Ch.16 Preprocessing and Multi source Files

What you will learn in this chapter
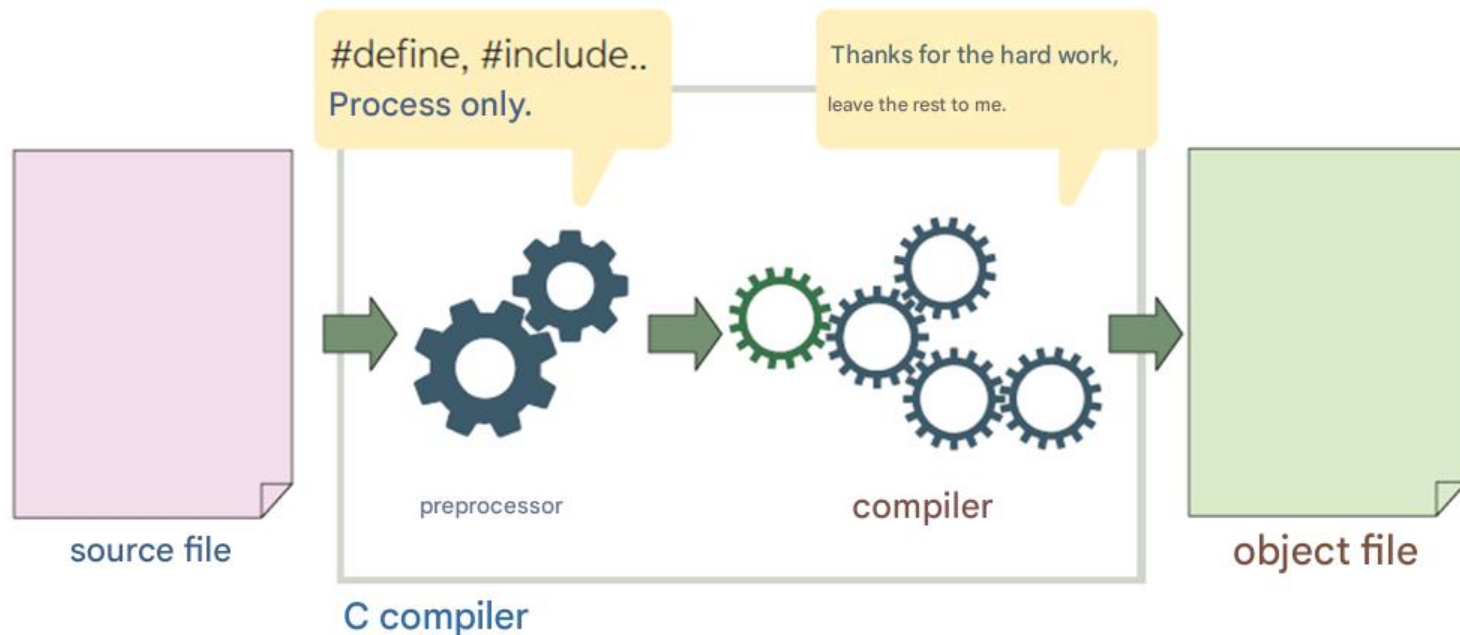
•Preprocessing directives
•Split compilation
•Command line parameters
•How to debug

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

Simple Macro

MAX_SIZE is easier to
understand than 100.

Simple Macro

Preprocessor
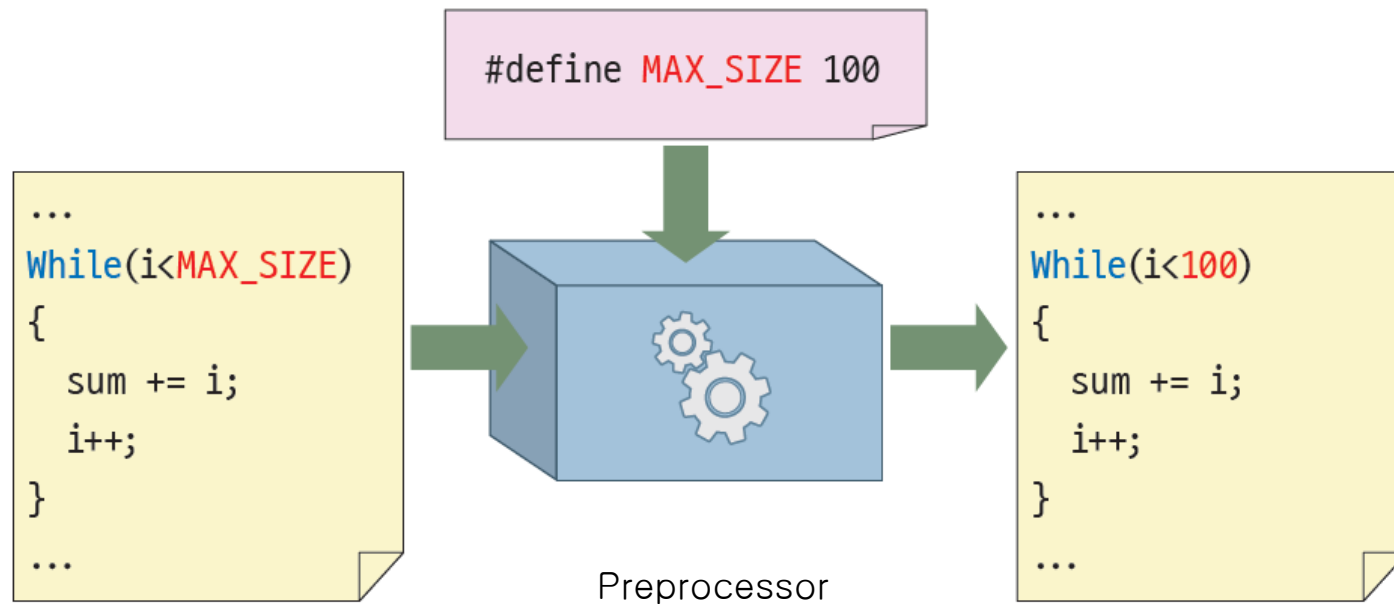
Advantages of Simple Macros

- 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 a simple 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
#define getchar () getc (stdin) // defined in
#define putchar () putc ( stdout ) // defined in
```

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

```c
#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 ;
}
```
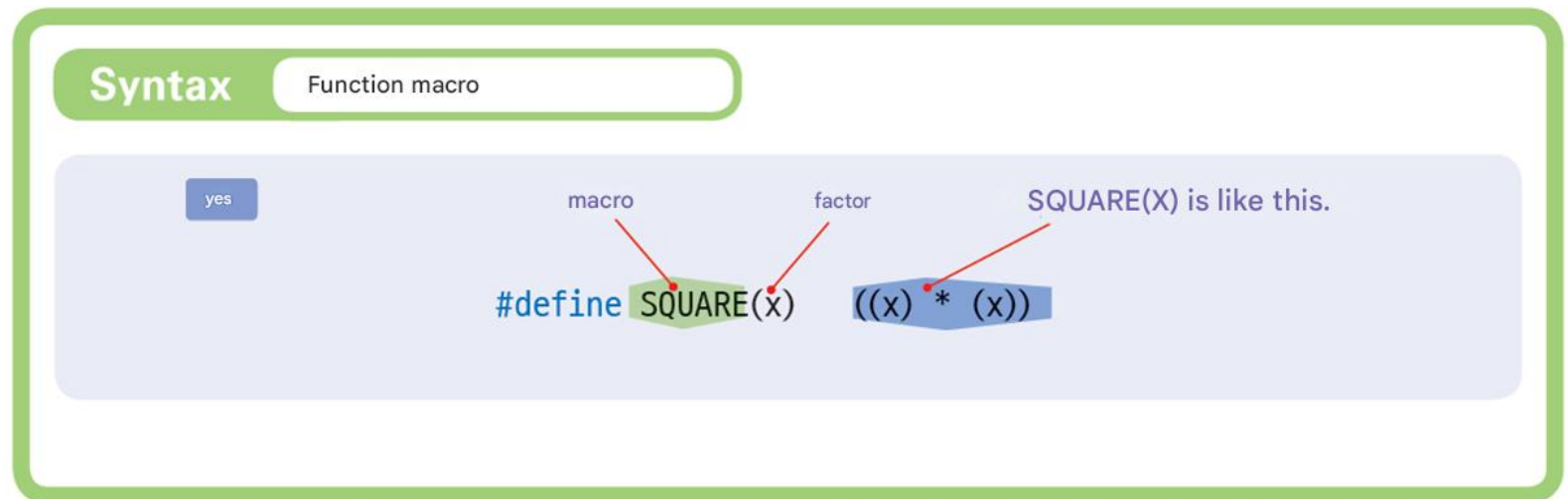
Example

```
int main( void )
{
    int m[] = { 1, 2, 3, 4, 5, 6, 7 };
    printf ( " Position of 5 in array =%d\n" , search(m, sizeof (m) / sizeof (m[0]), 5));
    return 0;
}
```
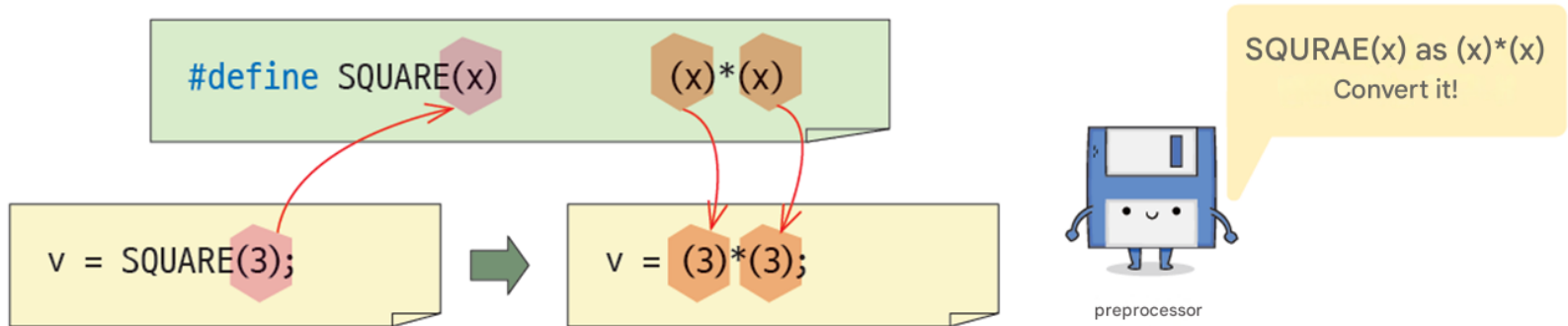
Position of 5 in array = 4

Function Macro

- *A function -like macro* is a macro that has parameters like a function.

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)
```

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

Pros and Cons of Function Macros

- Pros and Cons of Function Macros
  - It is fast to execute because there is no need for a function call step.
  - The length of the source code increases .

- Simple functions use macros
  - #define MIN(x, y) ((x) < (y) ? (x) : (y))

- How to extend a macro beyond one line
  #define PRINT(x) if(debug==1 && ₩
                    mode==1) ₩
                 printf ("%d", x);

Things to note

Precautions when using macros

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 name and the parentheses.

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

Because there is a space between ADD and (, the preprocessor thinks it is a symbolic constant definition and replaces the string ADD with

(x, y) ((x) + (y)).

Example #1

```c
// 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\kim\source\repos\Project14\Project14\ source .c Line number = 6

Lab: 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 .
```

Example : ASSERT Macro

```c
#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;
}
```
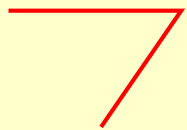
Used to extend a macro to the next line

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

Function Macros and Functions

- The advantage is that function macros are faster to execute than functions. Since macros are not called but rather code is inserted at that location, there is no need to go through the complicated steps of a function call.

- The length of the code cannot be longer than a certain limit. In many cases, the limit is one or two or three lines.

- Using macros increases the size of your source files.

#ifdef

Syntax — conditional compilation

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

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

Example of #ifdef

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

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

```c
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;
}
```
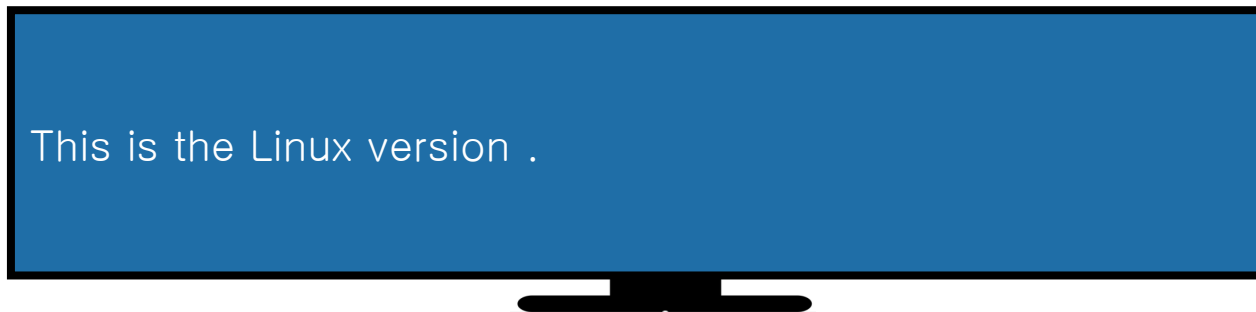
Lab: **Separate Linux and Windows versions**

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

This is the Linux version .

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
```

LIMIT가 정의되어 있지 않으면

LIMIT를 정의해준다.

- # undef
  - Cancels the definition of a macro .

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

SIZE의 정의를 취소한다.

#if

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

Syntax — conditional compilation

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

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

#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 .
```

Example

- Choose a sorting algorithm

```
#define SORT_METHOD 3

#if (SORT_METHOD == 1)
...    // Implementation of selection sort
#elif (SORT_METHOD == 2)
...    // Bubble sort implementation
#else
...    // Quick sort implementation
#endif
```
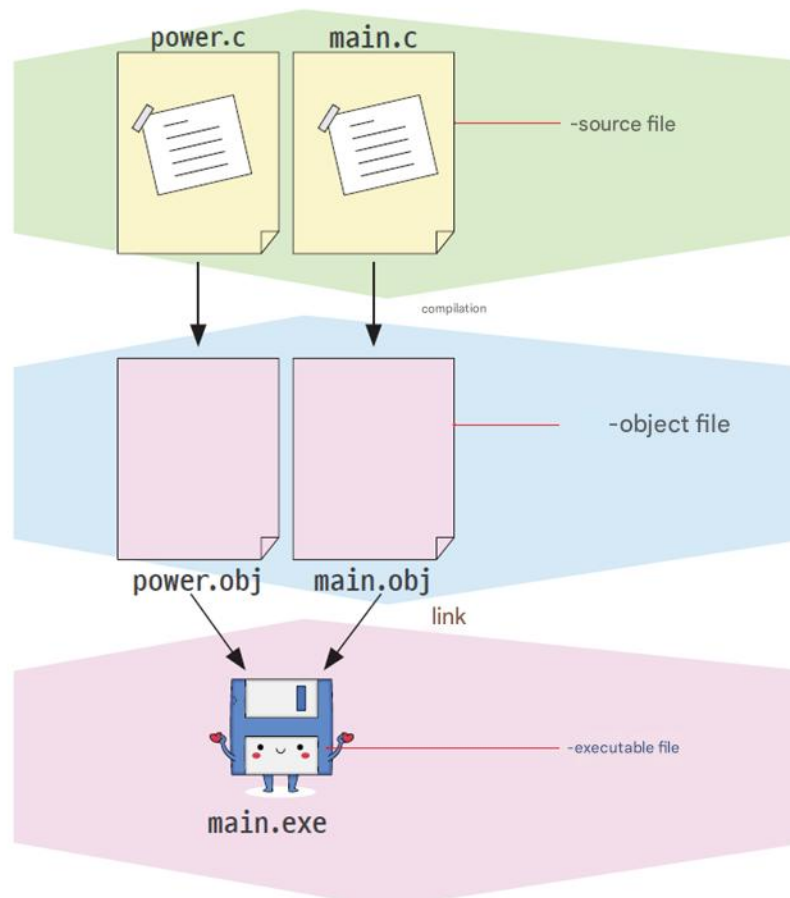
Multiple source files

- *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
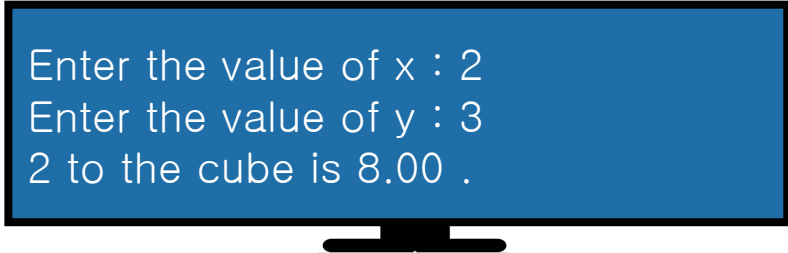
Multiple source files

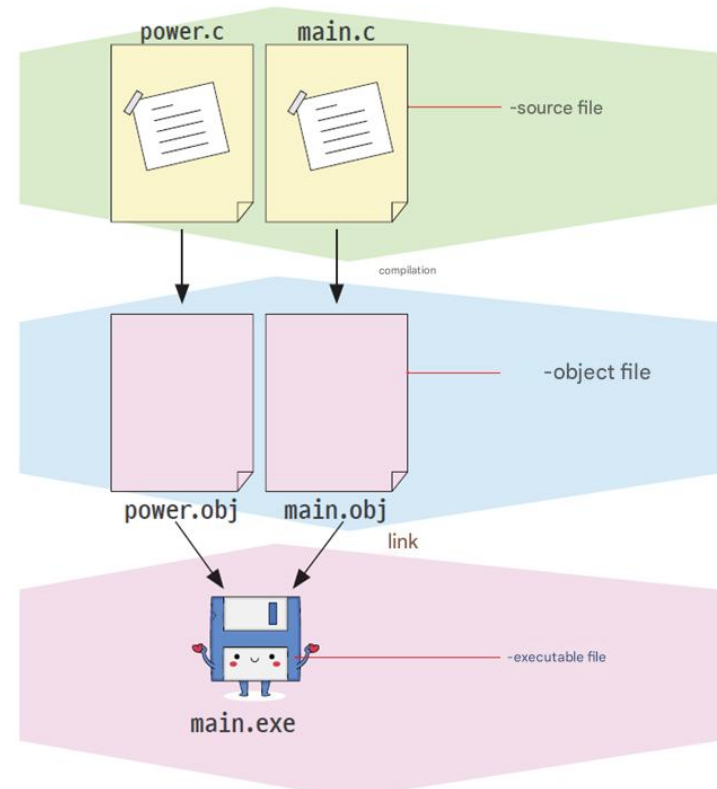# Example :

- 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

```c
// 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

```c
// 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
```

If you don't use header files

graphics.c

공급자

함수 원형 정의가 중복되어 있음

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

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


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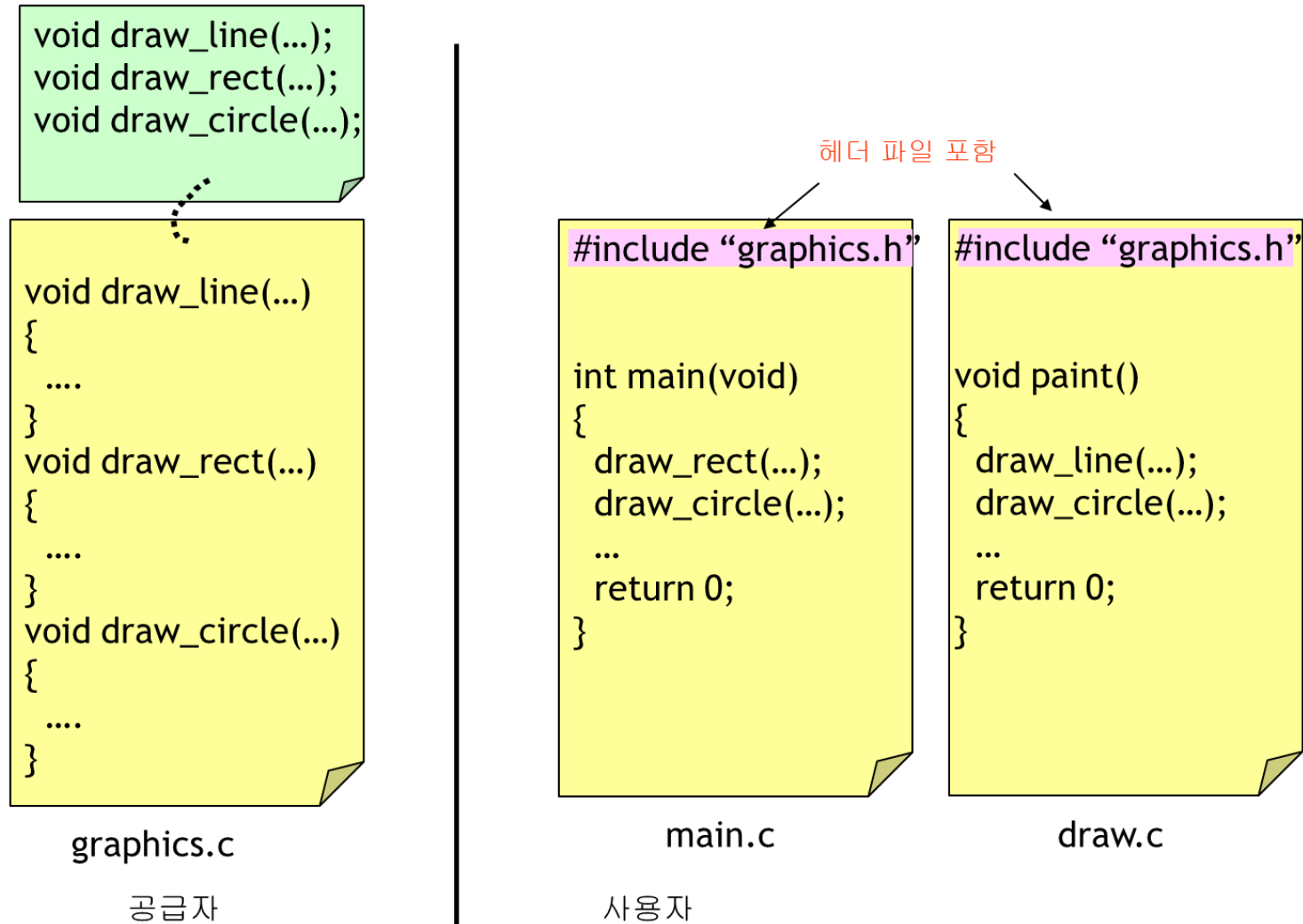
main.c

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


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

draw.c

사용자

Using header files

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

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

graphics.c

공급자

헤더 파일 포함

```
#include "graphics.h"

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

main.c

```
#include "graphics.h"

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

draw.c

사용자

External variables in multiple source files

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


```
double gx, gy;

int main(void)
{
    gx = 10.0;

    ...

}
```
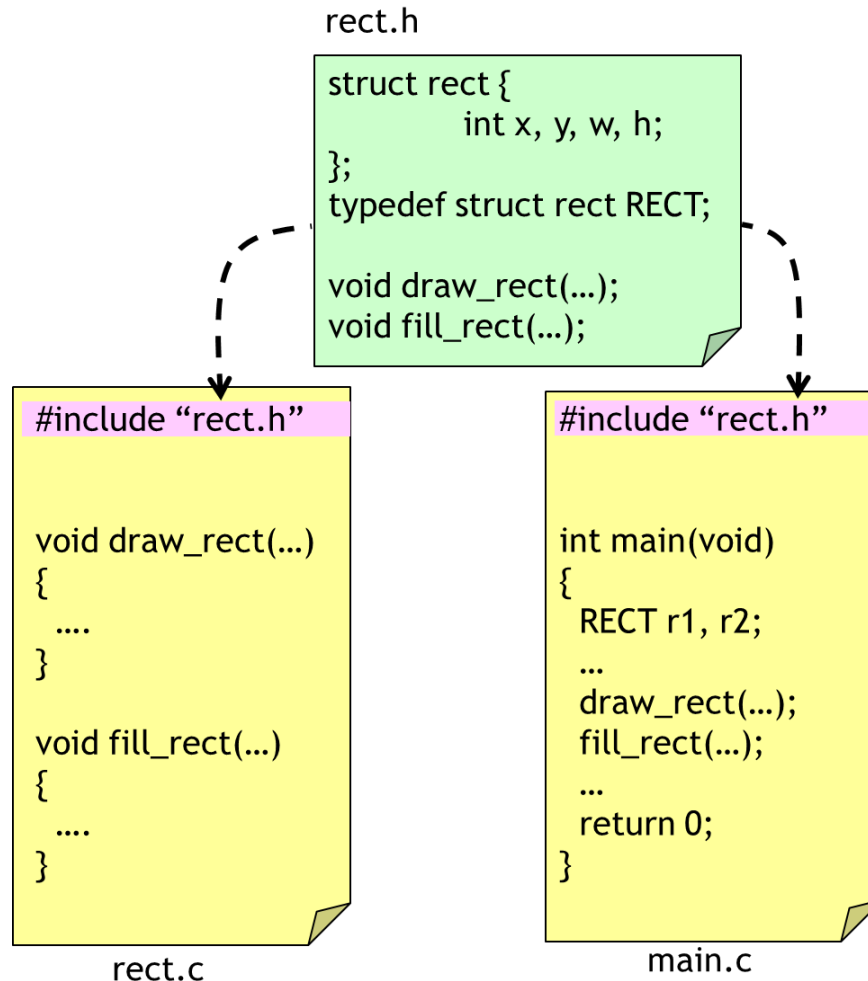
main.c

```
extern double gx, gy;

int power(void)
{
    ...

    result *= gx;

}
```

power.c

Example

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

rect.h

```
struct rect {
            int x, y, w, h;
};
typedef struct rect RECT;

void draw_rect(...);
void fill_rect(...);
```

```
#include "rect.h"


void draw_rect(...)
{
  ....
}

void fill_rect(...)
{
  ....
}
```

rect.c

```
#include "rect.h"


int main(void)
{
  RECT r1, r2;

  ...
  draw_rect(...);
  fill_rect(...);

  ...
  return 0;
}
```

main.c

rect.h

```c
#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

```c
#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

```c
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

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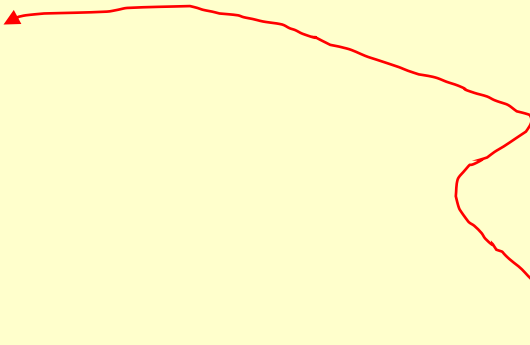calc_area ()=400.00
draw_rect (x=20, y=30, w=20, h=20)

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
```

sauce 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.
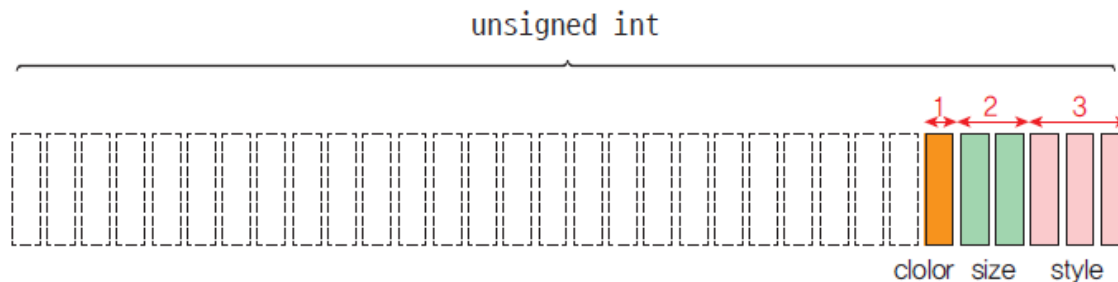
Bit field structure

- A structure whose members are divided into bit units.

```
struct Tag name {
 Data type member name 1: bit count ;
 Data type member name 2: bit count ;
 …
};
```

```
struct product {
 unsigned style : 3;
 unsigned size : 2;
 unsigned color : 1;
};
```

bit_field.c

```c
// Bit field structure
#include < stdio.h >

struct product {
    unsigned style : 3;
    unsigned size : 2;
    unsigned color : 1;
};

int main( void )
{
    struct product p1;

    p1.style = 5;
    p1.size = 3;
    p1.color = 1;

    printf ( "style=%d size=%d color=%d\n" , p1.style, p1.size, p1.color);
    printf ( " sizeof (p1)=%d\n" , sizeof (p1));
    printf ( "p1=%x\n" , p1);

    return 0;
}
```
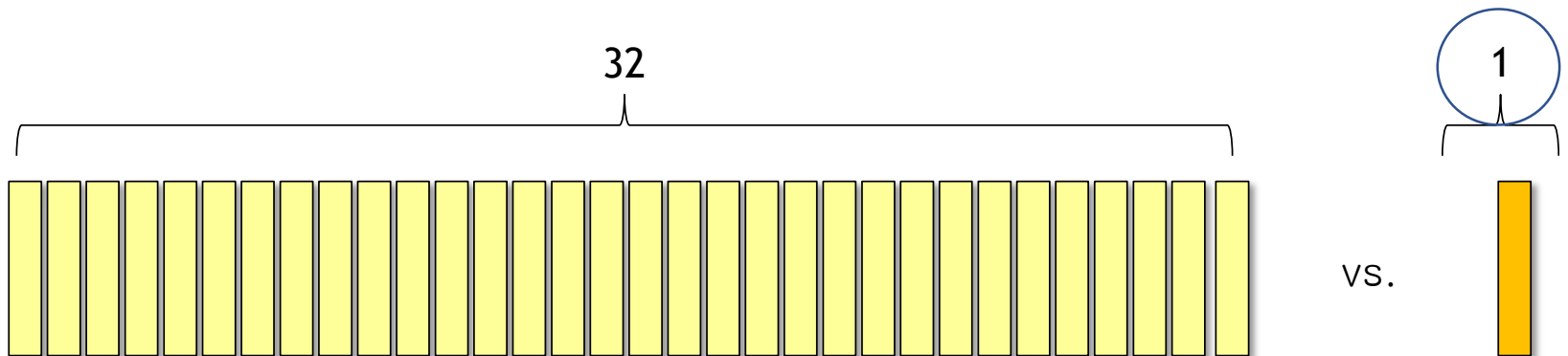
```
style=5 size=3 color=1
sizeof (p1)=4
p1= ccccccfd
```

Advantages of bit fields

- Memory is saved .
    - When storing a variable that only has a state of ON or OFF , it saves much memory to use a 1- bit bit field rather than a 32- bit int type variable .

Q & A