

Ch.16 Preprocessing and Multi source Files

What you will learn in this chapter



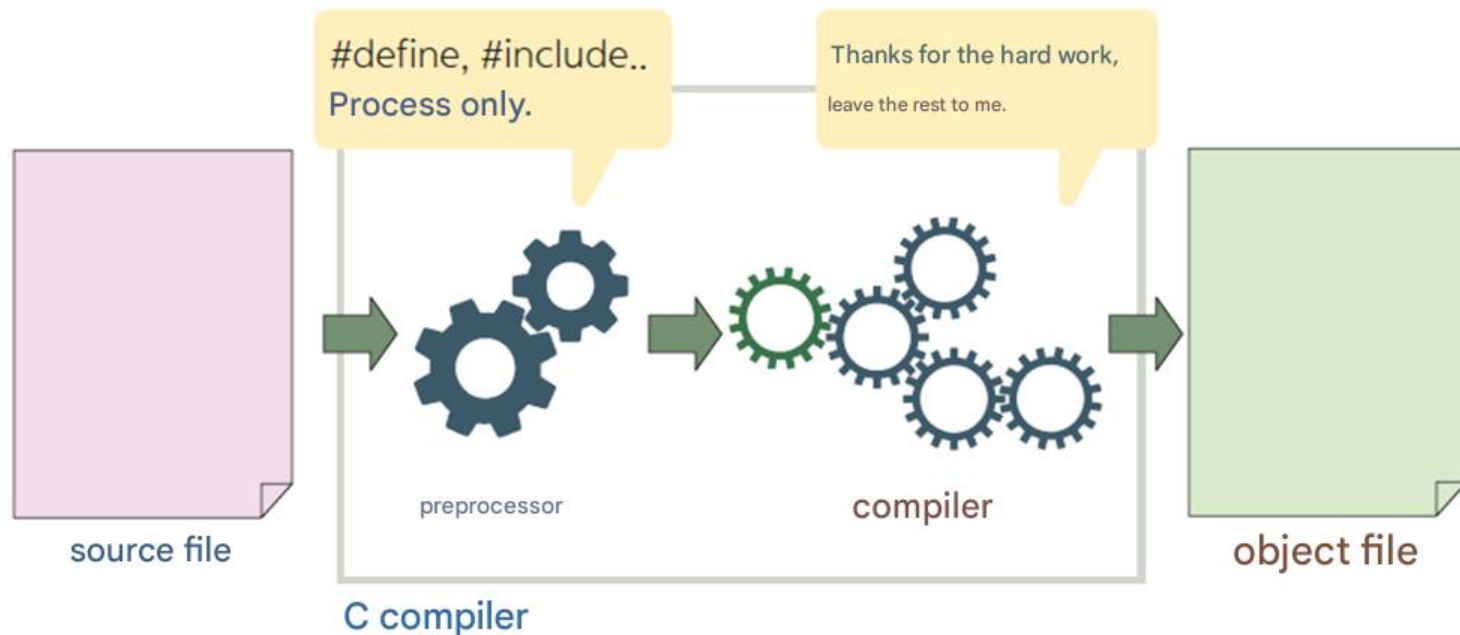
- Preprocessing directives
- Split compilation
- Command line parameters
- How to debug

Learn about preprocessing and other important topics .



What is a preprocessor ?

- A *preprocessor* is a part of a compiler that processes source files before compilation.



Summary of the preprocessor

Directive	meaning
#define	Macro definition
#include	Including files
#undef	Undefine a macro
#if	If the condition is true
#else	If the condition is false
#endif	End of conditional processing statement
#ifdef	If a macro is defined
#ifndef	If the macro is not defined
#line	Print
#pragma	Meaning varies depending on the system

Macro

A macro is a rule or pattern that specifies how input text should be transformed into replacement text during the preprocessing stage of compilation.

Syntax

Simple macro definition

yes

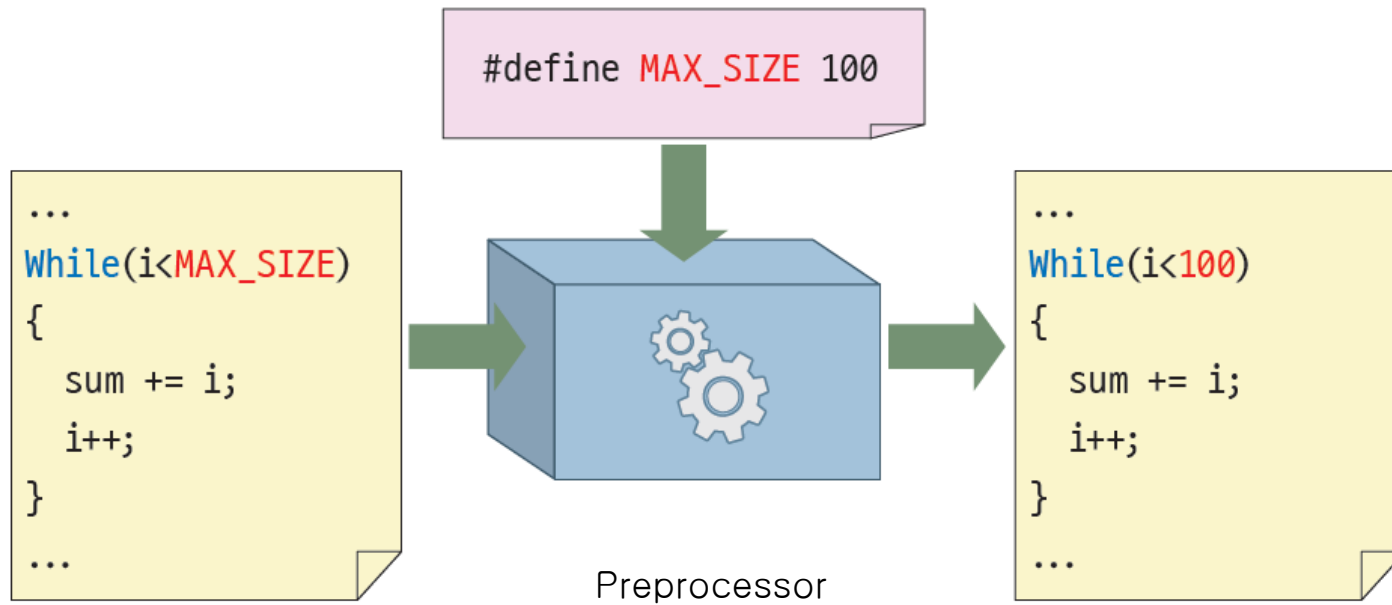
```
#define MAX_SIZE 100
```

Symbol constant MAX_SIZE
Defined as 100.

MAX_SIZE is easier to
understand than 100.



Macro



Advantages of Macros

- Improves the readability of the program .
- It is easy to change constants .

```
#define MAX_SIZE 100  
for(i=0;i<MAX_SIZE;i++)  
{  
    f += (float) i/MAX_SIZE;  
}
```



```
#define MAX_SIZE 200  
for(i=0;i<MAX_SIZE;i++)  
{  
    f += (float) i/MAX_SIZE;  
}
```

Example of macro

```
#define PI 3.141592 // pi
#define EOF (-1) // End of file indicator
#define EPS 1.0e-9 // Calculation limit for real numbers
#define DIGITS "0123456789" // Define character constants
#define BRACKET "(){}[]" // Define character constants
```

Example

- Let's try using the `#define` directive to change the operator `&&` to `AND` .

```
int i = 0;

while ( i < n AND list[ i ] != key )
    i ++;
if ( i IS n )
    return -1;
else
    return i ;
```

How should I define a
preprocessor ?



Example

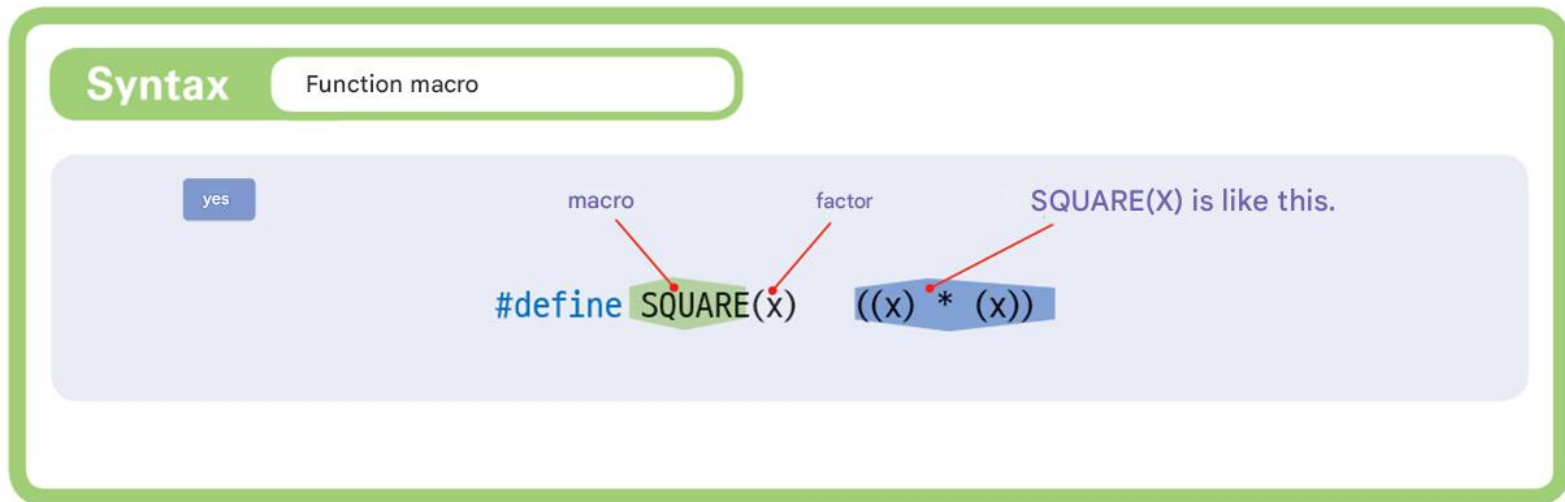
```
#include <stdio.h >
#define AND &&
#define OR ||
#define NOT !
#define IS ==
#define ISNOT !=

int search( int list[], int n, int key)
{
    int i = 0;

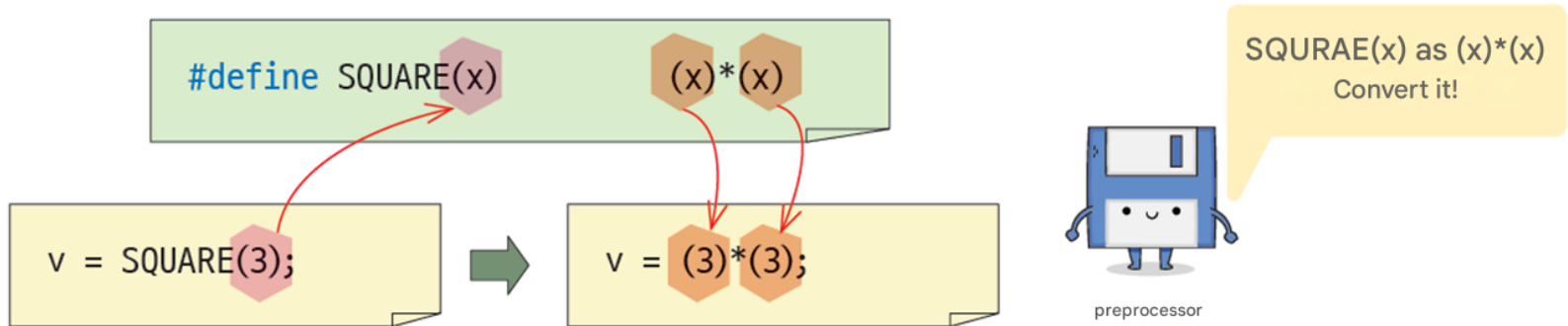
    while ( i < n AND list[ i ] != key )
        i ++;
    if ( i IS n )
        return -1;
    else
        return i ;
}
```

Function-like Macro

- A function-like macro is a macro that looks and behaves like a function call because it takes parameters, but the preprocessor simply performs textual substitution instead of real function execution.



Function Macro



Example of a function macro

```
#define SUM(x, y) ((x) + (y))  
#define AVERAGE(x, y, z) (( (x) + (y) + (z) ) / 3 )  
#define MAX(x,y) ( (x) > (y) ) ? (x) : (y)  
#define MIN(x,y) ( (x) < (y) ) ? (x) : (y)
```

Macro

Factor

simply performs textual substitution

Things to note

```
#define SQUARE(x) x*x // DANGER !!
```

```
v = SQUARE( a+b );
```



```
v = a + b*a + b;
```


In function macros, it is a good idea to surround parameters with **parentheses** .

```
#define SQUARE(x) (x)*(x) // Correct form
```

Caution


1. When defining a macro, all parameters must be used.

```
#define HALFOF(y, x) ((x) / 2) // ERROR!!
```



2. There should be no space between the macro and the parentheses

```
#define ADD(x, y) ((x) + (y)) // ERROR!!
```



Because there is a space ADD and (, the preprocessor thinks it is a symbolic constant definition and replace the string ADD with (x, y) ((x) + (y))

Example #1

```
// Macro Example
#include <stdio.h>
#define SQUARE(x) ((x) * (x))

int main( void )
{
    int x = 2;

    printf ( "%d\n" , SQUARE(x));
    printf ( "%d\n" , SQUARE(3));
    printf ( "%f\n" , SQUARE(1.2)); // Even if real number can be applied
    printf ( "%d\n" , SQUARE(x+3));
    printf ( "%d\n" , 100/SQUARE(x));
    printf ( "%d\n" , SQUARE(++x)); // Logic error

    return 0;
}
```

```
4
9
1.440000
25
25
16
```

Built-in macros

- Built-in macros : predefined macros

Built-in macros	explanation
__DATE__	When this macro is encountered, it is replaced with the current date (month day year) .
__TIME__	When this macro is encountered, it is replaced with the current time (hour : minute : second) .
__LINE__	When this macro is encountered, it is replaced with the current line number in the source file .
__FILE__	When this macro is encountered, it is replaced with the source file name .

```
printf("Compile date =%s\n" , __DATE__ );  
printf("Fatal error occurred File name = %s Line number = %d\n" , __FILE__ , __LINE__ );
```



```
Compile Date = Aug 23 2021  
Fatal error occurred File name =C:\Users\Wkim\source\repos\Project14\Project14\ s  
ource .c Line number = 6
```

ASSERT Macro

- ASSERT macro that is frequently used when debugging programs.



Assuming (sum == 0) this source file
C:\Users\chun\source\repos\Project21\Project21\macro4.c
Failed at line 12 .

```
#include <assert.h>
```

```
assert(x > 0);
```

If condition is false, then exits the program.

ASSERT Macro (User defined)

```
#include <stdio.h>
```

```
#define ASSERT( exp ) { if (!( exp )) \
    { printf("Assumption("# exp") source file %s % dth line failed.\n", \
    __FILE__, __LINE__), exit(1);}}
```

```
int main( void )
```

```
{
```

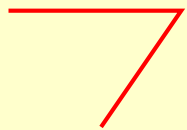
```
    int sum=100;    // The initial value of local variable's is not 0
```

```
    ASSERT(sum == 0); // sum should be 0
```

```
    return 0;
```

```
}
```

Used to extend a macro
to the next line



Assuming (sum == 0) this source file c:\user\igchun\documents\visual studio
2017\projects
Failed at line 12

Function Macros and Functions

- Advantages of function-like macros:
 - They run faster because they don't actually "call" a function. Instead, the code is copied directly into the program at that spot.
- Disadvantages of function-like macros:
 - Macros usually must stay short. Often they should only be one, two, or three lines long.
 - Using many macros can make your source code bigger, because every time you use a macro, the code gets copied again.

#ifdef

Syntax

conditional compilation

#if and #endif if the macro DEBUG is defined
Compile all sentences in between.

yes

```
#ifdef DEBUG  
    printf("value=%d\n", value);  
#endif
```

Example of #ifdef

```
int average(int x, int y)
{
    printf("x=%d, y=%d\n", x, y);

    return (x+y)/2;
}
```

```
int average(int x, int y)
{
#ifdef DEBUG
    printf("x=%d, y=%d\n", x, y);
#endif
    return (x+y)/2;
}
```

Include output statements only
when DEBUG is declared .



Macro declaration location

```
#define DEBUG
```

```
int average(int x, int y) {
```

컴파일에 포함

```
#ifdef DEBUG  
    printf("x=%d, y=%d\n", x, y);  
#endif
```

```
    return (x+y)/2;
```

```
}
```

```
int average(int x, int y) {
```

컴파일에 포함되지
않음

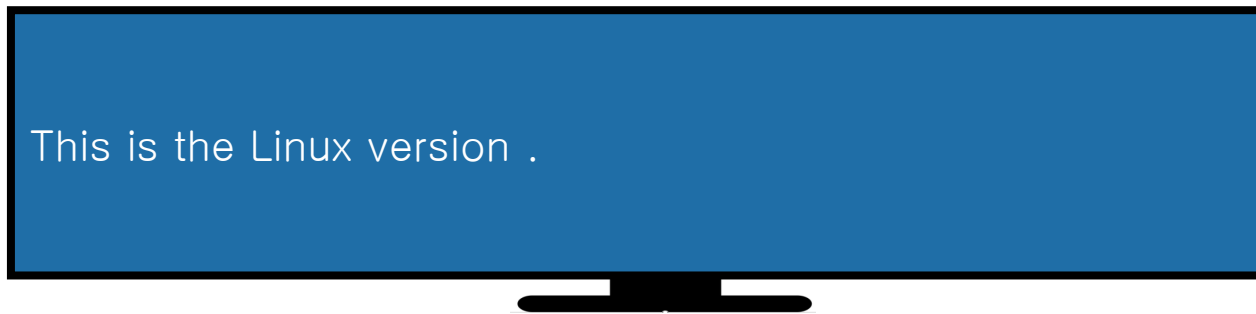
```
#ifdef DEBUG  
    printf("x=%d, y=%d\n", x, y);  
#endif
```

```
    return (x+y)/2;
```

```
}
```

Separate Linux and Windows versions

- For example, let's say a company develops a program for Linux and Windows.



Example

```
#include <stdio.h>
```

```
#define LINUX
```

```
int main( void )
```

```
{
```

```
# ifdef LINUX
```

```
printf ( " This is the Linux version . \n" );
```

```
#else
```

```
printf ( " Windows version . \n" );
```

```
# endif
```

```
return 0;
```

```
}
```

LINUX version

WINDOWS VERSION

ifndef , # undef

- #ifndef
 - If a macro is not defined, it is included in the compilation .

```
#ifndef LIMIT
#define LIMIT 1000
#endif
```

If LIMIT is not defined

Now define LIMIT

- #undef
 - Cancels the definition of a macro .

```
#define SIZE 100
..
#undef SIZE
#define SIZE 200
```

Cancels the definition

#if

- Compile if symbol evaluates to true
- Conditions must be constants and can use logical and relational operators.

Syntax

conditional compilation

If the value of the macro DEBUG is 1, all statements between #if and #endif are compiled.

yes

```
#if DEBUG==1
    printf("value=%d\n", value);
#endif
```

Usage Form	Meaning	How to use
#define DEBUG	Only checks existence	#ifdef DEBUG
#define DEBUG 1	Value-based condition	#if DEBUG >= 2

#if-#else-#endif

```
#define NATION 1

#if NATION == 1
    printf ( " Hello ?" );
#elif NATION == 2
    printf ( " Are you okay ?" );
#else
    printf ( "Hello World!" );
#endif
```

Various examples

```
#if (AUTHOR == KIM) // Possible !! KIM is another macro
```

```
#if (VERSION*10 > 500 && LEVEL == BASIC) // Possible !!
```

```
#if (VERSION > 3.0) // ERROR !! The version number is displayed as an integer
```

```
#if (AUTHOR == "CHULSOO" ) // ERROR !!
```

Comment out multiple lines

```
#if 0 // Start from here
```

```
void test()
```

```
{
```

```
/* If there is a comment here, it is not easy to comment out the entire code . */
```

```
sub();
```

```
}
```

```
#endif // Up to here, it is commented out .
```

Multiple source files

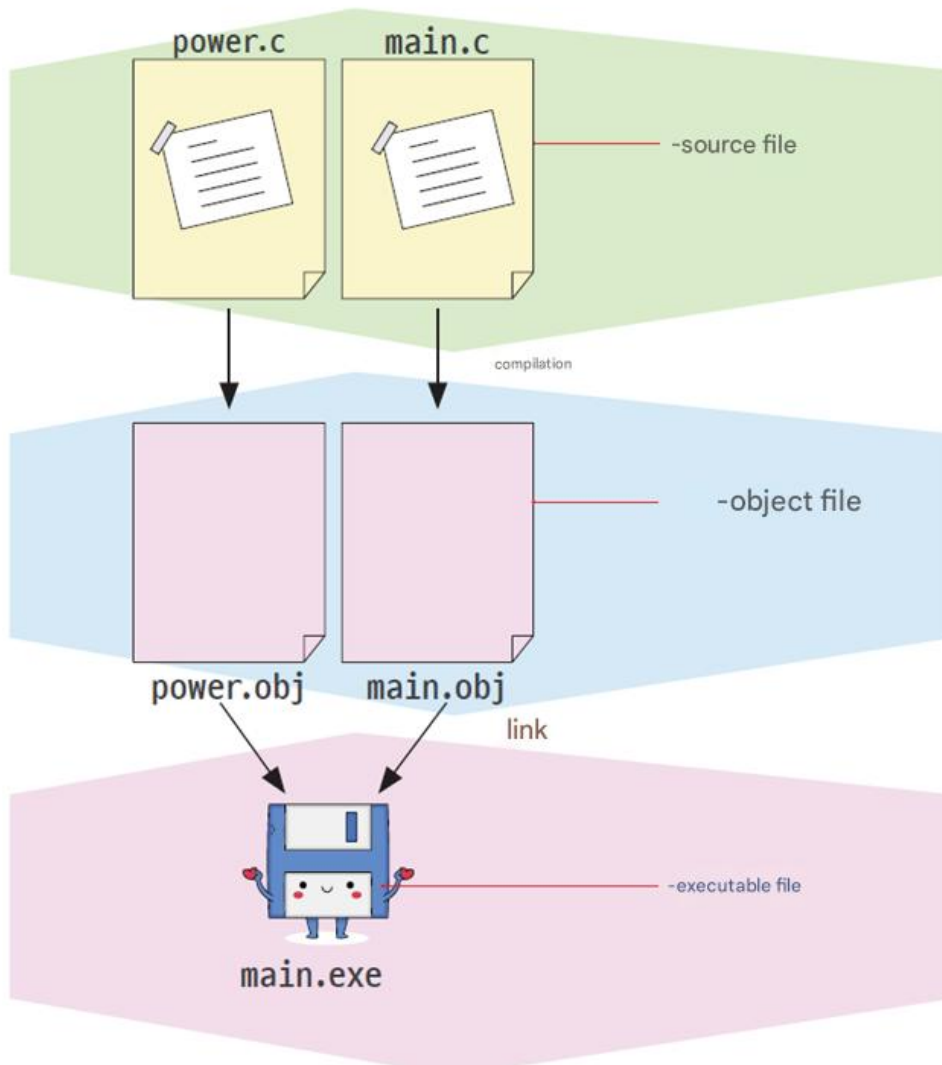
- *Single source file*

- The file size is too large .
- Difficult to reuse source files

- *Multiple source files*

- You can collect only related codes into a single source file.
- Easy to reuse source files

Multiple source files



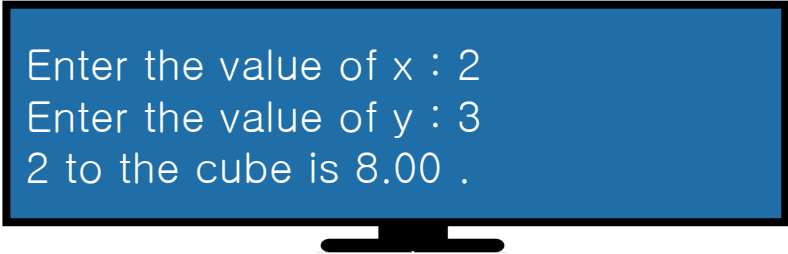
```
gcc -c main.c  
gcc -c power.c
```

```
gcc main.o power.o -o main
```

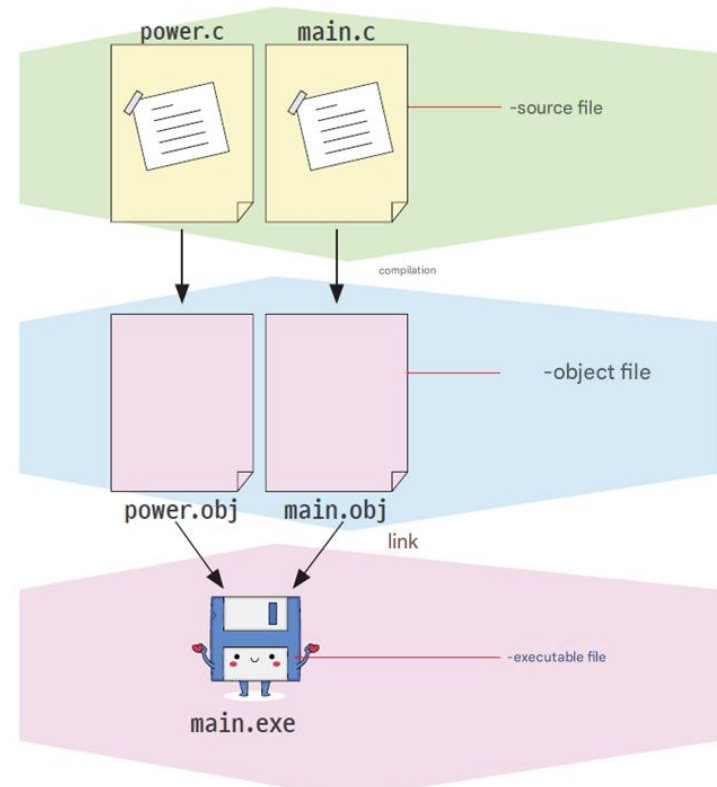
** gcc main.c power.c -o program*

Example :

- `power()` that calculates the power and save it in `power.c`
Then, `main.c` create a function and define the `main()` function in it,
then call `power()` from `main()` .



Enter the value of x : 2
Enter the value of y : 3
2 to the cube is 8.00 .



Example

```
// main.c
#include <stdio.h>
#include "power.h"

int main( void )
{
    int x, y;

    printf ( " Enter the value of x : " );
    scanf ( "%d" , &x);
    printf ( " Enter the value of y : " );
    scanf ( "%d" , &y);
    printf ( " The power of % d of %d is %f\n" , x, y, power(x, y));

    return 0;
}
```

Example

```
// power.c
#include "power.h"

double power( int x , int y )
{
    double result = 1.0; // Initial value is 1.0
    int i ;

    for (i = 0; i < y ; i++)
        result *= x ;

    return result;
}
```

Example

```
#pragma once
// power. h

double power( int x , int y ); // Function prototype definition
```

```
#ifndef POWER_H
#define POWER_H

// power. h

double power( int x , int y ); // Function prototype definition

#endif
```

If you don't use header files

```
void draw_line(...)
{
    ....
}
void draw_rect(...)
{
    ....
}
void draw_circle(...)
{
    ....
}
```

graphics.c

Provider

함수 원형 정의가 중복되어 있음

```
void draw_line(...);
void draw_rect(...);
void draw_circle(...);
```

```
int main(void)
{
    draw_rect(...);
    draw_circle(...);
    ...
    return 0;
}
```

main.c

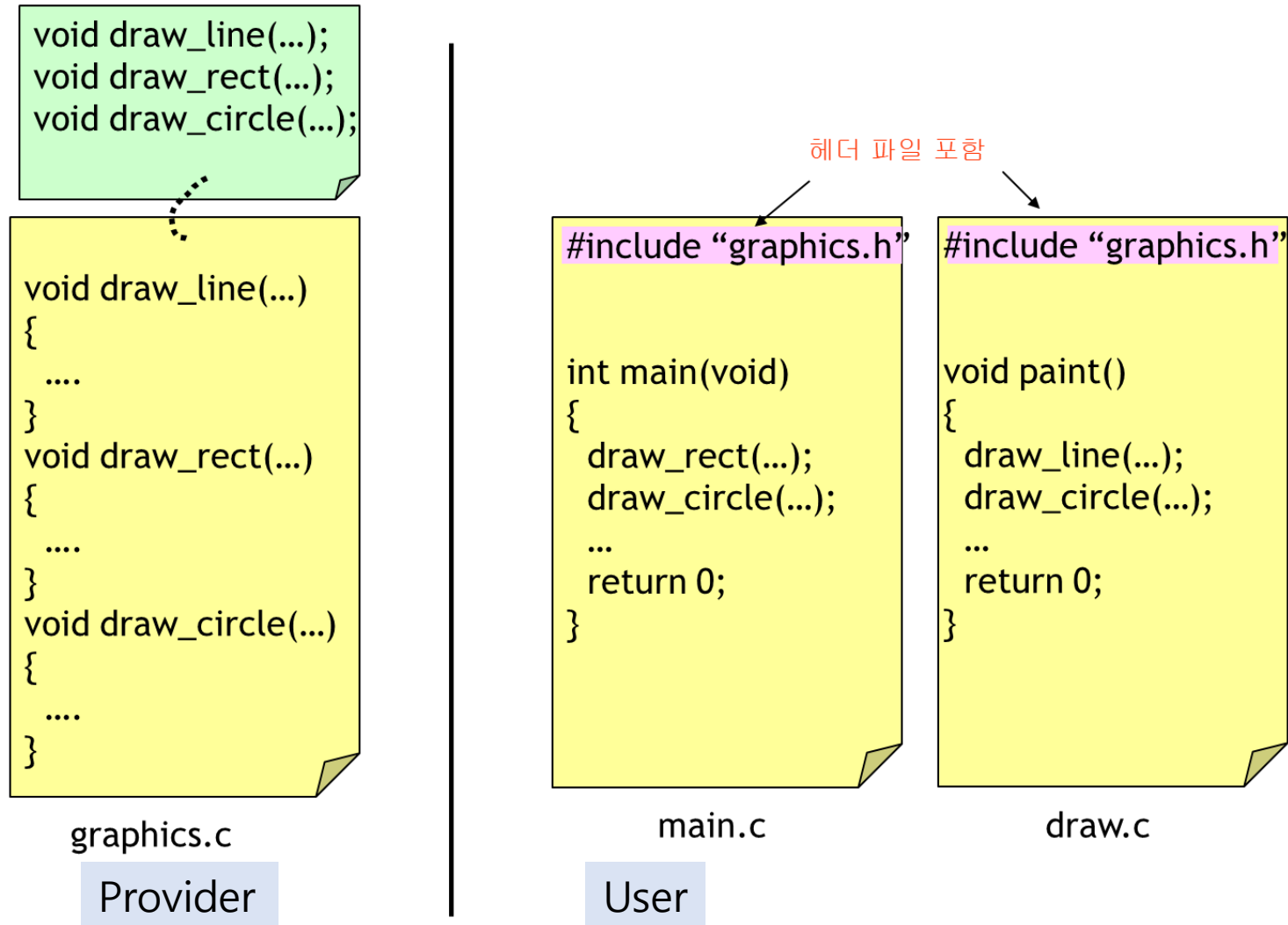
User

```
void draw_line(...);
void draw_rect(...);
void draw_circle(...);
```

```
void paint()
{
    draw_line(...);
    draw_circle(...);
    ...
    return 0;
}
```

draw.c

Using header files



External variables in multiple source files

Use variables declared in external source files
To do this, use extern.

하려면 extern을 사용한다.

```
double gx, gy;
```

```
int main(void)
{
    gx = 10.0;
    ...
}
```

main.c

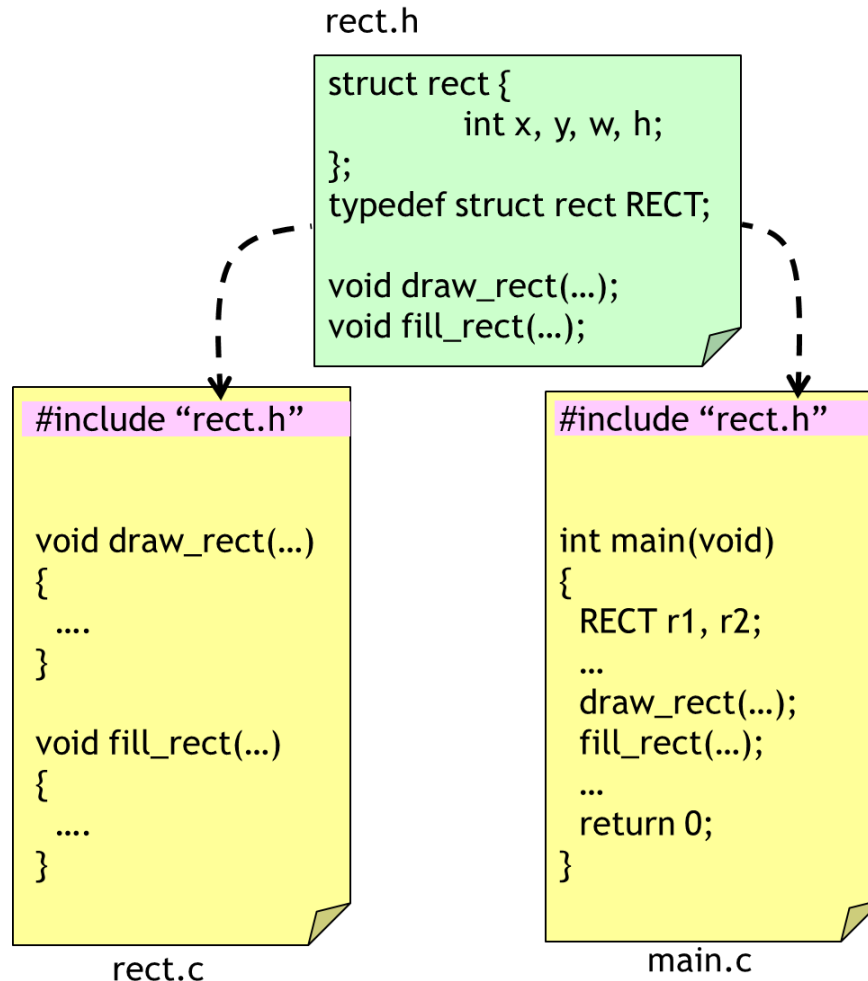
```
extern double gx, gy;
```

```
int power(void)
{
    ...
    result *= gx;
}
```

power.c

Example

- Let's write the following program with multiple sources .



rect.h

```
#pragma once
struct rect {
    int x, y, w, h;
};

typedef struct rect RECT ;

void draw_rect ( const RECT *);
double calc_area ( const RECT *);
void move_rect ( RECT *, int , int );
```

rect.c 1/2

```
#include <stdio.h>
#include "rect.h"
#define DEBUG

void draw_rect ( const RECT * r )
{
#ifdef DEBUG
    printf( "draw_rect (x=%d, y=%d, w=%d, h=%d) \n" , r ->x, r ->y, r ->w, r ->h);
#endif
}
```

rect.c 2/2

```
double calc_area ( const RECT * r )
{
    double area;
    area = r ->w * r ->h;
#ifdef DEBUG
    printf ( " calc_area ()=%f \n" , area);
#endif
    return area;
}

void move_rect ( RECT * r , int dx , int dy )
{
#ifdef DEBUG
    printf ( " move_rect (%d, %d) \n" , dx , dy );
#endif
    r ->x += dx ;
    r ->y += dy ;
}
```

main.c


```
#include <stdio.h>
#include "rect.h"

int main( void )
{
    RECT r = { 10,10, 20, 20 };
    double area = 0.0;

    draw_rect (&r);
    move_rect (&r, 10, 20);
    draw_rect (&r);
    area = calc_area (&r);
    draw_rect (&r);
    return 0;
}
```

Execution results

```
draw_rect (x=10, y=10, w=20, h=20)
move_rect (10, 20)
draw_rect (x=20, y=30, w=20, h=20)
calc_area ()=400.00
draw_rect (x=20, y=30, w=20, h=20)
```

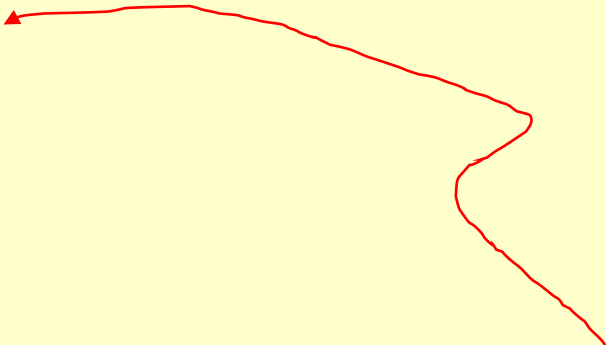


Lab: Header file Duplicate blocking

- a header file containing a structure definition twice in your source file, you will get a compilation error. To prevent this, use `#ifndef` You can use directives

```
#ifndef STUDENT_H
#define STUDENT_H

struct STUDENT {
    int number;
    char name[10];
};
#endif
```



No compilation errors even if included multiple times in a file

TIP

- recent C languages, you can achieve the same effect by adding the following statement to the beginning of the header file.
When you add a header file in Visual Studio, it is automatically added to the beginning.

#pragma once

Inspection

1. Tell the following sentences true or false.
" It is advantageous to create a single source file rather than using multiple source files in many ways ."
2. Let's create a source file and related header file that contain a function to find the factorial.
3. Let's create a header file that defines a point structure, which represents a point in two-dimensional space.



Q & A

