

# C++ 프로그래밍 기본

국민대학교 소프트웨어학부

# 프로그래밍 환경

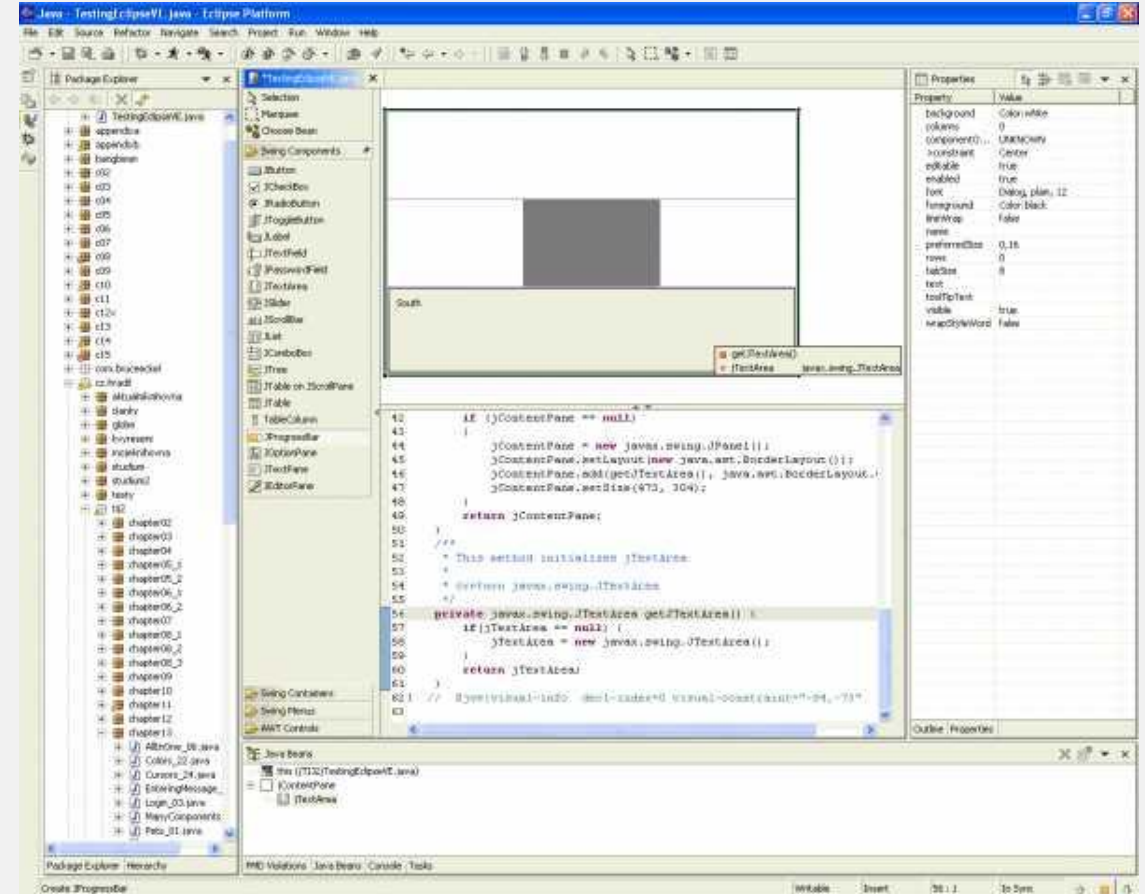
# 최근 프로그래밍 환경

- IDE (Integrated Development Environment)

- Editor
- Compiler
- Linker
- Debugger

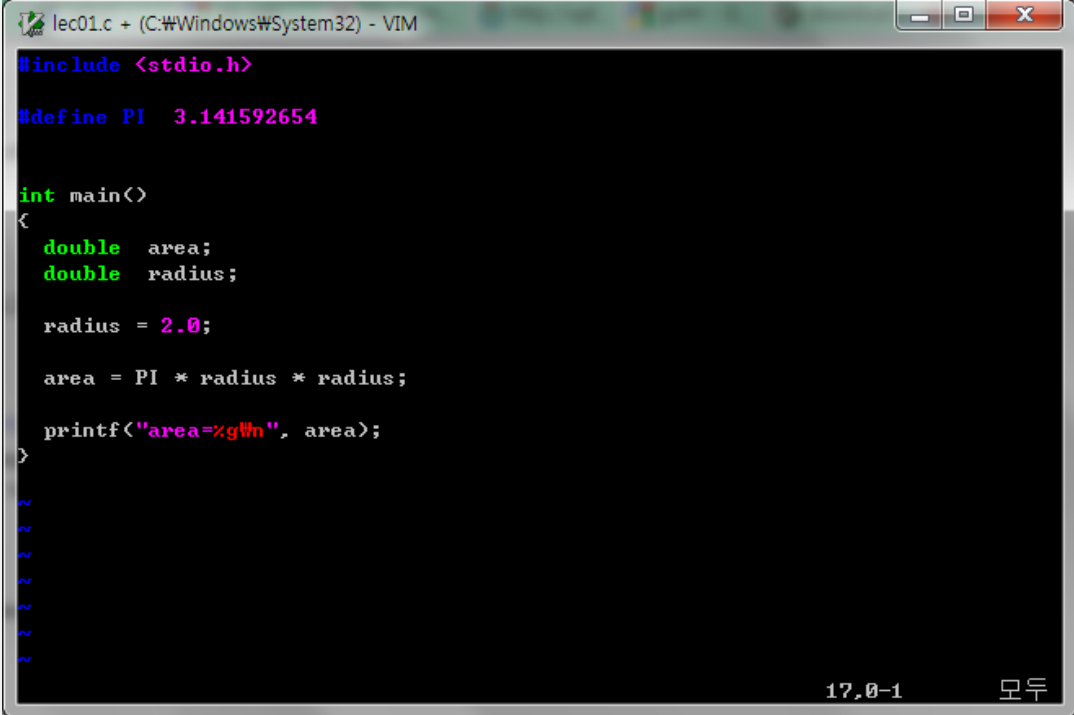
- Examples

- Visual Studio
- Eclipse
- etc



# 전통적인 프로그래밍 환경

- Editor
  - vi, emacs, notepad, etc.
- Compiler
  - cl, gcc, cc, etc.
- Linker
  - link, ld, etc.
- Debugger
  - gdb, ddd, etc.



The screenshot shows a VIM editor window titled 'lec01.c + (C:\Windows\System32) - VIM'. The code is a C program that calculates the area of a circle. It includes the standard input/output header, defines the constant PI, and implements a main function that sets a radius of 2.0, calculates the area using the formula  $area = PI * radius * radius$ , and prints the result. The code is color-coded: keywords are green, constants are magenta, and string literals are red. The status bar at the bottom right shows '17,0-1' and '모두'.

```
#include <stdio.h>

#define PI  3.141592654

int main()
{
    double  area;
    double  radius;

    radius = 2.0;

    area = PI * radius * radius;

    printf("area=%g\n", area);
}
```

# Linux에서의 전통적인 프로그래밍 환경: GNU toolchain

- Editor
  - vi (or vim), sublime text, 지에디트
- C/C++ compiler
  - [GNU Compiler Collection](#) (gcc)
- Linker
  - [GNU ld](#) (ld)
- Make
  - [GNU make](#) (make)

# 프로그램 생성과정 다시보기

- Linux에서의 전통적 프로그래밍 환경을 중심으로

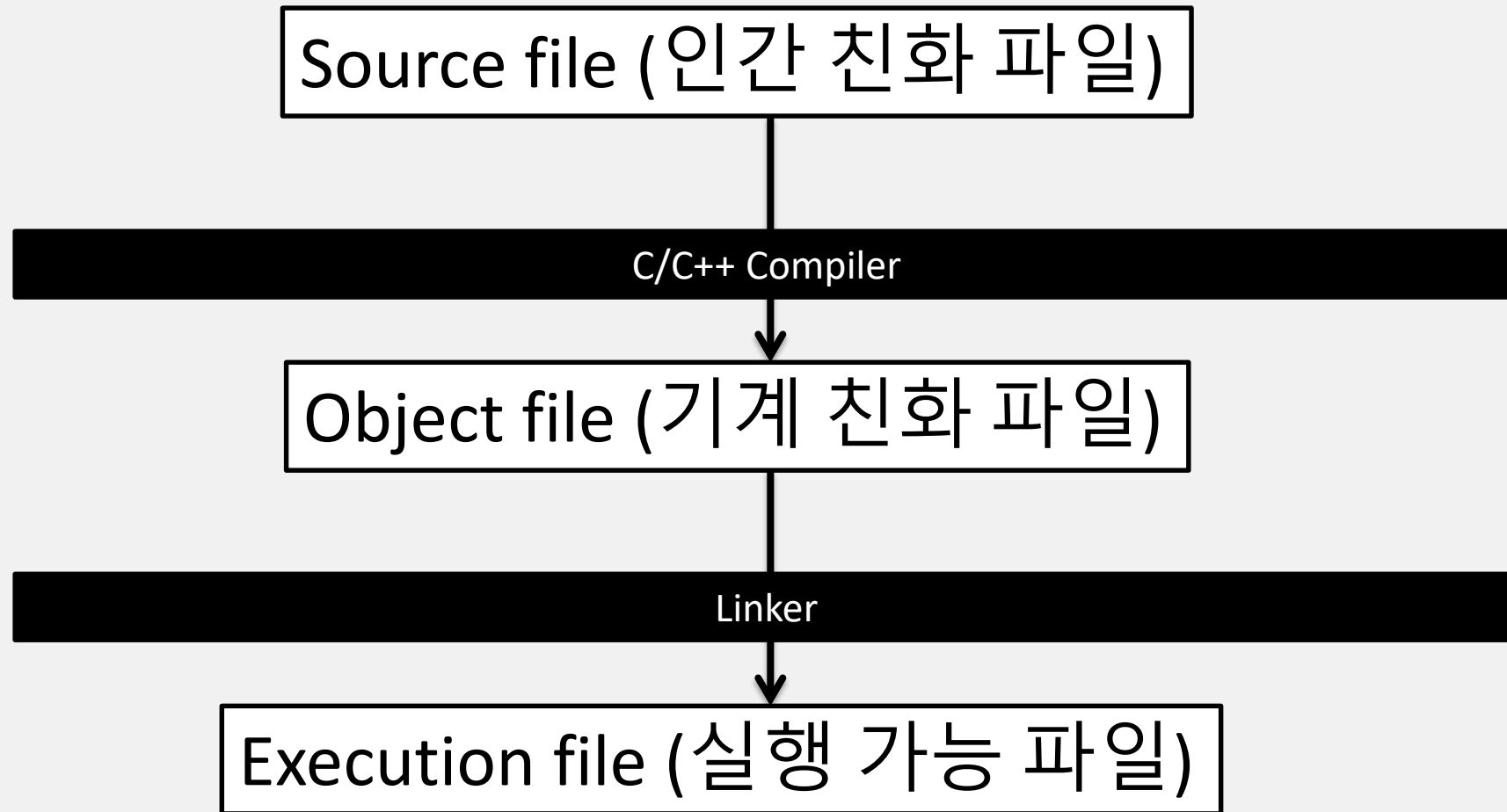
# 개발환경 설치하기

- Ubuntu 개발 필수 환경 설치
  - 터미널 환경에서 아래 명령 실행

```
> sudo apt install build-essential
```

- C/C++ 컴파일러 등, 기본 개발환경이 설치됨
- 텍스트 에디터 설치
  - 터미널 환경(CUI)을 선호한다면
    - [vi](#), [emacs](#), [nano](#)
  - 그래픽 환경(GUI)을 선호한다면
    - [Sublime Text](#), [Visual Studio Code](#), [Atom](#)

# 프로그램 생성과정





# HelloWorld 예제

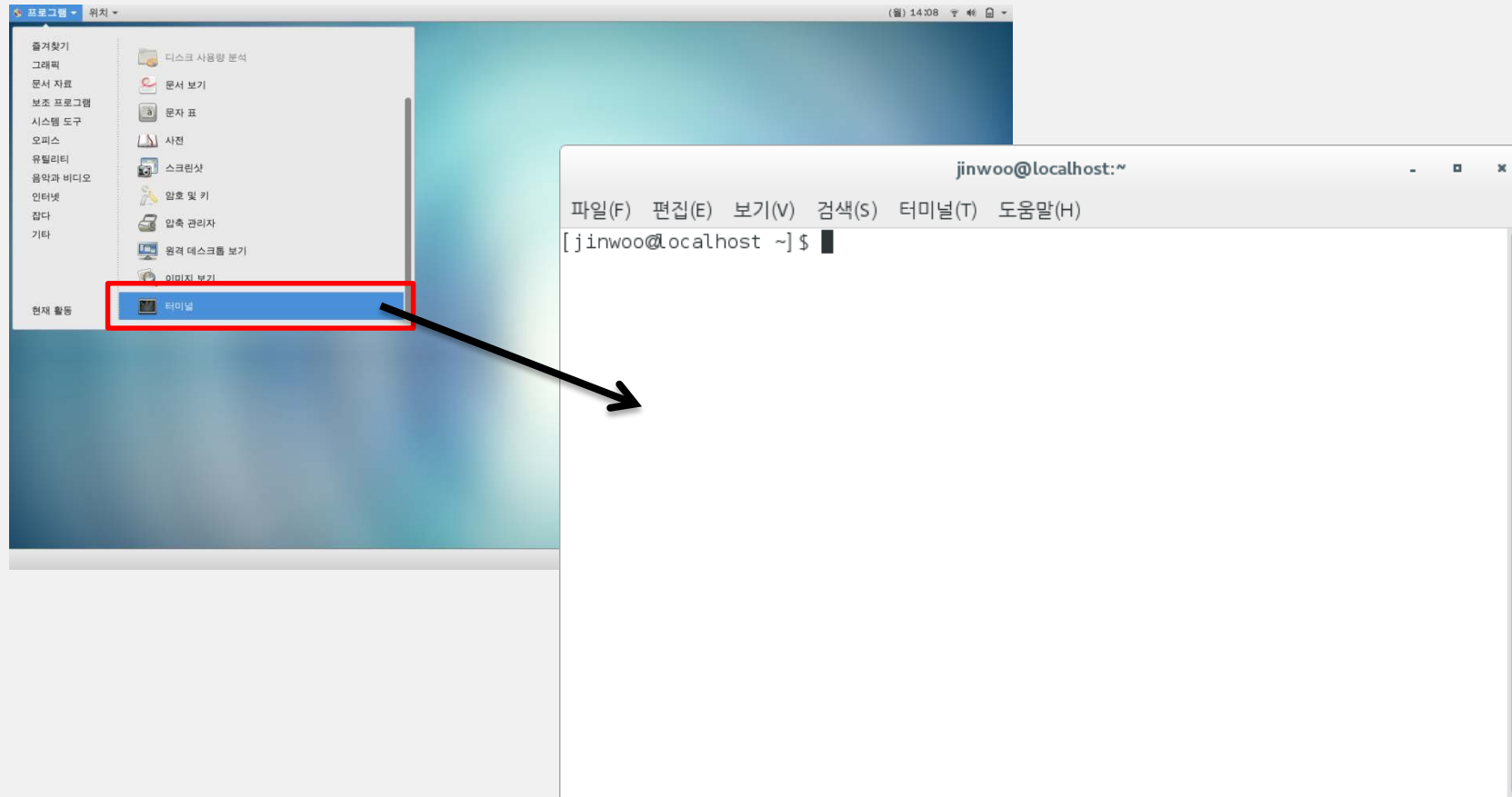
## HelloWorld.cpp

(C++ source file)

```
#include <iostream>
int main()
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

# HelloWorld 예제를 짜보자!

- 터미널 띄우기



# HelloWorld 예제를 짜보자!

- 프로그램을 생성할 디렉토리 만들기

- 1) 프로그램을 작성할 디렉토리 생성

```
> mkdir HelloWorld
```

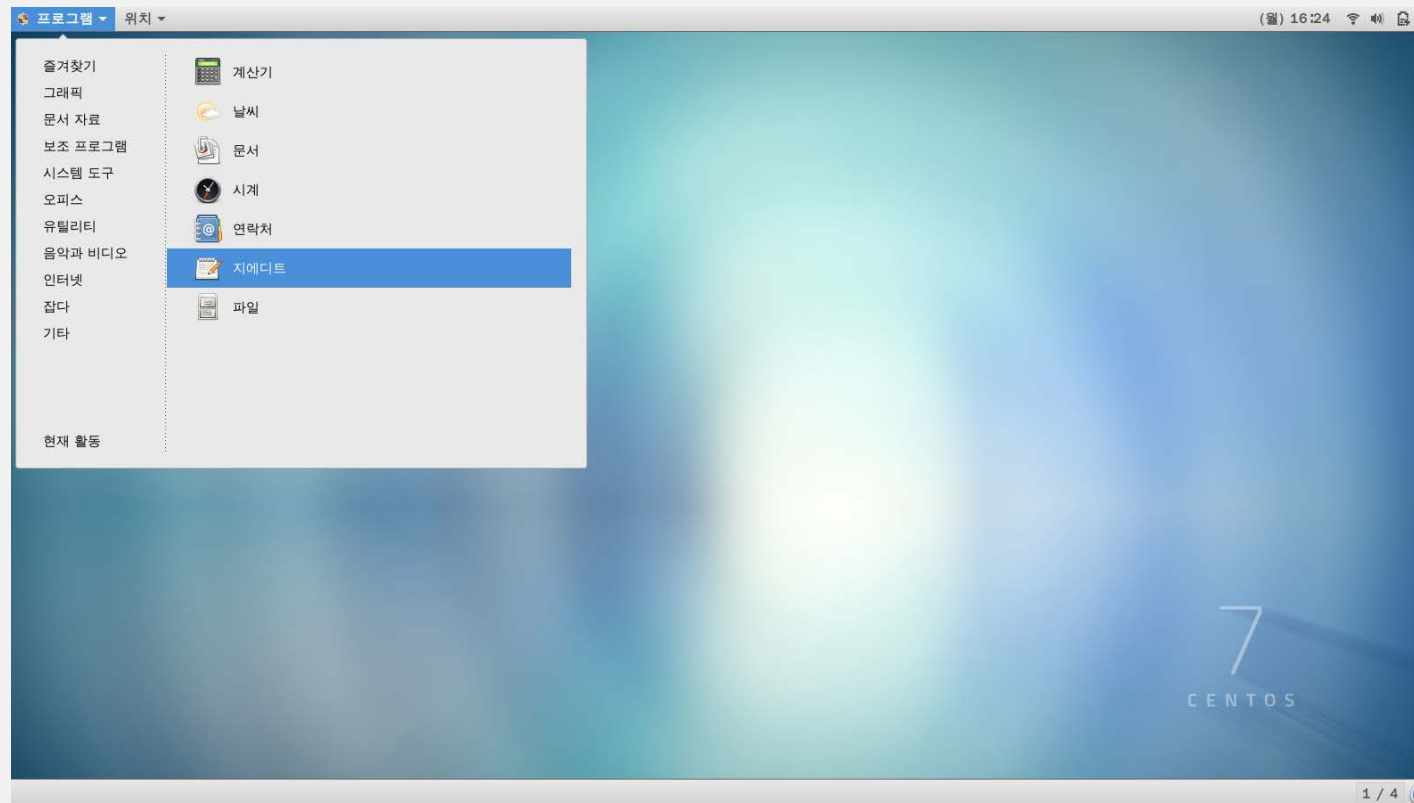
- 2) 생성된 디렉토리로 이동

```
> cd HelloWorld
```

(\* 탐색기로 홈 폴더 아래에 HelloWorld 폴더가 생성되어 있는지 확인해 보자)

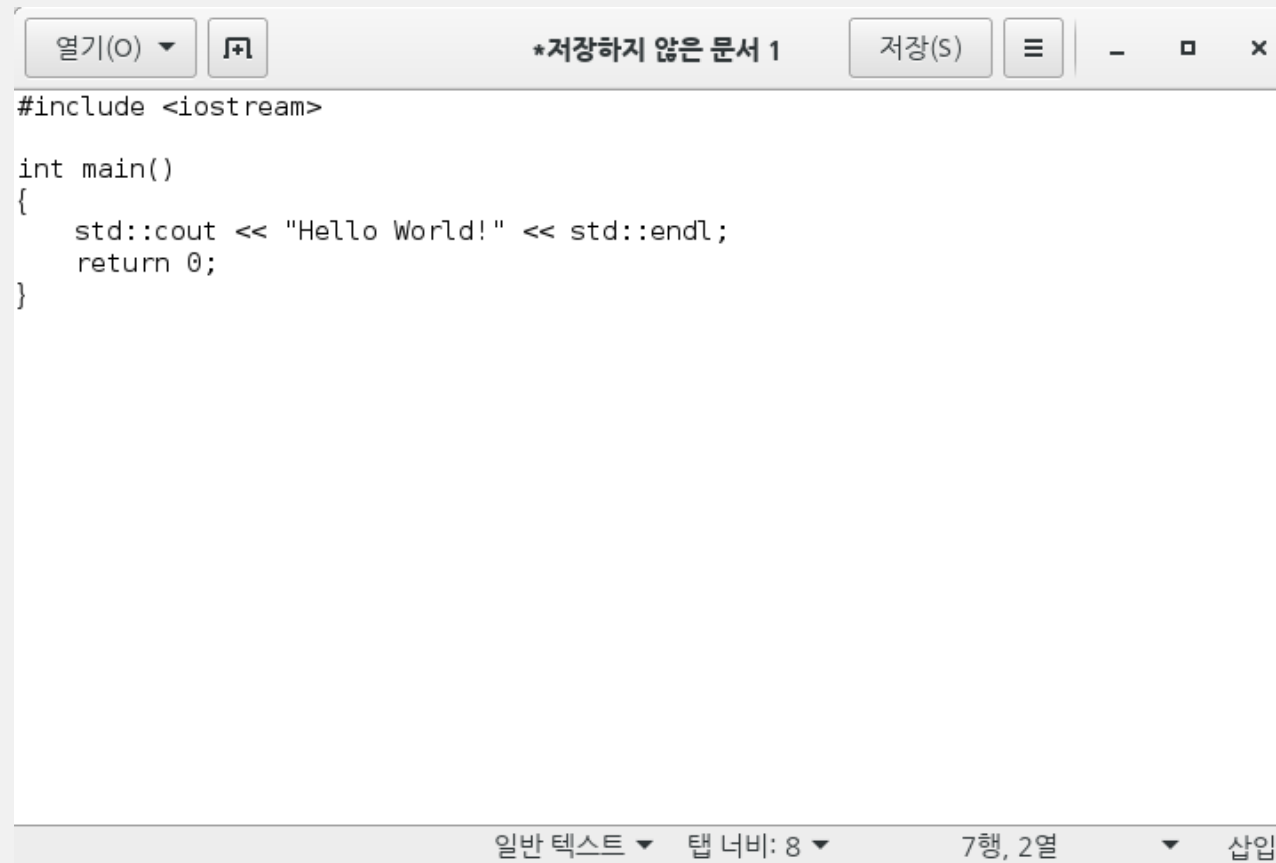
# HelloWorld 예제를 짜보자!

- 프로그램 파일 작성하기
  - 1) 소스코드 작성을 위한 에디터 실행
    - 좌측 상단의 프로그램 - 보조 프로그램 - 지에디트



# HelloWorld 예제를 짜보자!

- 프로그램 파일 작성하기
  - 2) 소스코드 작성



The image shows a screenshot of a code editor window. The title bar at the top contains the text "열기(O)" with a dropdown arrow, a file icon, "\*저장하지 않은 문서 1", a "저장(S)" button, a menu icon, and window control buttons (minimize, maximize, close). The main area of the window contains the following C++ code:

```
#include <iostream>

int main()
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

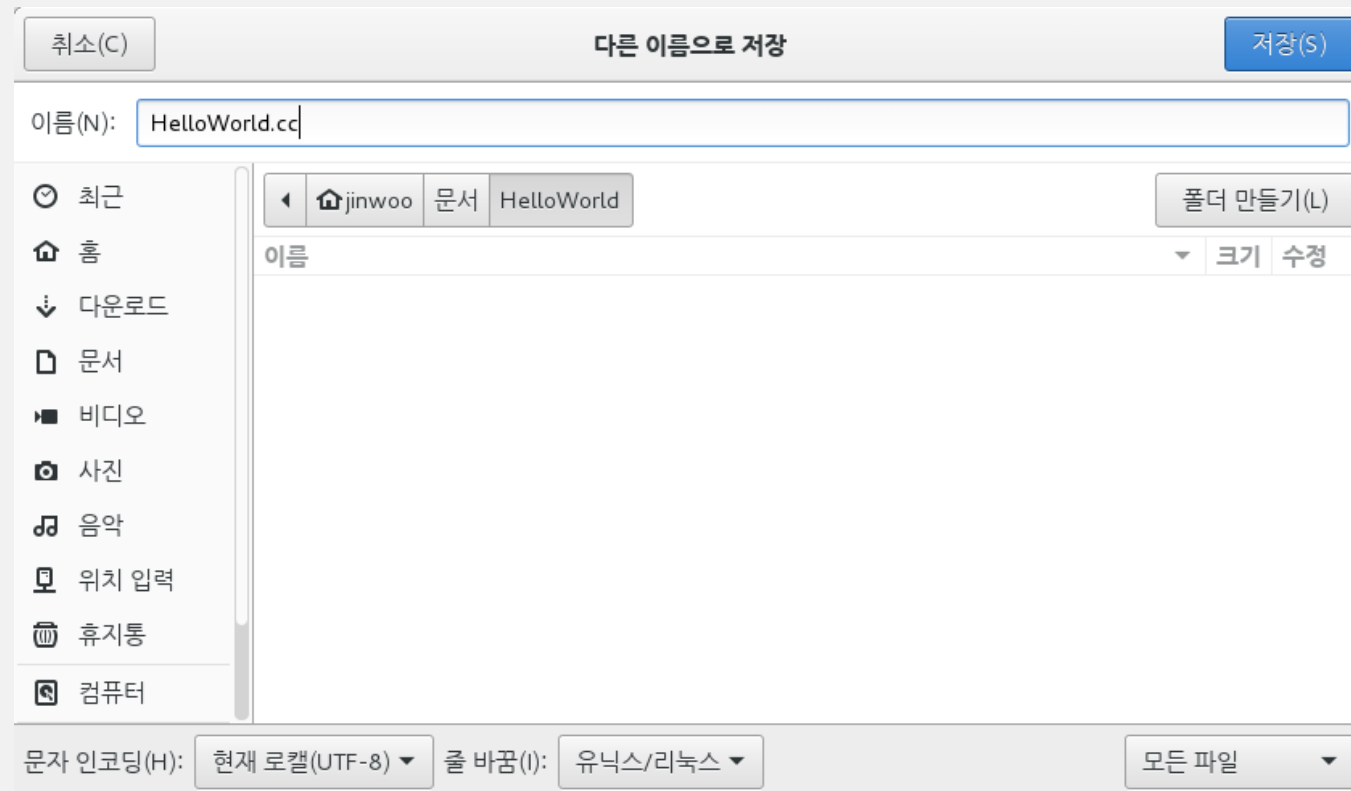
The status bar at the bottom of the window displays "일반 텍스트" with a dropdown arrow, "탭 너비: 8" with a dropdown arrow, "7행, 2열" with a dropdown arrow, and "삽입" with a dropdown arrow.

# HelloWorld 예제를 짜보자!

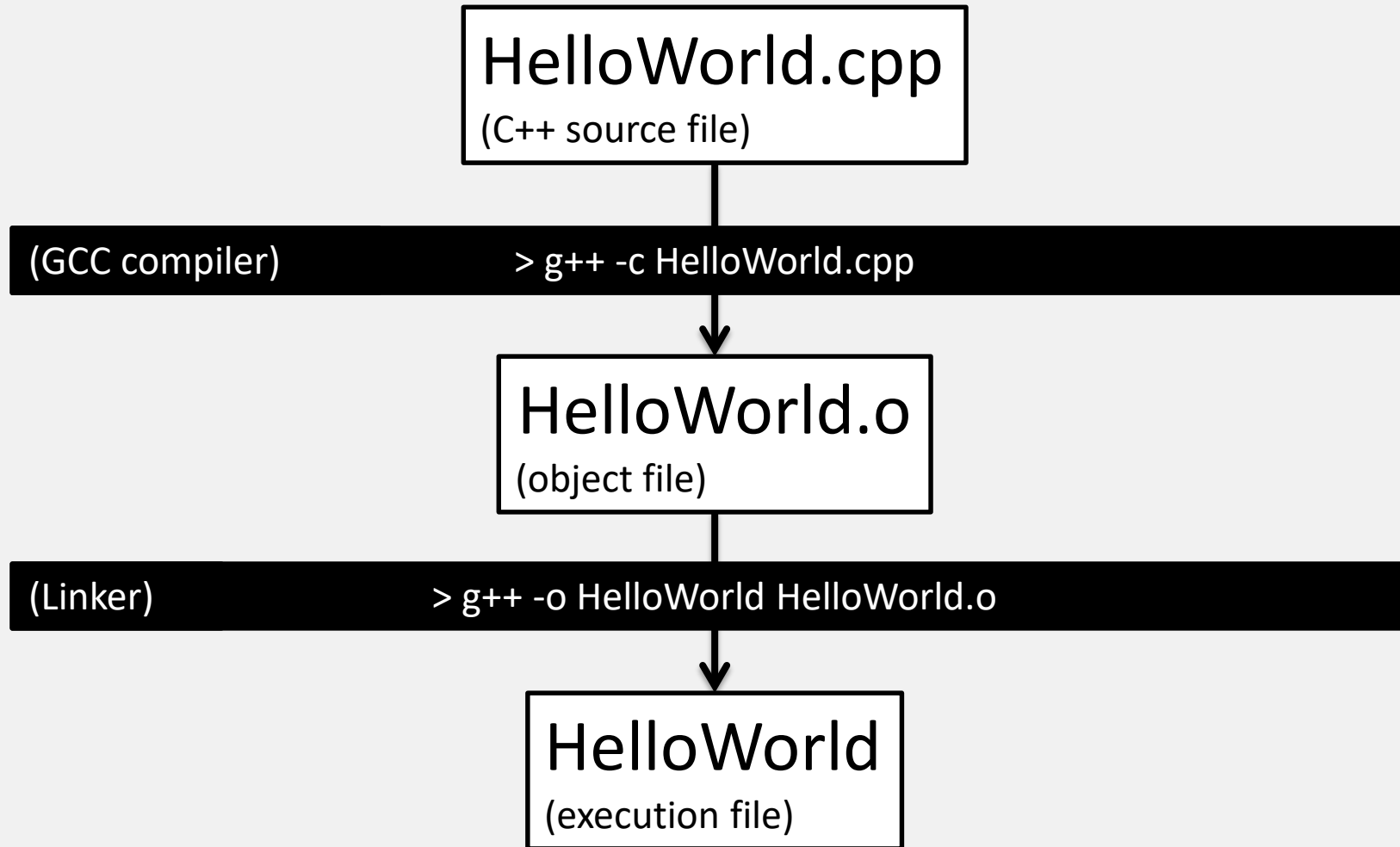
- 프로그램 파일 작성하기

- 3) HelloWorld.cpp 파일 저장

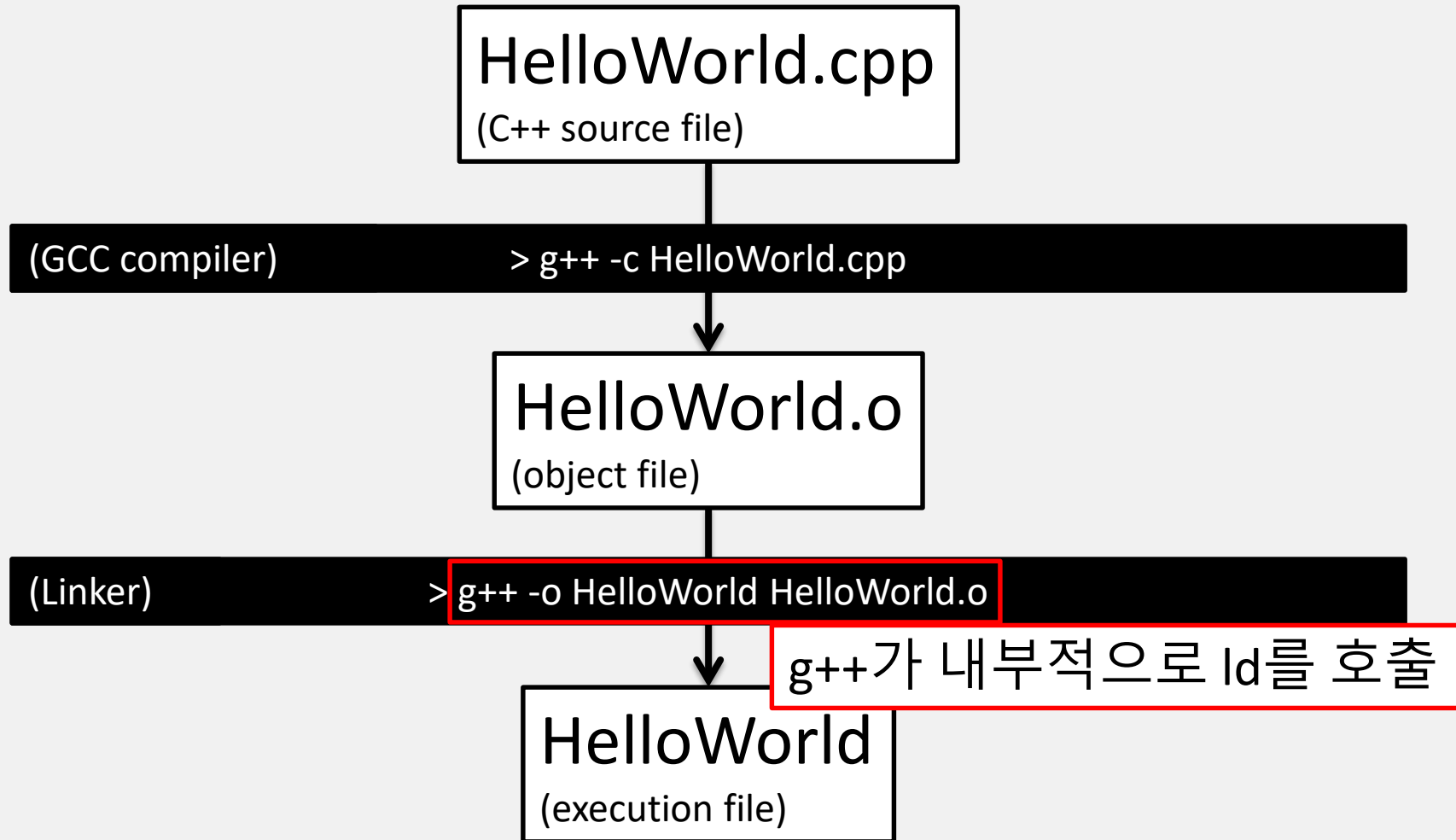
(\* 탐색기로 HelloWorld 폴더 안에 HelloWorld.cpp 파일이 생성되어 있는지 확인해 보자)



# HelloWorld 프로그램 만들기



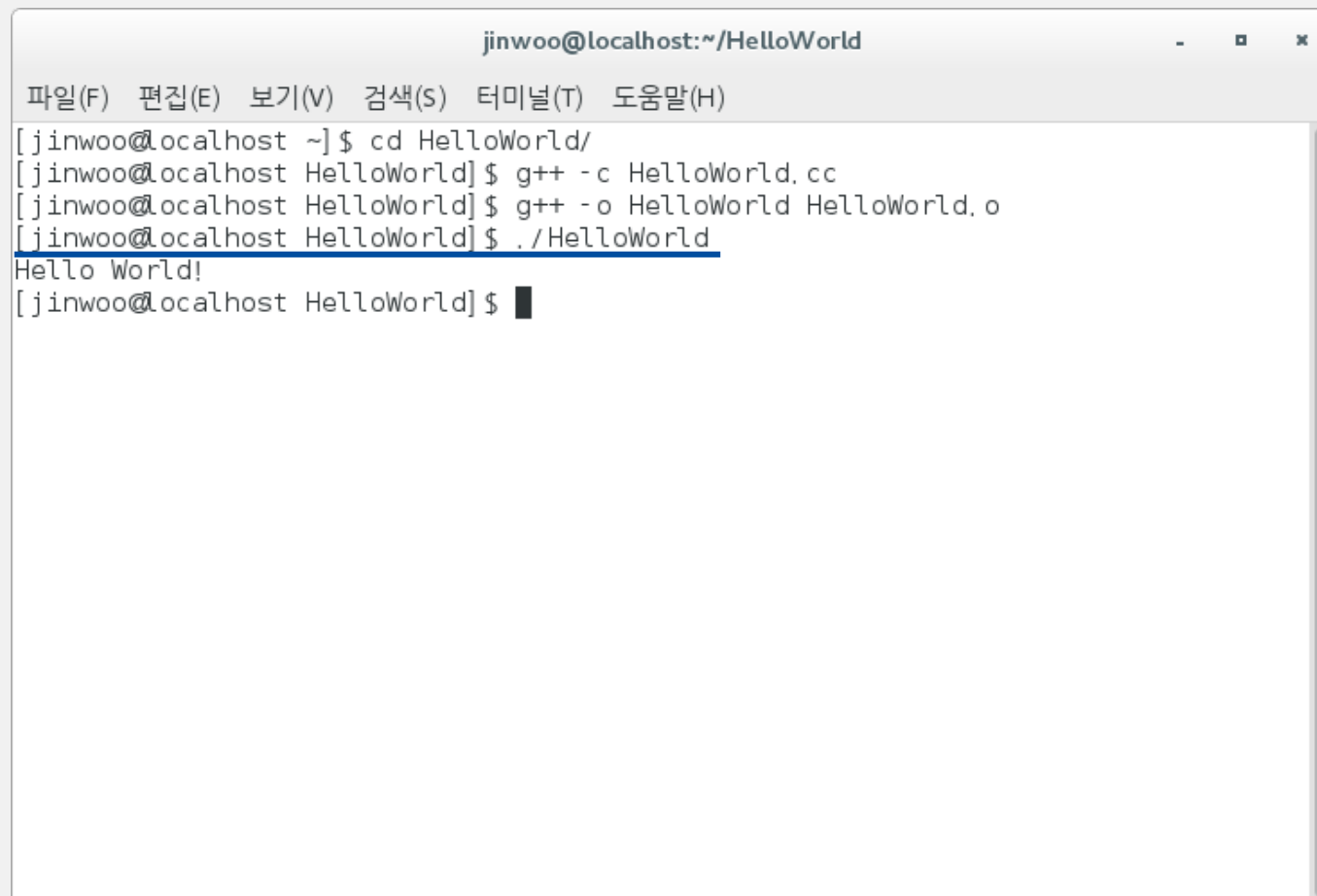
# HelloWorld 프로그램 만들기





# HelloWorld 프로그램 실행하기

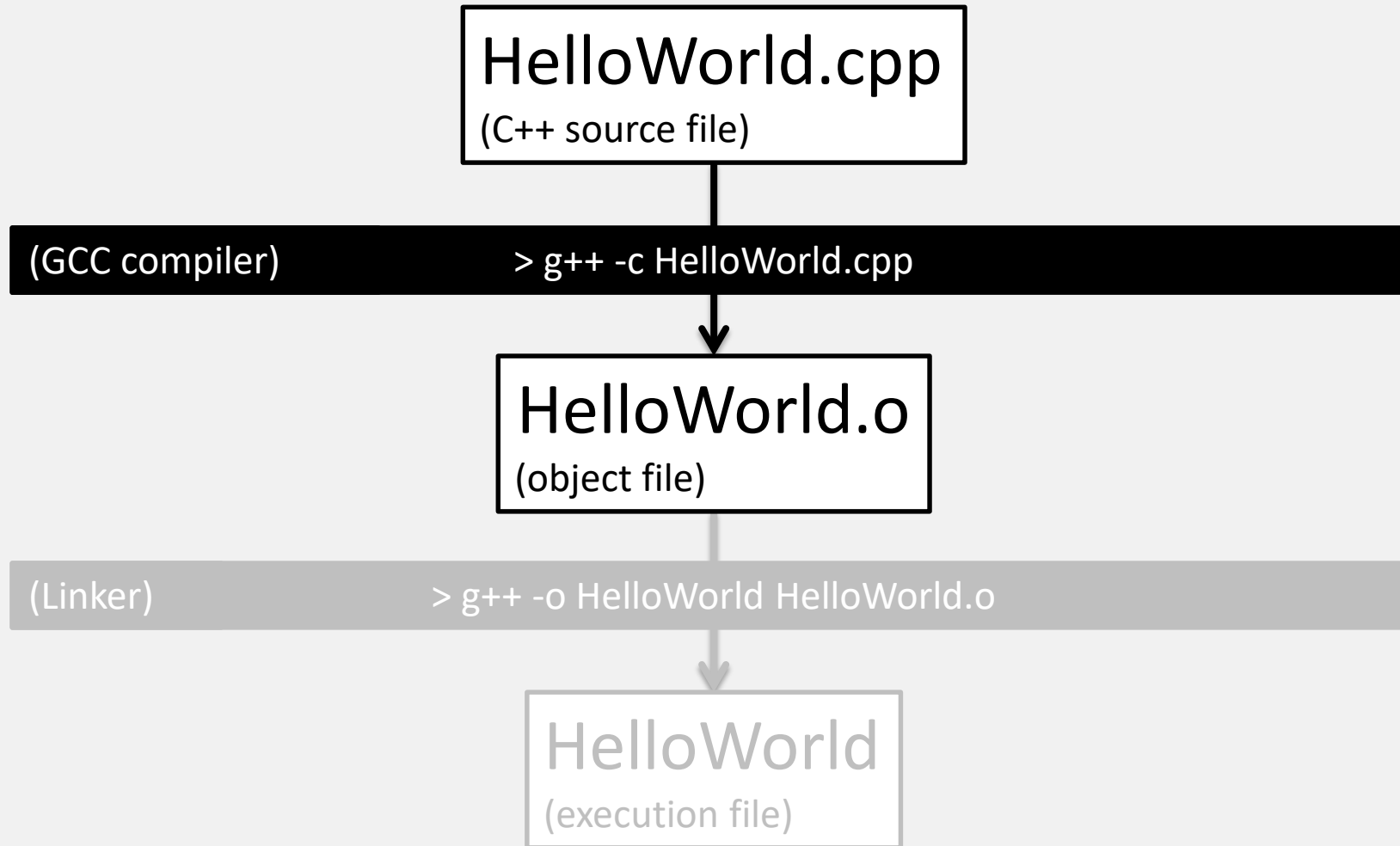
- ./HelloWorld



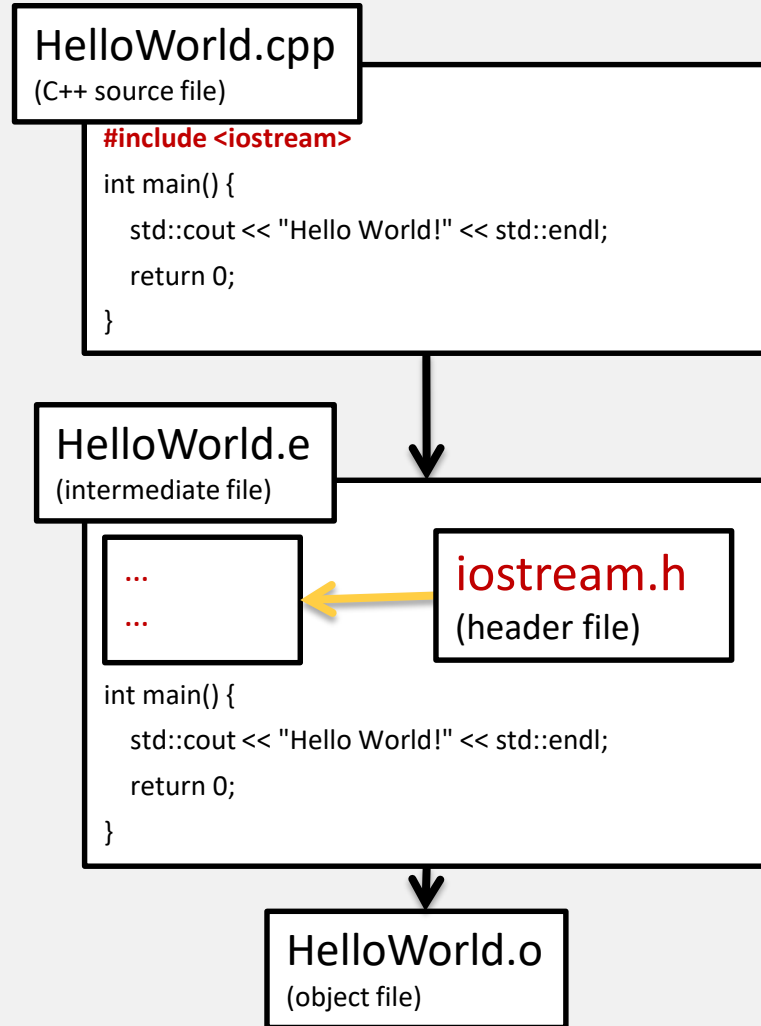
```
jinwoo@localhost:~/HelloWorld
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)
[jinwoo@localhost ~]$ cd HelloWorld/
[jinwoo@localhost HelloWorld]$ g++ -c HelloWorld.cc
[jinwoo@localhost HelloWorld]$ g++ -o HelloWorld HelloWorld.o
[jinwoo@localhost HelloWorld]$ ./HelloWorld
Hello World!
[jinwoo@localhost HelloWorld]$
```

# 컴파일 과정 다시보기

# 컴파일러의 기본적인 역할



# 컴파일러가 내부적으로 하는 일



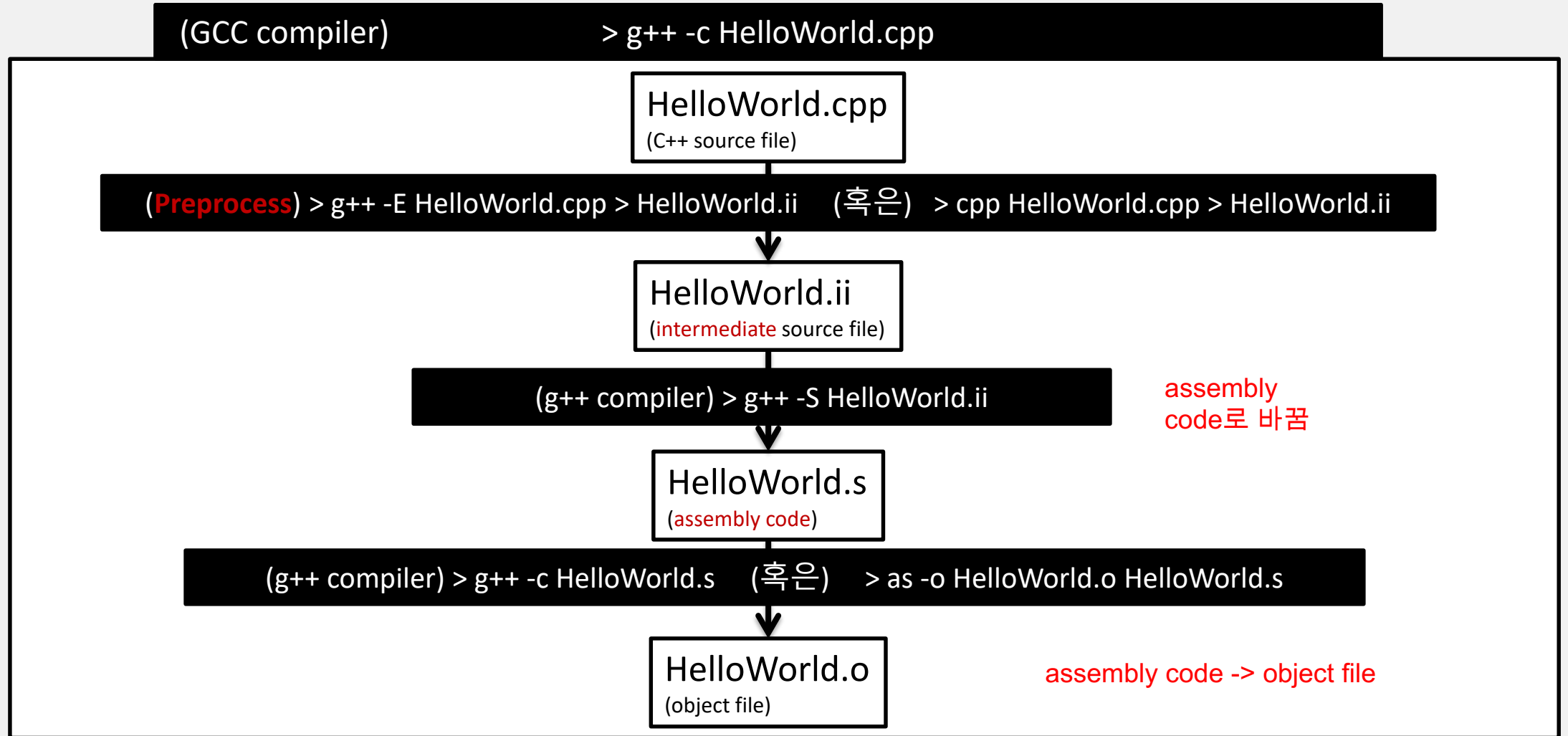
- 1) 전처리기(preprocessor)를 처리하여 임시파일 (intermediate file)을 생성 #으로 되어있는것

HelloWorld.cpp → HelloWorld.ii

- 2) 임시파일을 컴파일하여 목적파일 생성

HelloWorld.ii → HelloWorld.o

# 컴파일러가 내부적으로 하는 일



# 생각해 볼 문제

- HelloWorld.cpp 파일과 HelloWorld.e 파일을 열어 서로 비교해 보자.
- `#include <iostream>`에서 지정한 `iostream.h` 파일은 어디에 있는가?
- 다음의 `#include` 문은 서로 어떻게 다른가?
  - `#include <iostream>`
  - `#include "circle.h"`

# 컴파일러가 하는 일

- #으로 시작하는 전처리를 다룬다
  - #include 의 경우는 이미 살펴 봤음
- **컴파일러는 문법체크만 한다!**
  - 다음 프로그램은 컴파일 에러가 발생할까?

```
#include <iostream>

void my_func(int i);

void main()
{
    my_func(3);
}
```

# Make를 활용한 프로그래밍



# 왜 Make가 필요한가?

- 프로그램에 에러가 있는 경우
  - 에러를 수정하고 컴파일, 링크를 수행
- 매번 컴파일, 링크를 위해 동일한 명령을 타이핑하기 귀찮음

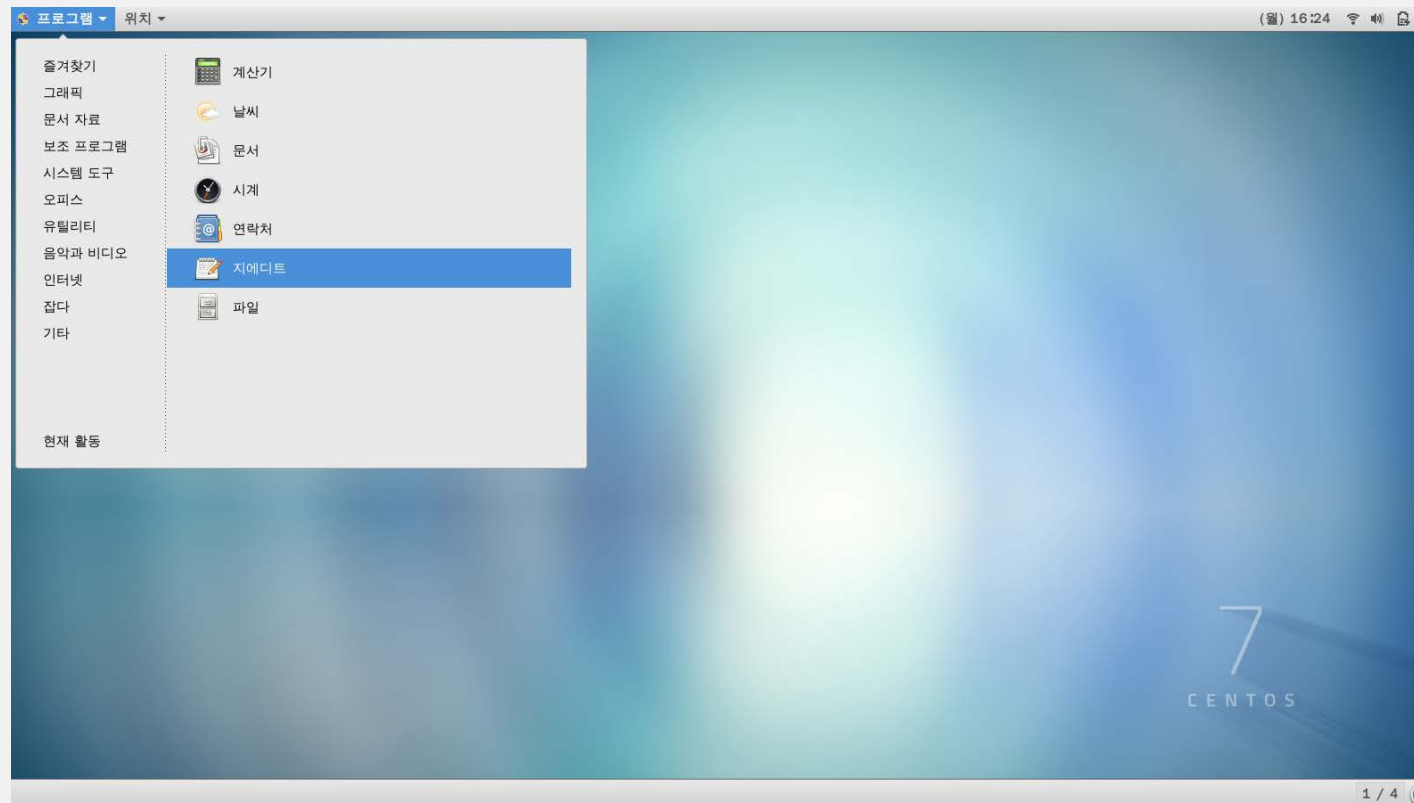
```
> g++ -c HelloWorld.cpp  
> g++ -o HelloWorld HelloWorld.o
```

- Make를 쓰게 되면 간단해짐

```
> make
```

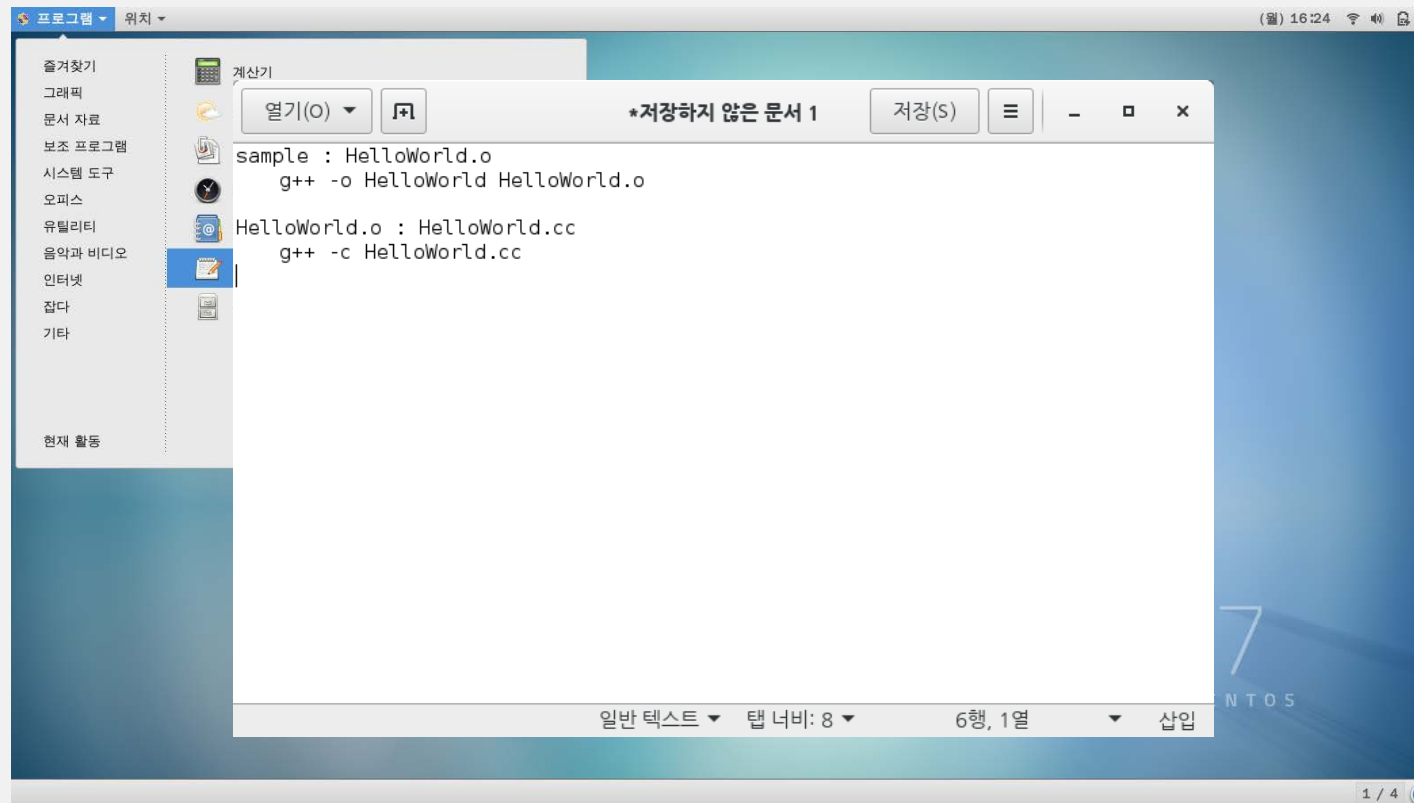
# Make 사용하기

- Makefile 작성
  - 매번 수행될 컴파일, 링크의 정보가 담긴 파일 작성
    - 좌측 상단의 프로그램 - 보조 프로그램 - 지에디트



# Make 사용하기

- Makefile 작성
  - 아래와 같이 Makefile을 작성



The screenshot shows a Windows Notepad window titled "\*저장하지 않은 문서 1" (Untitled - 1). The window contains a Makefile with the following content:

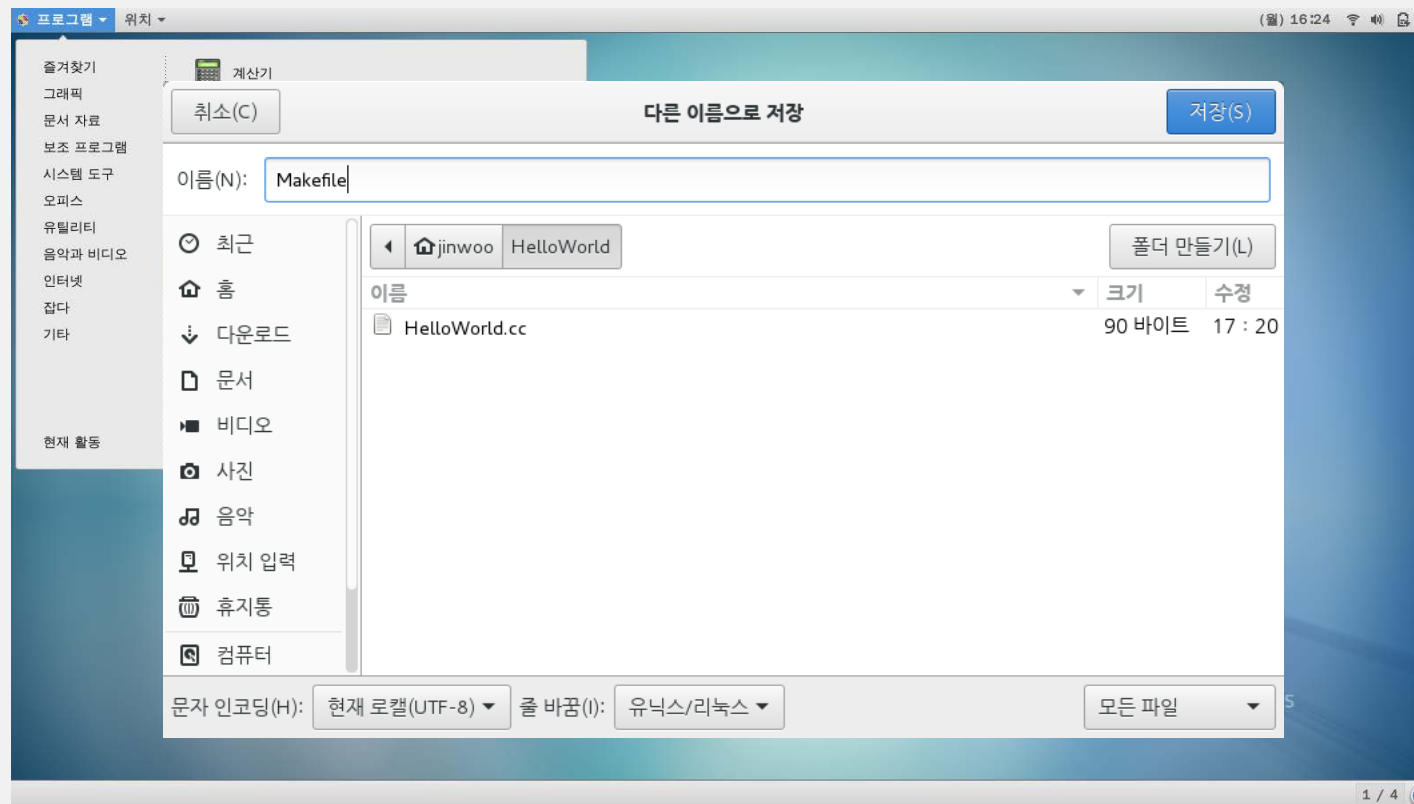
```
sample : HelloWorld.o
g++ -o HelloWorld HelloWorld.o

HelloWorld.o : HelloWorld.cc
g++ -c HelloWorld.cc
```

The window's status bar at the bottom indicates "일반 텍스트" (Plain Text), "탭 너비: 8" (Tab width: 8), "6행, 1열" (6 lines, 1 column), and "삽입" (Insert). The Windows taskbar at the bottom shows the time as 16:24 and the date as (월) 16:24.

# Make 사용하기

- Makefile 작성
  - 소스 코드와 동일한 폴더에 'Makefile' 파일명으로 저장



# Make 사용하기

- Makefile 작성
  - 매번 수행될 컴파일, 링크의 정보가 담긴 파일 작성

sample : 컴파일 -> object code ->  
실행 code (link)

```
sample : HelloWorld.o
    g++ -o HelloWorld HelloWorld.o

HelloWorld.o : HelloWorld.cpp
    g++ -c HelloWorld.cpp
```

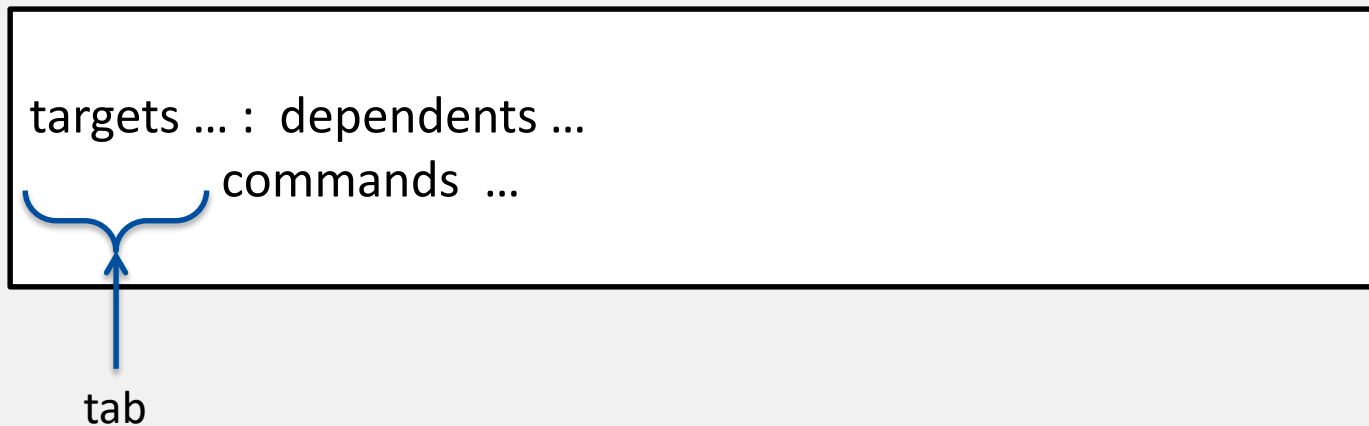
make를 하면 HelloWorld.o 먼저 실행. 목적 파일로 바뀌기 위해선 미리 컴파일 된 파일이 필요하기 때문에 HelloWorld.o이 수행된 후 sample이 수행됨.

- make 수행
  - 작성된 Makefile을 자동으로 읽어서 처리함

```
> make
```

# Makefile의 구조

- 여러 개의 규칙(rule)들의 나열
  - 각 규칙은 목표(target), 의존 관계(dependent), 명령(command)로 구성됨
  - 명령 부분은 반드시 tab 글자로 시작함



# Makefile의 구조

- 오른쪽 Makefile은 2개의 목표로 구성되어 있음
  - [목표 1] sample
    - 목표 1은 HelloWorld.o 파일이 존재해야 실행되며, (dependents)  
만일 HelloWorld.o파일이 있으면  
g++ -o HelloWorld HelloWorld.o 를 수행함 (commands)
  - [목표 2] obj
    - 목표2는 HelloWorld.cpp 파일이 존재해야 실행되며, (dependents)  
만일 HelloWorld.cpp 파일이 있으면  
g++ -c HelloWorld.cpp를 수행함 (commands)

## Makefile

```
sample : HelloWorld.o
    g++ -o HelloWorld HelloWorld.o

HelloWorld.o : HelloWorld.cpp
    g++ -c HelloWorld.cpp
```

# Makefile의 구조

- 목표들 간의 의존관계로부터 최종적인 산출물이 결정됨
- 일반적으로 아래와 같이 이해할 수 있음
  - 의존관계를 목표의 입력
  - 명령은 목표의 출력을 만드는 과정

## Makefile

```
sample : HelloWorld.o
    g++ -o HelloWorld HelloWorld.o

HelloWorld.o : HelloWorld.cpp
    g++ -c HelloWorld.cpp
```

목표: exe(execution file)

입력(의존관계):  
HelloWorld.o

명령:  
g++ -o HelloWorld HelloWorld.o

출력:  
HelloWorld

목표들 간의 의존관계

목표: o(object file)

입력(의존관계):  
HelloWorld.cpp

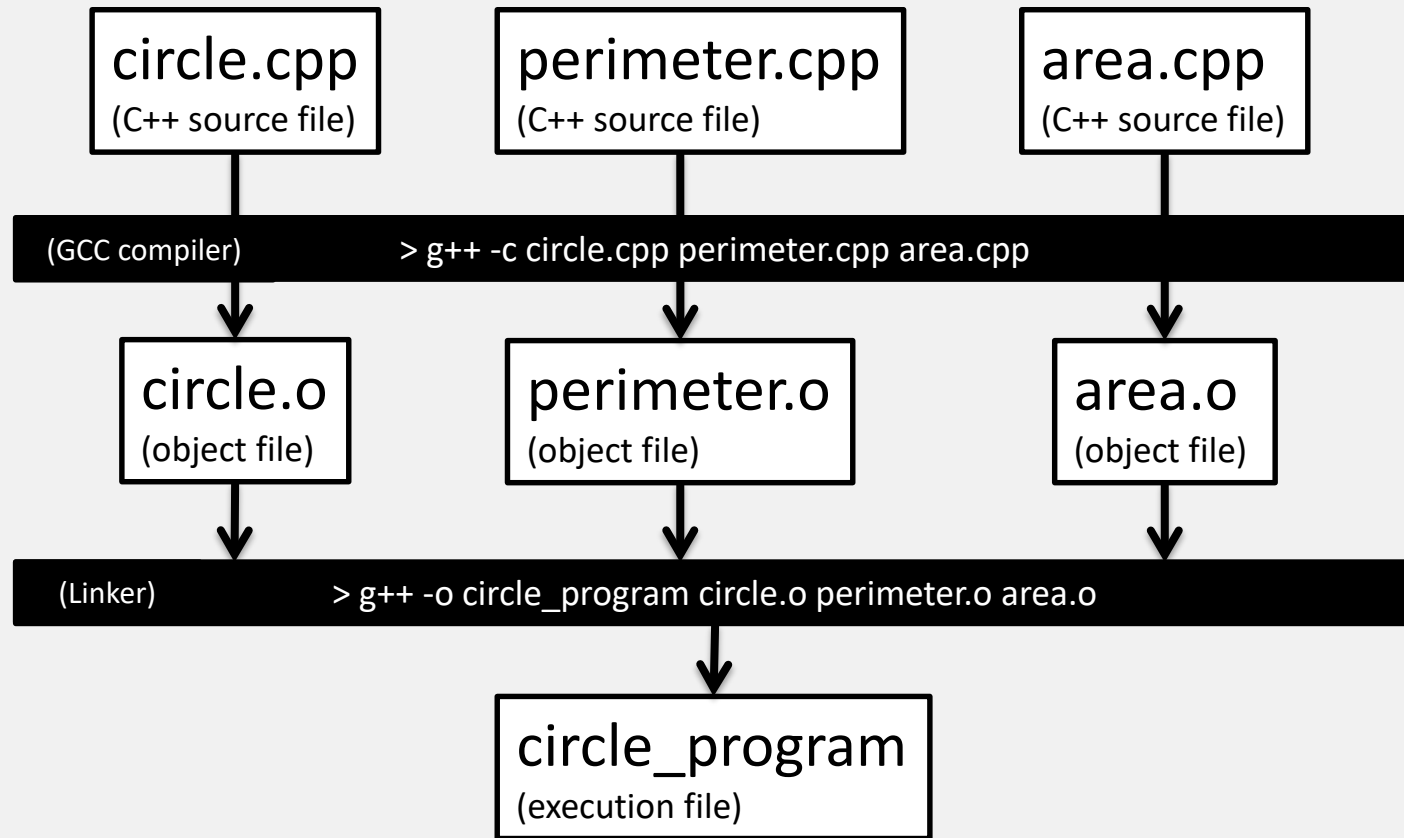
명령:  
g++ -c HelloWorld.cpp

출력:  
HelloWorld.o



# 생각해 볼 문제

- 다음의 복잡한 프로그램에 대한 Makefile을 작성해 보자



# 링커의 역할 다시보기

# 여러 개의 소스코드로 작성된 복잡한 프로그램

**circle.h**

(header file) \*/

```
#define PI 3.14
```

```
double calc_perimeter(double r);  
double calc_area(double r);
```

**circle.cpp**

(C++ source file)

```
#include <iostream>  
#include "circle.h"
```

```
void main()  
{  
    double perimeter, area, radius;
```

```
    radius = 1.0;  
    perimeter = calc_perimeter(radius);  
    area = calc_area(radius);
```

```
    std::cout << "perimeter=" << perimeter << std::endl;  
    std::cout << "area=" << area << std::endl;  
}
```

**perimeter.cpp**

(C++ source file)

```
#include "circle.h"
```

```
double calc_perimeter(double r)  
{  
    return 2*PI*r;  
}
```

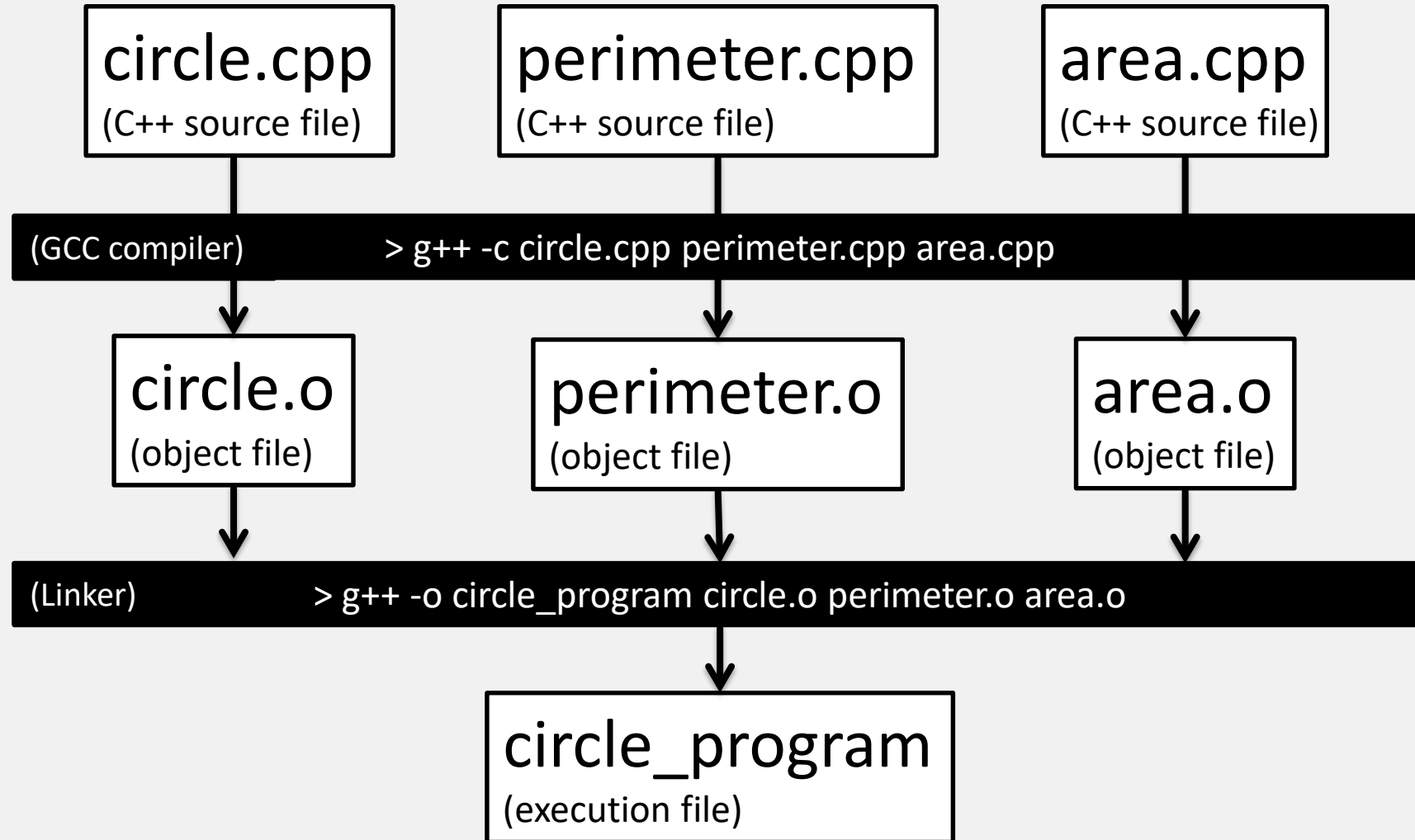
**area.cpp**

(C++ source file)

```
#include "circle.h"
```

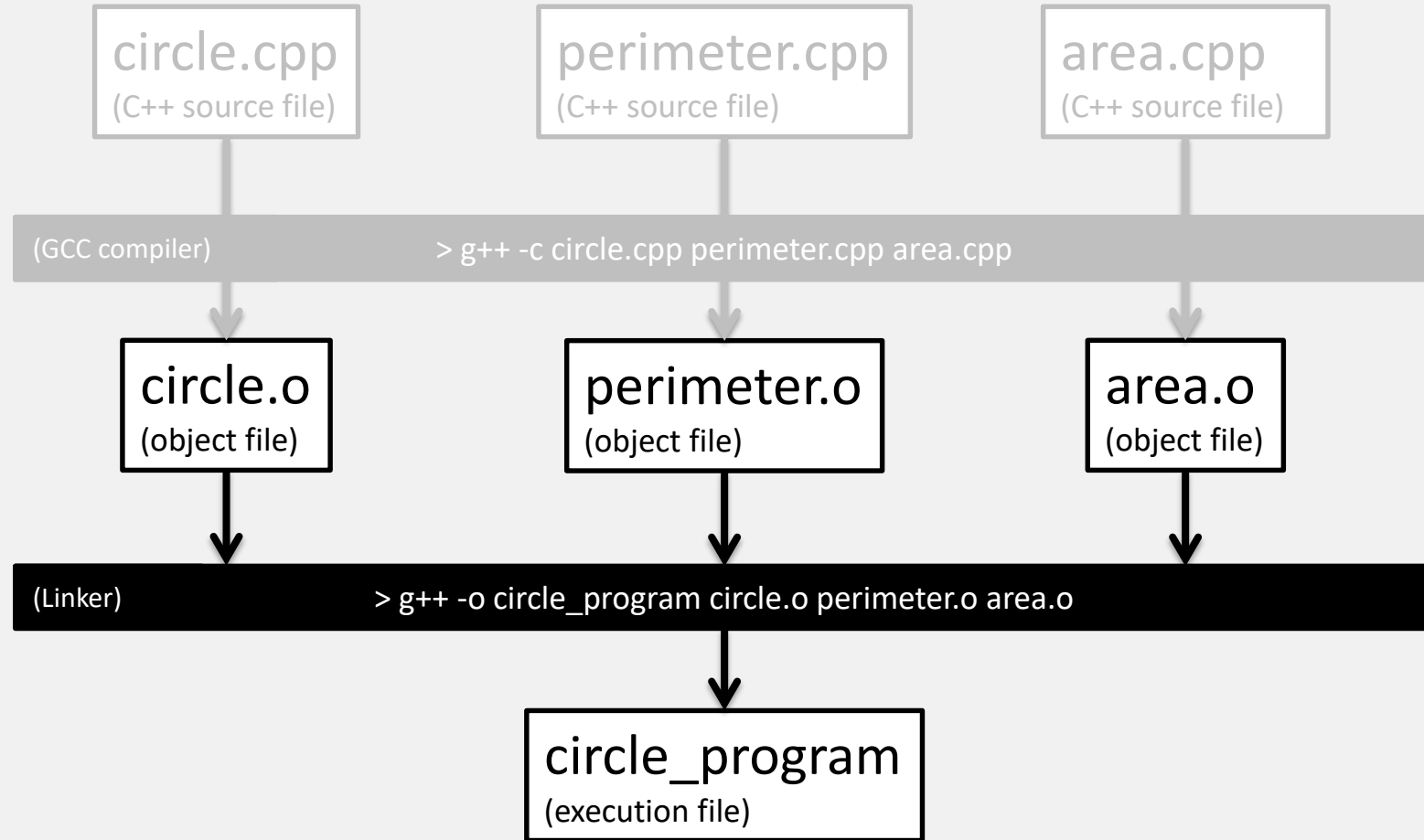
```
double calc_area (double r)  
{  
    return PI*r*r;  
}
```

# 복잡한 프로그램 작성법



# 생각해 볼 문제

- 링커의 역할은 무엇인가?



# 링커가 하는 일

- 기계친화적인 여러 파일들을 모아 실행파일을 만들어 줌
  - 기계친화적인 파일: \*.o, \*.a
- 링커는 함수호출 부분에 실제 함수구현 부분을 연결시키는 일을 수행
  - 다음 프로그램은 에러가 발생할까?

```
#include <iostream>
```

```
void my_func(int i);
```

```
void main()
```

```
{
```

```
    my_func(3);
```

```
}
```

compiler는 문법적인 것만 보기 때문에 compile error 발생X

=> ~~~.o 파일 생성 가능

my\_func이 구현된 목적 파일이 없으면 link error 발생

# 라이브러리 생성 및 라이브러리를 활용한 프로그래밍

# 프로그램

- 프로그램 = 메인 함수 + 헤더파일 + 기타 부가적인 함수들
  - 메인 함수는 프로그램의 시작점(entry point)에 해당

## [Main function]

circle.cpp  
(C++ source file)

```
#include <iostream>
#include "circle.h"

void main()
{
    double perimeter, area, radius;

    radius = 1.0;
    perimeter = calc_perimeter(radius);
    area = calc_area(radius);

    std::cout << "perimeter=" << perimeter << std::endl;
    std::cout << "area=" << area << std::endl;
}
```

## [Header files]

circle.h  
(header file)

```
#define PI 3.14

double calc_perimeter(double r);
double calc_area(double r);
```

## [Collection of functions]

perimeter.cpp  
(C++ source file)

```
#include "circle.h"

double calc_perimeter(double r)
{ return 2*PI*r; }
```

area.cpp  
(C++ source file)

```
#include "circle.h"

double calc_area (double r)
{ return PI*r*r; }
```



# 라이브러리 파일

- 프로그램 = 메인 함수 + 헤더파일 + 라이브러리
  - 라이브러리 = 기타 추가적인 함수들을 모아놓은 파일

## [Main function]

circle.cpp  
(C++ source file)

```
#include <iostream>
#include "circle.h"

void main()
{
    double perimeter, area, radius;

    radius = 1.0;
    perimeter = calc_perimeter(radius);
    area = calc_area(radius);

    std::cout << "perimeter=" << perimeter << std::endl;
    std::cout << "area=" << area << std::endl;
}
```

## [Header files]

circle.h  
(header file)

```
#define PI 3.14

double calc_perimeter(double r);
double calc_area(double r);
```

## [Library file]

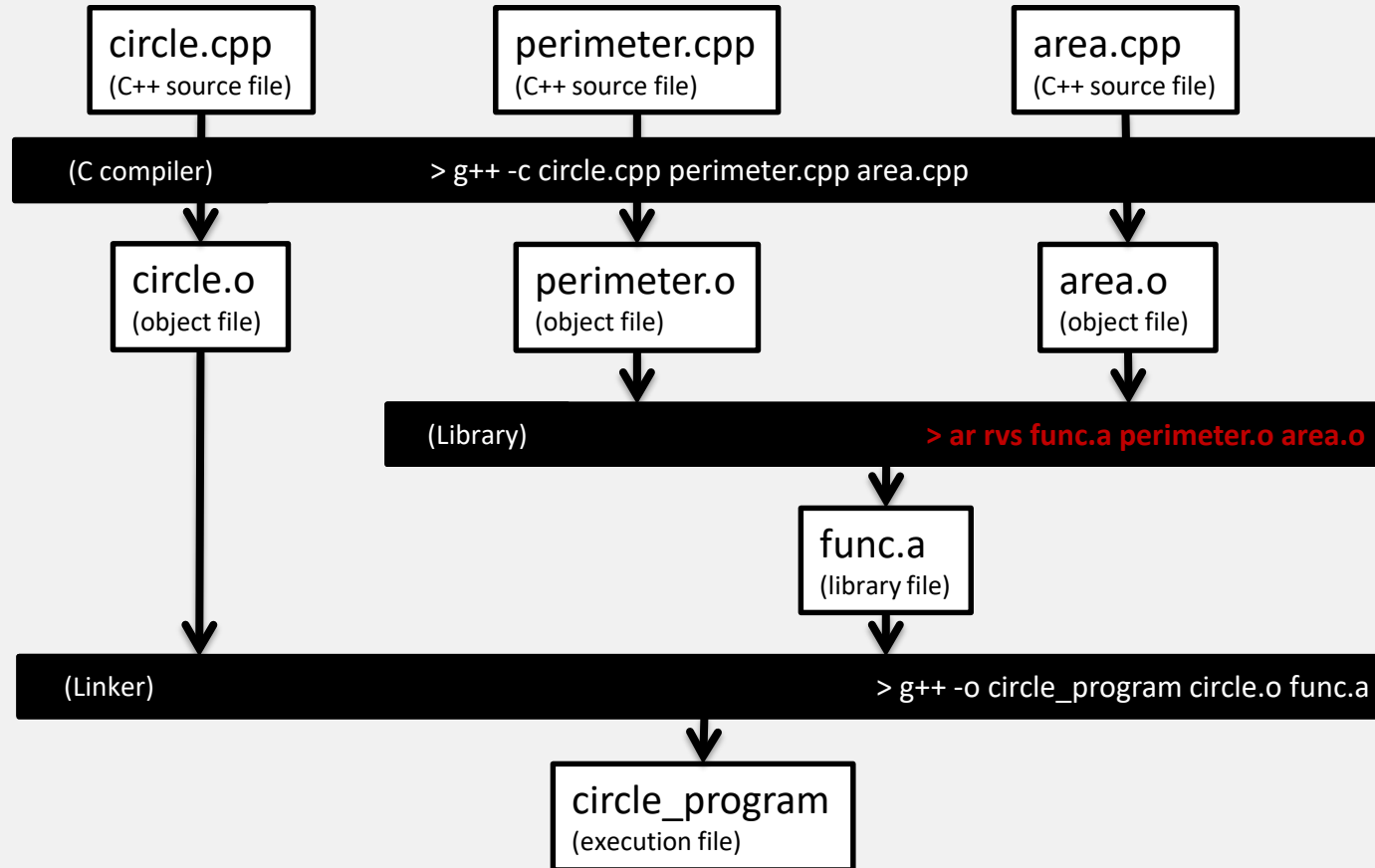
func.a  
(library file)

```
/** perimeter.o + circle.o, the binary codes for the functions,
    double calc_perimeter(double) and double calc_area (double)
 */
0101000100101010 ...
```

# 왜 라이브러리를 사용하나?

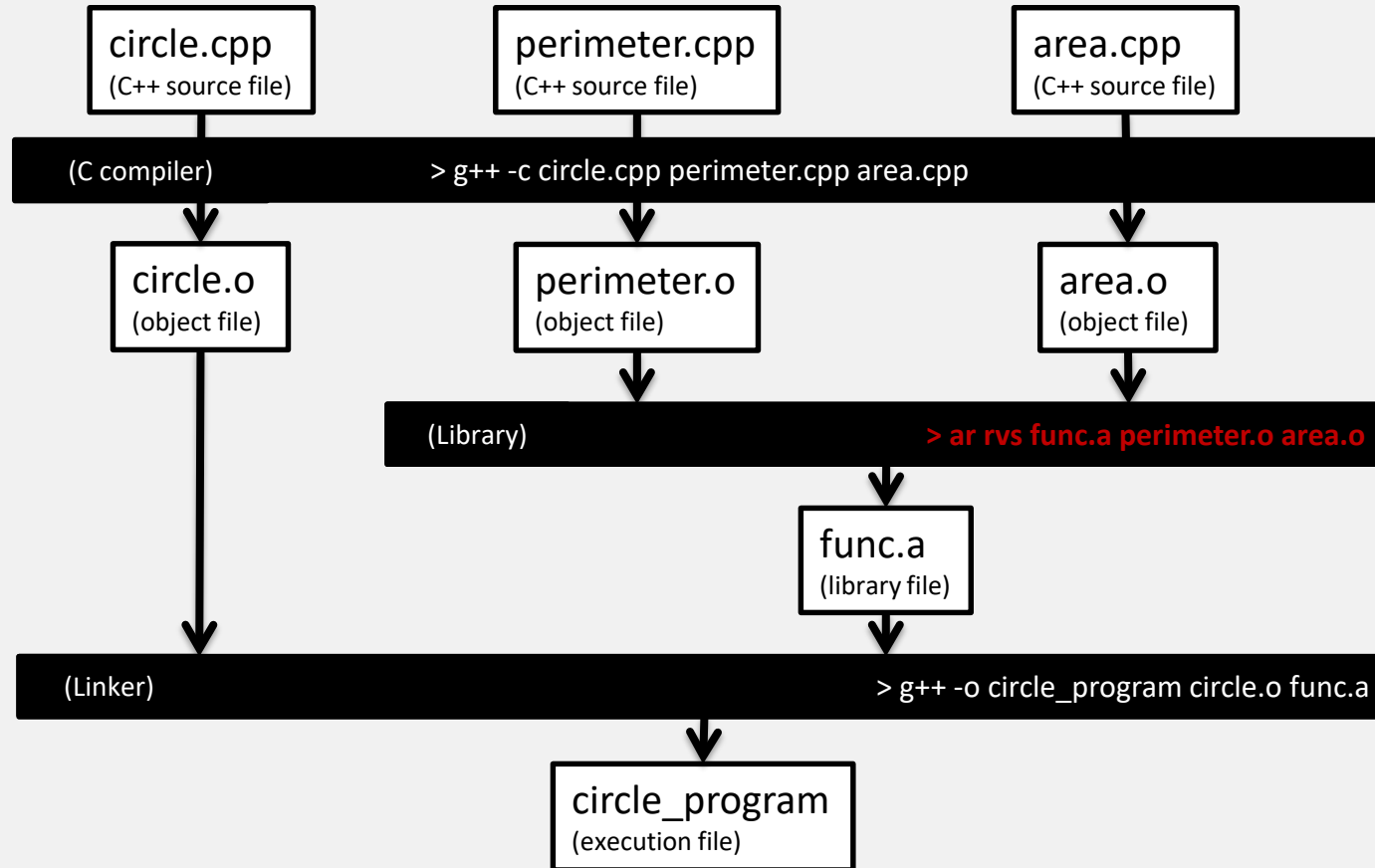
- 내가 작성한 함수들을 라이브러리 형태로 손쉽게 배포할 수 있음
- 다른 사람들은 배포된 라이브러리를 통해 내가 작성한 함수들의 기능을 모두 쓸 수 있지만, 소스코드는 볼 수 없음
- 라이브러리를 통한 함수 배포는 다음과 같이 함
  - 헤더파일 + 라이브러리 파일
    - 헤더파일: 라이브러리 파일 내에 정의된 함수들의 signature 정보 제공
    - 라이브러리 파일: 기계친화적으로 변형된 함수들의 묶음

# 라이브러리를 활용한 프로그래밍



# 생각해 볼 문제

- 라이브러리를 활용한 프로그램에 대한 Makefile을 작성해 보자



# 참고자료

- g++ (GNU C++ 컴파일러) 사용법
  - man g++
  - GCC (GNU Compiler Collection) 사이트
    - <https://gcc.gnu.org/>
  - 온라인 매뉴얼
    - <https://gcc.gnu.org/onlinedocs/>
- ld (GNU linker) 사용법
  - man ld
  - <https://sourceware.org/binutils/docs/ld/>
- make (GNU make) 사용법
  - man make
  - <https://www.gnu.org/software/make/>

# Take Home Messages!

- 전통적인 개발환경을 통해 프로그램이 형성되는 과정을 이해
  - 컴파일러 (gcc/g++), 링커 (ld), Make (make)
- 컴파일러는 문법체크만 수행함
  - 함수가 구현되어 있지 않더라도 문법적인 문제가 없으면 컴파일 에러는 발생되지 않음
- 링커는 함수의 호출과 실제 함수의 구현 부분을 연결시켜 줌
  - 실제로 함수가 구현되어 있지 않으면, 함수의 호출 부분과 연결시키는 것이 불가능하므로 링크 에러가 발생됨
- Makefile을 이용하여 컴파일, 링크 등의 작업을 일괄적으로 수행할 수 있음
- 여러 개의 소스코드로 하나의 프로그램을 개발할 수 있음
- 함수들의 묶음을 라이브러리 파일로 만들어 배포할 수 있음
  - 소스코드는 공개하지 않은 채, 함수의 기능만 배포가 가능함