.cpp 파일을 위한 Makefile처럼 작성한다면 어려움이 있을 것입니다. 컴파일해야 할 파일의 개수가 많아질수록 파일들을 Makefile에 나열하여 작성할 때에 실수가 발생할 수 있습니다. 이러한 실수를 방지하고자 Makefile은 보통 **매크로**를 사용하여 아래와 같이 작성됩니다.

```
CC = g++
CCFLAGS = -Wall -std=c++11
LD_FLAGS = -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 $@ $^ $(LD_FLAGS)

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

■ 내부 매크로 (Internal macro)

- \$@: 현재 타겟의 이름
- \$^: 현재 타겟의 종속 항목 리스트
- \$<: 현재 타겟의 첫 번째 종속
- \$%: 현재의 타겟이 라이브러리 모듈일 때에 오브젝트 파일에 대응되는 이름

■ 미리 정의된 매크로 (Pre-defined macro)

- **CCFLAGS:** g++의 옵션 설정 (-Wall: 사소한 오류까지 검출)
 - **LDLFLAGS:** 라이브러리 링크 설정
 - **LIBDIR:** 라이브러리 폴더(lib) 위치
 - **INCDIR:** 포함 폴더(include) 위치
 - **SRCS:** 사용될 cpp 파일
 - **OBJS:** 사용된 cpp 파일을 통해 생성할 오브젝트 파일
 - **TARGET:** 생성할 실행 파일과 그에 필요한 명령어
 - **.PHONY:** 포니 타겟은 실제 파일 이름을 나타내는 타겟 이름이 아닙니다. make 명령어를 실행했을 때에만 사용됩니다. 포니 타겟을 사용하는 이유로는 동일한 이름의 파일 사용 충돌을 피하기 위함이며, make을 통한 빌드의 성능 향상을 위함입니다.
- 모든 매크로는 **\$()** 방식을 통해 사용할 수 있습니다.

■ Makefile 해석

- 1) g++ 컴파일러 이용
- 2) c++의 11 버전으로 사소한 오류까지 출력하도록 g++ 컴파일러 옵션 설정
- 3) 라이브러리로(.a)는 "sort"와 "nowic" 파일 이름 사용
- 4) 라이브러리 폴더로는 현재 위치하고 있는 폴더 전에 위치하고 있는 "lib" 폴더
- 5) 포함 폴더로는 현재 위치하고 있는 폴더 전에 위치하고 있는 "include" 폴더
- 6) 현재 폴더에 존재하는 모든 cpp 유형의 파일을 재료로써 사용
- 7) 현재 폴더에 존재하는 모든 cpp 유형의 파일의 이름과 동일한 오브젝트 파일 생성
- 8) 생성할 실행 파일(타겟)이름을 "sortx"로 지정
- 9) g++ -Wall -std=c++11 -I../include -o sortx -L../lib -lsort -lnowic -lm과 동일
- 10) 포니 타겟을 통해 clean 명령문 지정 (생성한 파일 제거)

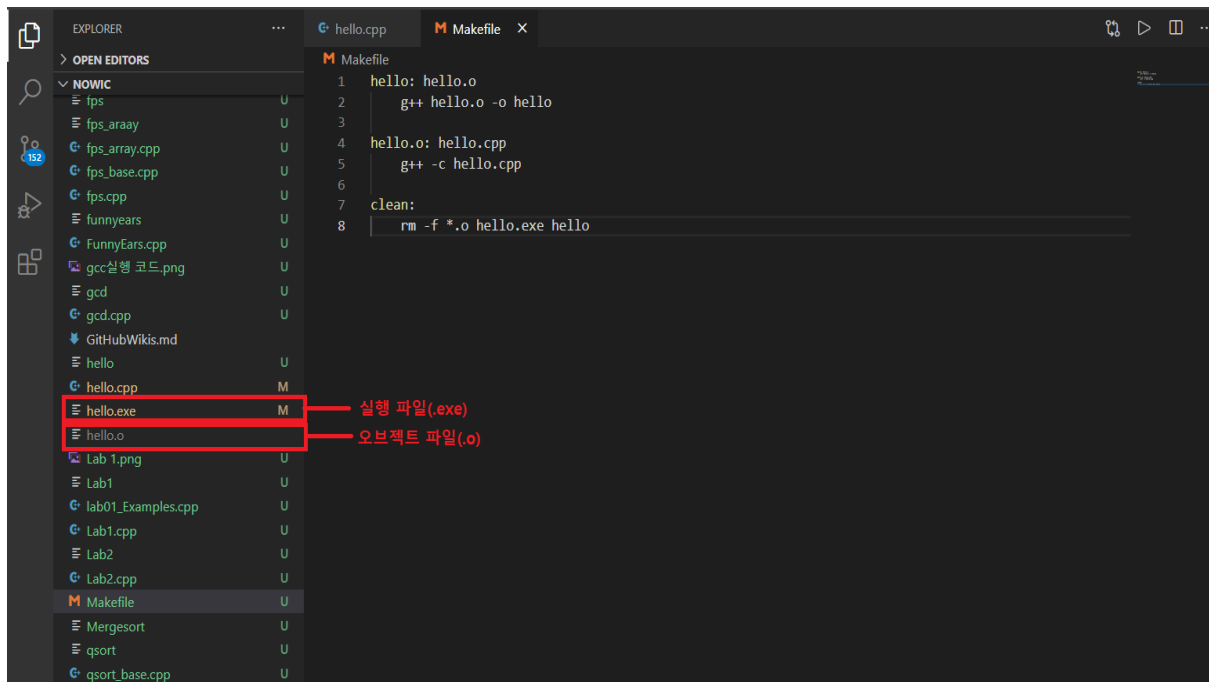
5. Makefile을 통한 파일 빌드 및 제거

■ 파일 빌드

make

VSCode의 Command line에서 “**make**”을 입력한 후에 **[Enter]**을 눌러줍니다. 이때에 먼저 현재 자신의 폴더 위치에 생성한 Makefile이 있는지 확인해줍니다. make을 통해 파일 빌드에 성공했다면 해당 폴더 안에 **오브젝트 파일(.o)**과 **실행 파일(.exe)**이 **생성되었음을 확인할 수 있습니다**. “./hello” 또는 “./[자신이 생성한 실행 파일명]”을 입력하여 실행 파일이 정상적으로 작동되는지 확인해줍니다.

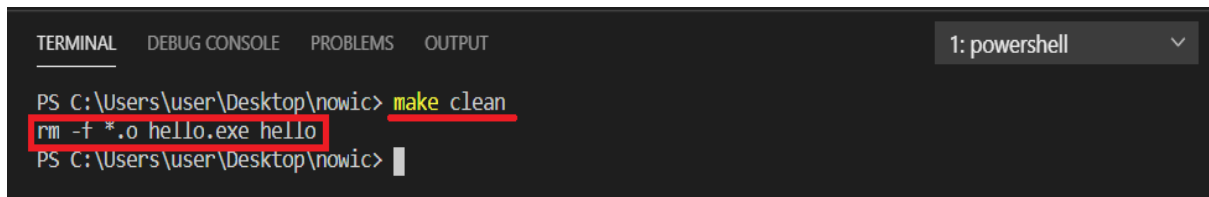
```
TERMINAL  DEBUG CONSOLE  PROBLEMS  OUTPUT  1: powershell
PS C:\Users\user\Desktop\nowic> make
g++ -c hello.cpp
g++ hello.o -o hello
PS C:\Users\user\Desktop\nowic> ./hello
Hello World!
```



■ 파일 제거

make clean

VSCode의 Command line에서 “**make clean**”을 입력한 후에 **[Enter]**을 눌러줍니다.
Makefile에 작성해놓은 **clean 명령문**이 실행되어 자신이 빌드한 **모든 오브젝트 파일(.o)**
과 실행 파일(.exe)을 모두 제거해줍니다.



The screenshot shows a VS Code terminal window with the following content:

```
TERMINAL  DEBUG CONSOLE  PROBLEMS  OUTPUT  1: powershell
PS C:\Users\user\Desktop\nowic> make clean
rm -f *.o hello.exe hello
PS C:\Users\user\Desktop\nowic> |
```

The command `make clean` is highlighted in yellow. The output `rm -f *.o hello.exe hello` is highlighted in red.