Advanced C Programming & Lab

15. Preprocessing and Separate Compilation

Sejong University

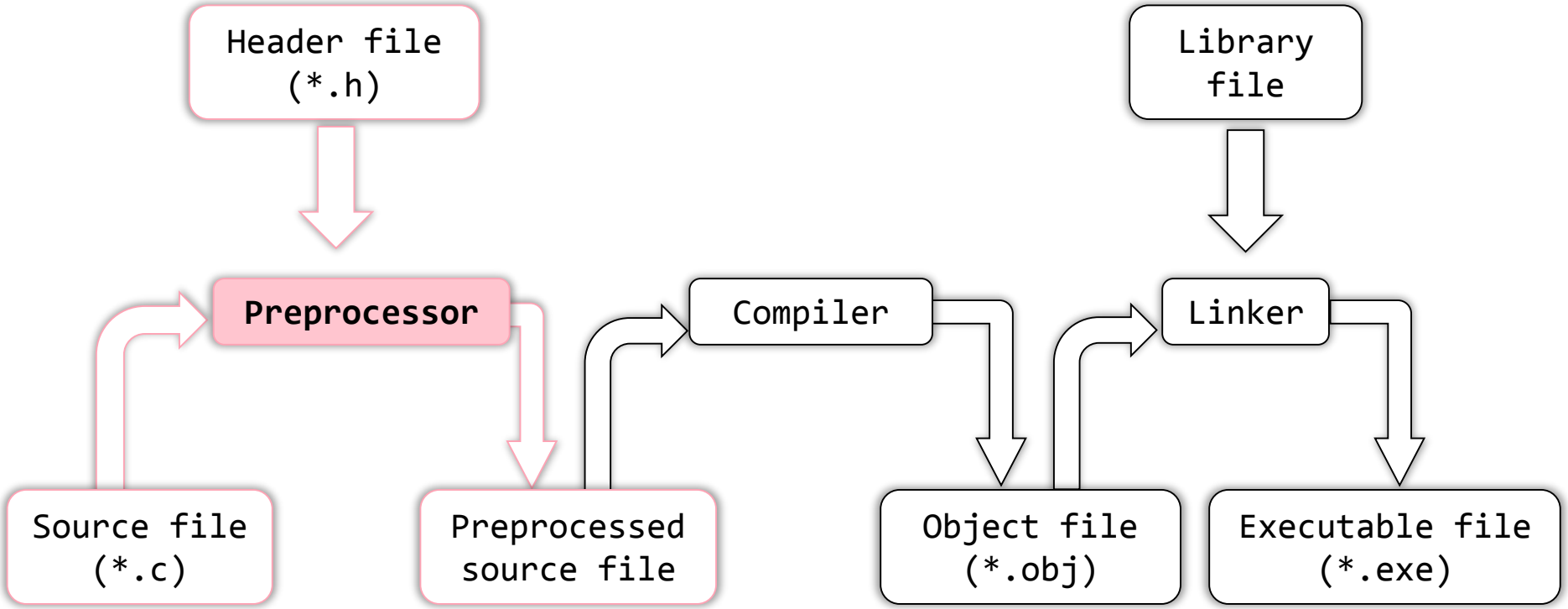
Outline

1) **Preprocessor?**
2) **Preprocessor Directives**
3) **const**
4) **Separate Compilation?**
5) **Variable range and duration**

1) Preprocessor?

- **Preprocessor?**
  - Procedures before C compiler converts the source file
- **Preprocessor directives?**
  - #include statements
  - Statements that are processed before compilation
  - Location in the beginning of a program
  - Start with # and do not use semicolon ;
- **Why?**
  - Easy to extend a program
  - Easy to maintain a program

1) Preprocessor?



Outline

1) Preprocessor ?

2) **Preprocessor Directives**

3) const

4) Separate Compilation?

5) Variable range and duration

2) Preprocessor Directives

Preprocessor Directives	Role
#include	Before compilation, include files outside a program
#define	Define macro constant/function
#undef	Cancel macro
#if ~ (#elif ~ #else ~) #endif	Conditional compilation
#ifdef ~ (#else ~) #endif	
#ifndef ~ (#else ~) #endif	

2) Preprocessor Directives: #include

- **Include particular header files**
- **Insert the designated header files**

  → Just like write the same content of the header file in a program
- **Create header files for functions and structure that are frequently used**
  - Efficient management
  - Easy to modify a program

2) Preprocessor Directives : #include

- **Method 1: Use < >**
  - Library header files that compile provides
    - ✓ Include function prototypes and structures
    - ✓ The actual code of the functions has already been compiled and included in the library files
    - ✓ Linker finds the library files
  - Search the directories that contain header files
    - ✓ Ex) C:₩Program Files (x86)₩Microsoft Visual Studio 12.0₩VC₩include
  - #include <header file name>
  - Ex) #include <stdio.h>, #include <stdlib.h> …

2) Preprocessor Directives : #include

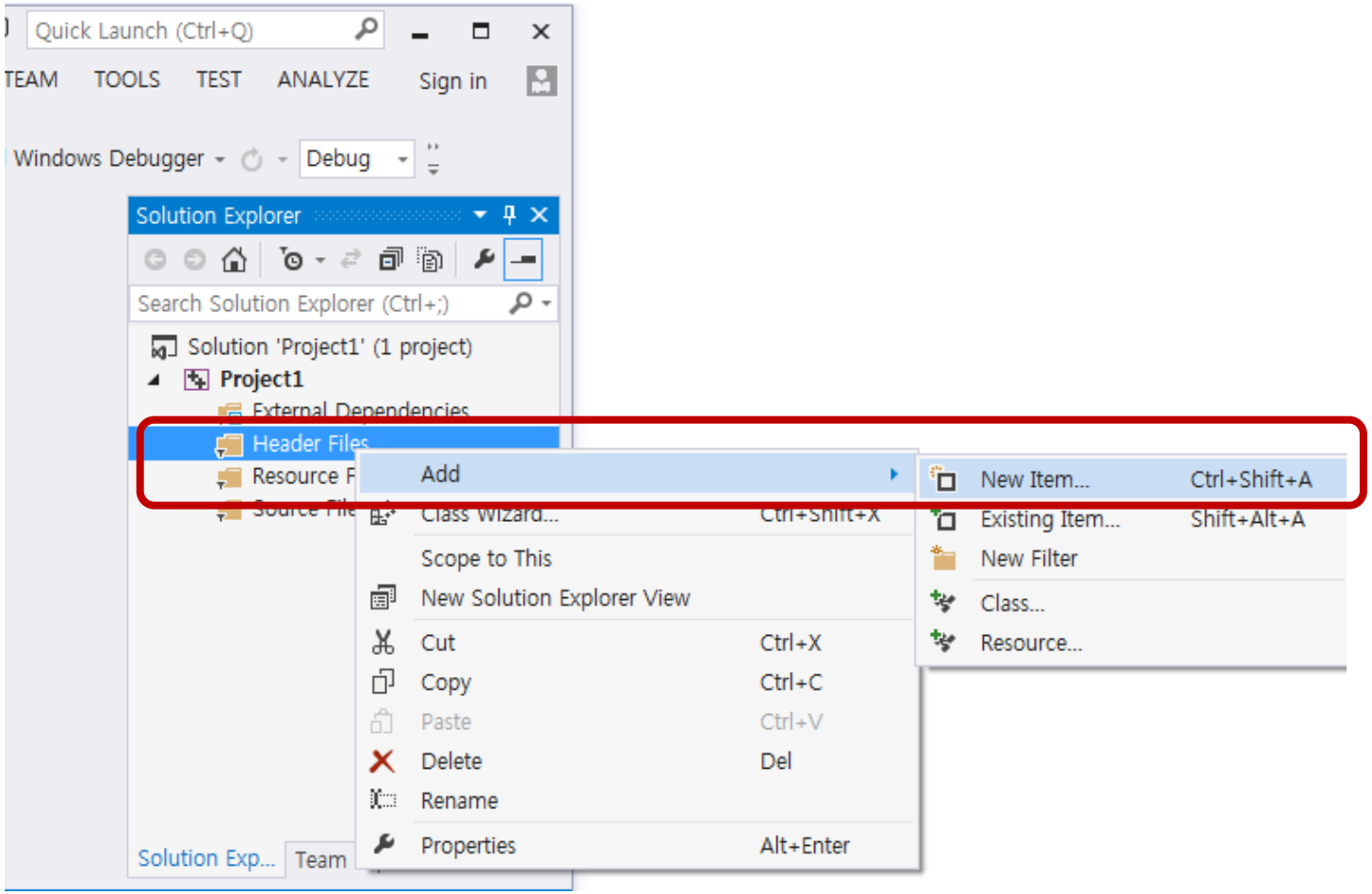
- **Method 2: Use “ “**
  - User-defined header file
  - Search the header file in the current working directory
    - ✓ Can use absolute path or relative path to include header files
    - ✓ Caution) Only use ₩
  - #include “user-defined header file name“
  - Ex) #include “myheader.h“

    #include “C:₩mywork₩project₩myheader.h” (Windows)

2) Preprocessor Directives : #include

- **Create a header file (Visual Studio)**

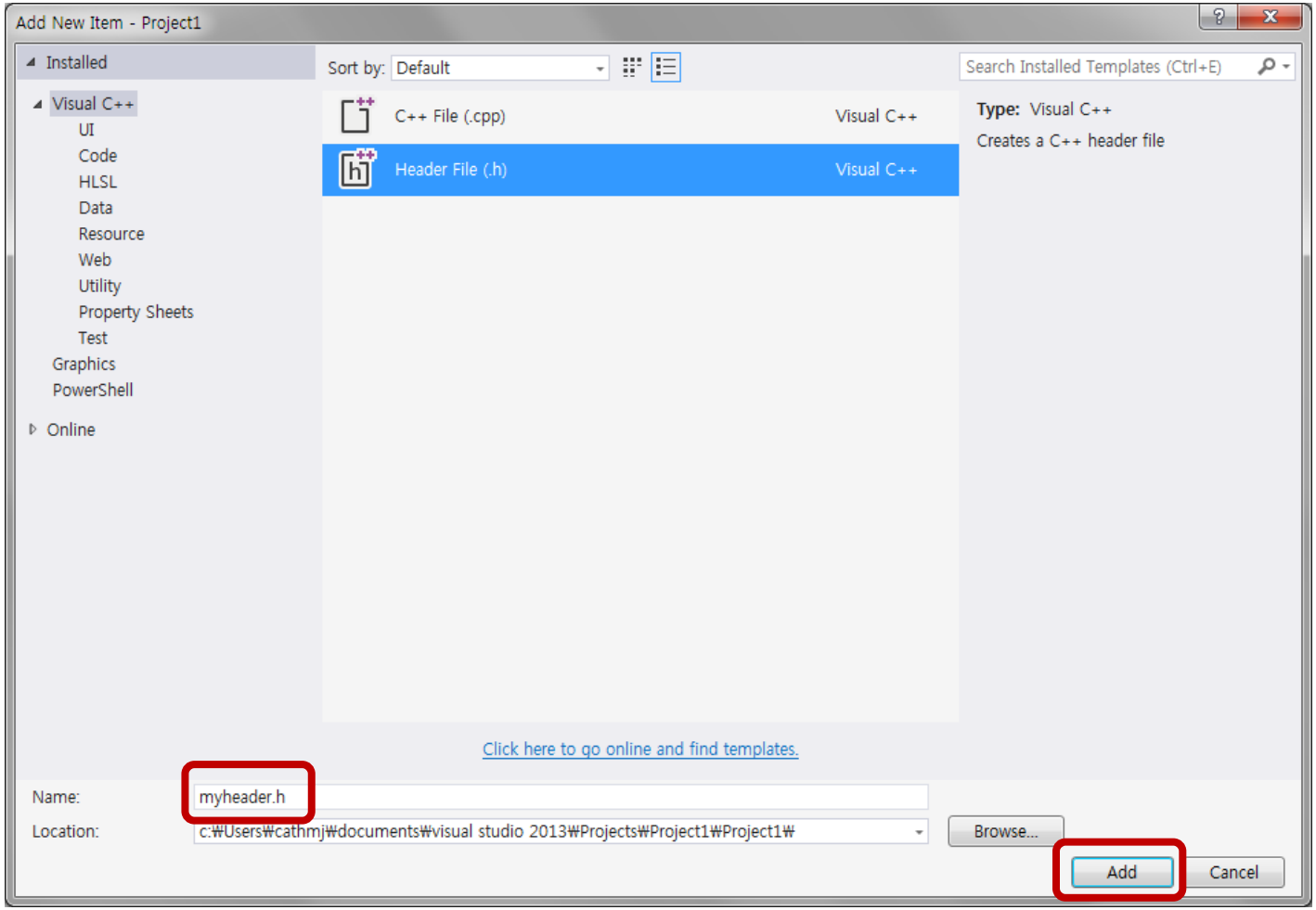
  ① Add a new item



2) Preprocessor Directives : #include

- **Create a header file (Visual Studio)**

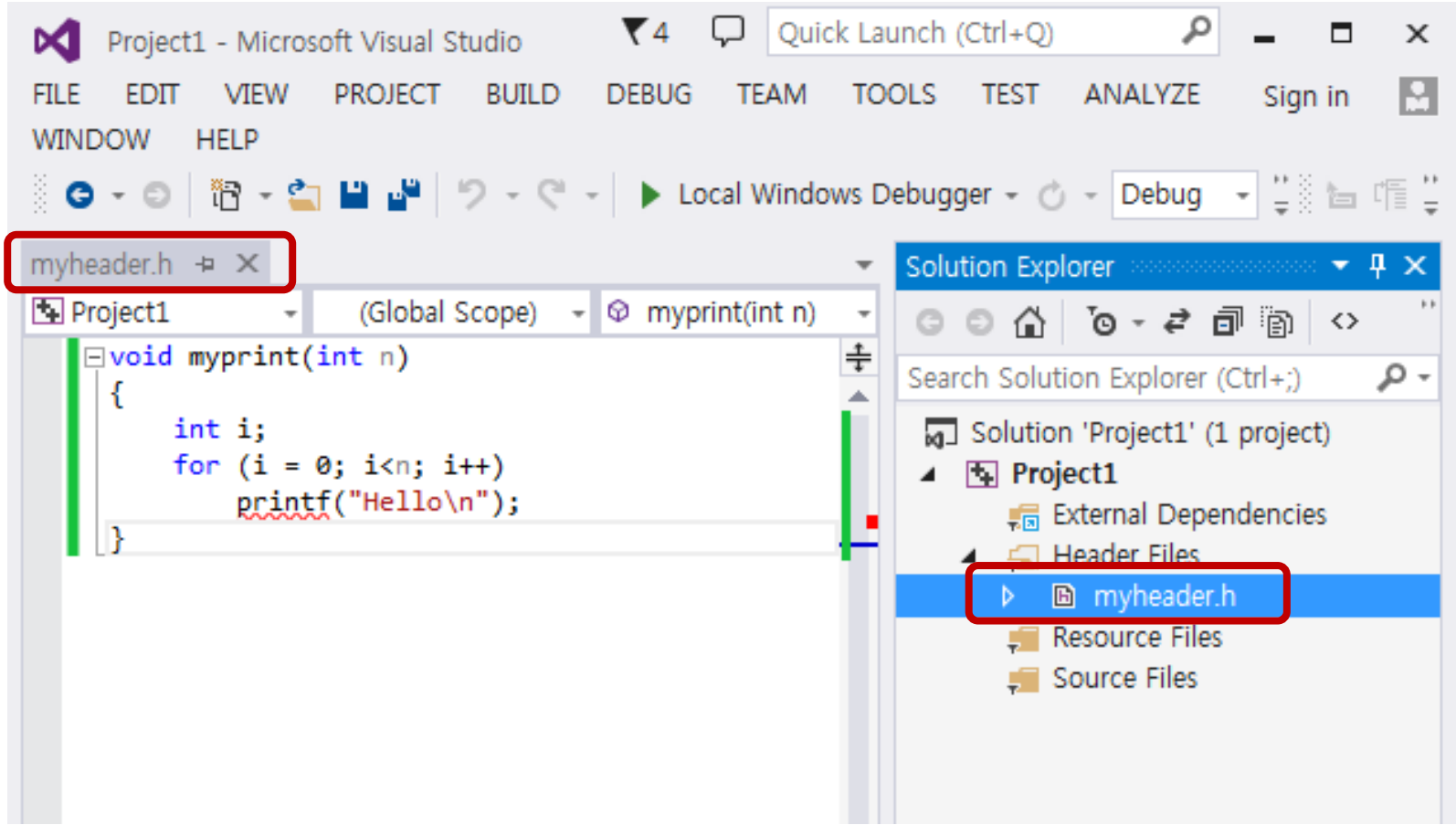
  ② Select "Header File(.h)", write the header file name, and click "Add"



2) Preprocessor Directives : #include

- **Create a header file (Visual Studio)**

  ③ Write the code



2) Preprocessor Directives : #include

- **Use the header file**
  - Print "Hello"

myheader.h

```
void myprint(int n)
{
    int i;
    for (i=0; i<n; i++)
        printf("Hello\n");
}
```

main.c

```
#include <stdio.h>
#include "myheader.h"
int main()
{
    int x;
    scanf("%d", &x);
    myprint(x);
    return 0;
}
```

2) Preprocessor Directives : #include

```
void myprint(int n)
{
    int i;
    for (i=0; i<n; i++)
        printf("Hello\n");
}
```

```
#include <stdio.h>
#include "myheader.h"
int main()
{
    int x;
    scanf("%d", &x);
    myprint(x);
    return 0;
}
```


```
#include <stdio.h>

void myprint(int n)
{
    int i;
    for (i=0; i<n; i++)
        printf("Hello\n");
}

int main()
{
    int x;
    scanf("%d", &x);
    myprint(x);
    return 0;
}
```

2) Preprocessor Directives : #include

- **Visual studio**

Project1 - Microsoft Visual Studio

FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ANALYZE Sign in
WINDOW HELP

Quick Launch (Ctrl+Q)

Local Windows Debugger | Debug

myheader.h | main.c

Project1 | (Global Scope)

```
void myprint(int n)
{
    int i;
    for (i = 0; i<n; i++)
        printf("Hello\n");
}
```

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

int main()
{
    int x;
    scanf("%d", &x);
    myprint(x);
    return 0;
}
```

Solution Explorer

Search Solution Explorer (Ctrl+;)

- Solution 'Project1' (1 project)
  - Project1
    - External Dependencies
    - Header Files
      - myheader.h
    - Resource Files
    - Source Files
      - main.c

Solution Exp... Team Explorer Class View

C:₩Windows₩system32₩cmd.exe

```
3
Hello
Hello
Hello
계속하려면 아무 키나 누르십시오 . . .
```

Item(s) Saved Ln 11 Col 1 Ch 1 INS

2) Preprocessor Directives : #define

- Define macro constant and function
  - Macro constant: For a constant that is repeatedly used
  - Macro function
    - For a program module that is repeatedly used
    - Simple functions are defined as macro functions

  ➔Easy to understand and modify a program
- Write in one line
- If too long, use ₩(backslash)

2) Preprocessor Directives : #define

- **Macro constant**
  - Preprocessor replace it by the constant
  - Format: #define macro constant name constant
    - Macro constant name: capital letters, in general
    - constant: number, character, string, system name, data type
    - Ex) #defind PI 3.14

      #define END "Terminate a program."

      #define NQ !=

2) Preprocessor Directives : #define

- **Why need them?**

```
#include <stdio.h>
int main()
{
    int r;                  //radius of a circle
    double cir;             //circumference
    double area;            //area

    r = 2;
    cir = 2 * 3.14 * r;
    area = 3.14 * r * r;

    printf("cir = %f, area = %f \n", cir, area);
    return 0;
}
```

➔ Want change PI 3.14 to 3.1415? Do it one by one

➔ Better way??

2) Preprocessor Directives : #define

- **Why need them?**

```
#include <stdio.h>

#define PI 3.14

int main()
{
    int r;                  //radius of a circle
    double cir;             //circumference
    double area;            //area

    r = 2;
    cir = 2 * PI * r;
    area = PI * r * r;

    printf("cir = %f, area = %f \n", cir, area);
    return 0;
}
```

Only need to change PI!

2) Preprocessor Directives : #define

- **Macro constant example**

```
#include <stdio.h>
#define LENGTH 50
#define EQ ==
#define STRING unsigned int
#define END “terminate program”

int main()
{
        int i, x = 0;
        STRING table[LENGTH];
        if (x EQ 0)
        {
                for (i=0; i<LENGTH; i++)
                        table[i] = i+1;
                printf(END);
        }
        return 0;
}
```

2) Preprocessor Directives : #define

- **Normal variable vs. Macro constant**

	Normal Variable	**Macro Constant**
Name	Lower-case letter, in general	Upper-case letters, in general
Assignment	Anytime	✓ Constant (not a variable) ✓ Only Initialization
Data type	Need to define	None

2) Preprocessor Directives : #define

- **Caution: macro constant**
  - Do not use semicolon (;)
  - Follow the rule to name macro constant
    - ✓ No spaces in the middle
    - ✓ Cannot begin with a number
    - ✓ Cannot use the same name
  - Can be used again
    - ✓ Same order
      - ✓ X should be defined first, then define Y

```
#define X 20
#define Y ((X)*20)
```

  - Can be empty macro
    - ✓ Format: #define macro name
      - ✓ Ex) #define EMPTY
    - ✓ Only macro constant itself
    - ✓ Use it with conditional compilation directives, in general
    - ✓ Check whether it exists or not

2) Preprocessor Directives : #define

- **Macro function**
  - Preprocessor replace it by the content of the function
  - Format: #define macro function name(argument) function definition
    - Macro function name: capital letters, in general
    - No spaces inbetween macro function name and arguments
    - definition: Use () to include arguments and statements
      - If do not use ( ) …
        - Inappropriate compilation
        - Difficult to debug

2) Preprocessor Directives : #define

```
#include <stdio.h>
#define PI 3.14                  //macro constant

#define SQUARE(x) (x * x)        //macro function

int main()
{
    double area;
    area = PI * SQUARE(4);  //area = 3.14 * (4 * 4)

    printf("area = %f \n", area);
    return 0;
}
```

What if (2+2) instead of 4?

2) Preprocessor Directives : #define

- **Result**

```
SQUARE(4): area = 50.240000
SQUARE(2 + 2): area = 25.120000
```

- Why?
  - ✓ Simply replacing the value: SQUARE(2+2) ➔ (2+2*2+2)
- How to resolve it?
  - ✓ Use ()
  - ✓ #define SQUARE(x) ((x) * (x))

    → Check the previous example!

2) Preprocessor Directives : #define

- **Macro function**

  1) Macro function name: MIN

     Macro function argument: two integers a and b

     Macro function Role: return smaller integer

  2) Macro function name : F

     Macro function argument : integer x

     Macro function Role : return x times 5

  3) Macro function name : ODD

     Macro function argument : positive integer x

     Macro function Role : return true if x is odd

2) Preprocessor Directives : #define

- **Normal function vs. Macro function**

	Normal function	**Macro function**
Name	Lower-case letters	Upper-case letters
Data type	Need to determine data type	None
Return type	Use return	none
Procedure	Do calculation	replacement
conversion	compiler	preprocessor

2) Preprocessor Directives : #define

- **Advantages and disadvantages**

Advantages	Disadvantages
✓ Easy to code ✓ Make it simple ✓ Easy to modify ✓ Faster than normal functions ✓ Do not need separate functions for differing data types	✓ Difficult to define ✓ Difficult to debug ① to prevent logical errors, use () ② do not determine data type, may encounter errors

2) Preprocessor Directives : #undef

- **Undefine (or remove) the previously created macro**
- **Can re-redefine the undefined macro**
- **Usage: #undef macroname**
- **Ex)**

```
#define DATE "Sep1st"

#undef DATE

#define DATE "Sep25th"
```

2) Preprocessor Directives : Conditional Compilation

- **Conditional Compilation?**
  - Conditionally include or exclude portions of a source file to be compiled

- **When to use?**
  - Avoid multiple inclusions of variables, functions, macro
  - Different versions of the same program

2) Preprocessor Directives : Conditional Compilation

- **Conditional compilation directives**
  - Similar to “if”: conditional statement/macroname
  - Do not use ( )
  - Do not use { }
  - Must include #endif
  - Can generate different executable files
  - #else and #elif are optional

Conditional compilation directives	**Functions**
#if ~ #endif	Depends on the value of a macro
#ifdef ~ #endif	If a macro has been defined
#ifndef ~ #endif	If a macro has not been previously defined

2) Preprocessor Directives : #if ~ #endif

- **Usage**

```
#if condition/macroname
    statement 1
#else
    statement 2
#endif
```

- condition/macroname True → statement1 compiled
- condition/macroname False → statement2 compiled
- Note) condition…
  - ✓ Cannot use real number, strings constant, variables
  - ✓ Can use operators (relational, logical, arithmetic)
  - ✓ Ex) #if SYS == 3.24 (X), #if PL == "Python" (X) …

2) Preprocessor Directives : #if ~ #endif

- **Ex 1)**

```
#define TEST 1

#if TEST
        #define ASZIE 10
#else
        #define ASZIE 10000
#endif
```

- **Ex 2)**

```
#define TESTING 1
void MyFunc();

#if TESTING
void main()
{
        MyFunc();
}
#endif

void MyFunc()
{  ……    }
```

2) Preprocessor Directives : #elif

- **elif = else if**
- **Multiple options: choose the code to be compiled**
- **Usage**

```
#if condition1
    statement 1
#elif condition2
    statement 2
#elif condition3
    statement 3
…
…
#else
    statement 4
#endif
```

2) Preprocessor Directives : #elif

- **Ex**

```
#include <stdio.h>
#define LEVEL 1

#if LEVEL == 1
        #include “beginner.h”
#elif LEVEL == 2
        #include “intermediate.h”
#elif LEVEL == 3
        #include “expert.h”
#else
        #include “general.h”
#endif
```

2) Preprocessor Directives : #ifdef ~ #endif

- **ifdef = if define**
- **If a maco has been defined**
- **Only macroname (dissimilar to #if)**
- **Usage**

```
#ifdef macroname
    statement 1
#else
    statement 2
#endif
```

  - If a macro has been defined,

    → compile statement1

  - otherwise → compile statemtn2

2) Preprocessor Directives : #ifdef ~ #endif

- **Ex**

```
#include <stdio.h>

#define LEVEL 1

int main()
{
#ifdef LEVEL
        printf("Expert!\n");
#else
        printf("Beginner!\n");
#endif

    return 0;
}
```

2) Preprocessor Directives : #ifndef ~ #endif

- **#ifndef: opposite to #ifdef**
- **If a maco has not been defind**
- **Usage**

```
#ifndef macroname
    statement 1
#else
    statement 2
#endif
```

  - If a macro has not been defined

    → compile statement1

  - Otherwise → compile statement2

2) Preprocessor Directives : #ifndef ~ #endif

- **Ex**

```
#include <stdio.h>


#ifndef PI
   #define PI 3.14
#endif


int main()
{
   printf("%f\n", PI*5.4);
   return 0;
}
```

Outline

1) Preprocessor ?

2) Preprocessor Directives

3) **const**

4) Separate Compilation?

5) Variable range and duration

3) const

- **const keyword**
  - Define a constant (not modifiable)
  - A variable name is used as a constant
  - Cannot modify the value ➔ Must declare and initialize simultaneously!
  - Usage) **const** int MAX = 100;

    MAX is 100!
  - const: better than macro
    - ✓ Compiler handles it
    - ✓ Valid within a function

3) const

- **Note) const - Pointer variables (1/3)**

  1) Declaring the value pointed by a pointer variable as const

     ✓ Ex) A function receives strings constant

  2) Declaring the value of a pointer as const

3) const

- **Note) const – pointer variables (2/3)**

1) Declaring the value pointed by a pointer variable as cons

✓ Cannot modify the value pointed by the pointer (declared as const)

```
int main()
{
        int count = 10;
        const int *p = &count;
        *p = 20;                    → Compilation Error!
        count = 1;                  → Successful!
        .........
}
```

3) const

- **Note) const – pointer variables (3/3)**

2) Declaring the value of a pointer as const

✓ Once an address is stored, not modifiable

✓ Must point one variable

```
int main()
{
    int n1 = 10;
    int n2 = 5;
    int * const p = &n1;
    p = &n2;                    → Compilation Error!
    *p = 30;                    → Successful!
    . . . . . . . .
}
```

Outline

1) Preprocessor ?
2) Preprocessor Directives
3) const
4) **Separate Compilation?**
5) Variable range and duration

4) Separate Compilation?

- **In general, one project consists of multiple modules**
  - Module: a set of functions
- **Compile each module, Link them and generate one combined program**
- **Separate compilation?**
  - Compile the source code separately (per module)
  - Develop a large program: work sharing
  - Reusability
  - Efficient management
    - ✓ Do not need to compile the entire program to modify a portion of the program

4) Separate Compilation?

- **Ex: Separate compilation "one.c"**
  - Divide into 3 files
    - ✓ main.c
    - ✓ myfunc1.c
    - ✓ myfunc2.c
  - Compile each, and generate one executable file (Linker)

    → In Visual studio, "Build"

4) Separate Compilation?

Source code (one.c)

```
#include <stdio.h>

int myfunc1(int x, int y);
void myfunc2(int n);

int main()
{
    int a, b, sum = 0;
    scanf("%d %d", &a, &b);

    sum = myfunc1(a, b);
    myfunc2(sum);

    return 0;
}

int myfunc1(int x, int y)
{
    return (x+y);
}

void myfunc2(int n)
{
    int i;
    for (i=0; i<n; i++)
        printf("%d: Good!\n", i+1);
}
```

main.c

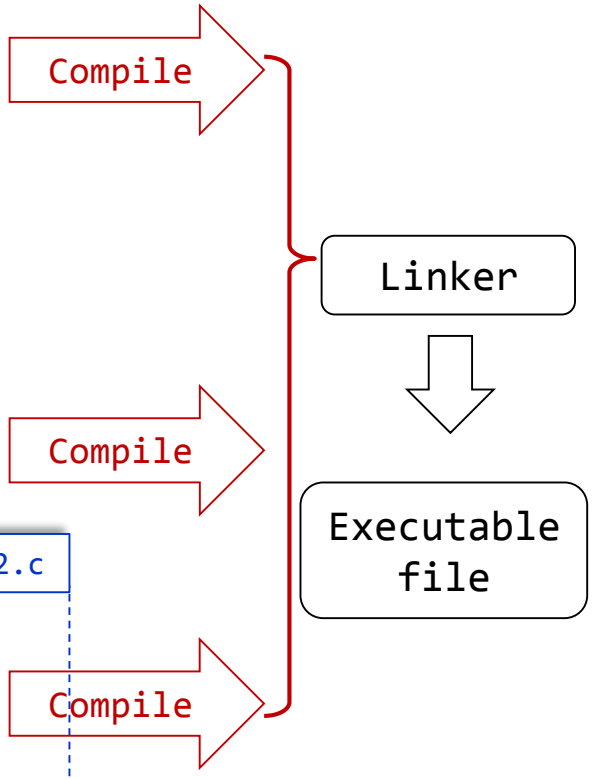
```
#include <stdio.h>
int myfunc1(int x, int y);
void myfunc2(int n);
int main()
{
    int a, b, sum = 0;
    scanf("%d %d", &a, &b);
    sum = myfunc1(a, b);
    myfunc2(sum);
    return 0;
}
```

myfunc1.c

```
int myfunc1(int x, int y)
{
    return (x+y);
}
```

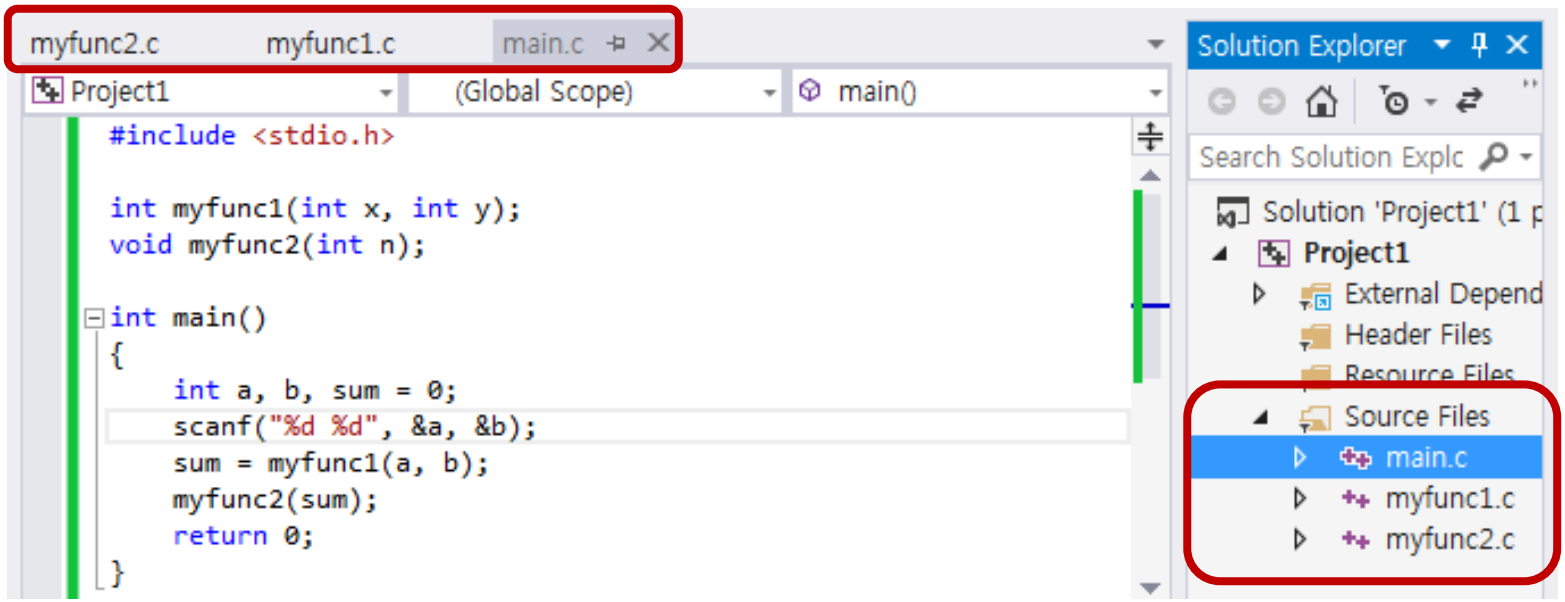
myfunc2.c

```
#include <stdio.h>
void myfunc2(int n)
{
    int i;
    for (i=0; i<n; i++)
        printf("%d: Good!\n", i+1);
}
```



4) Separate Compilation?

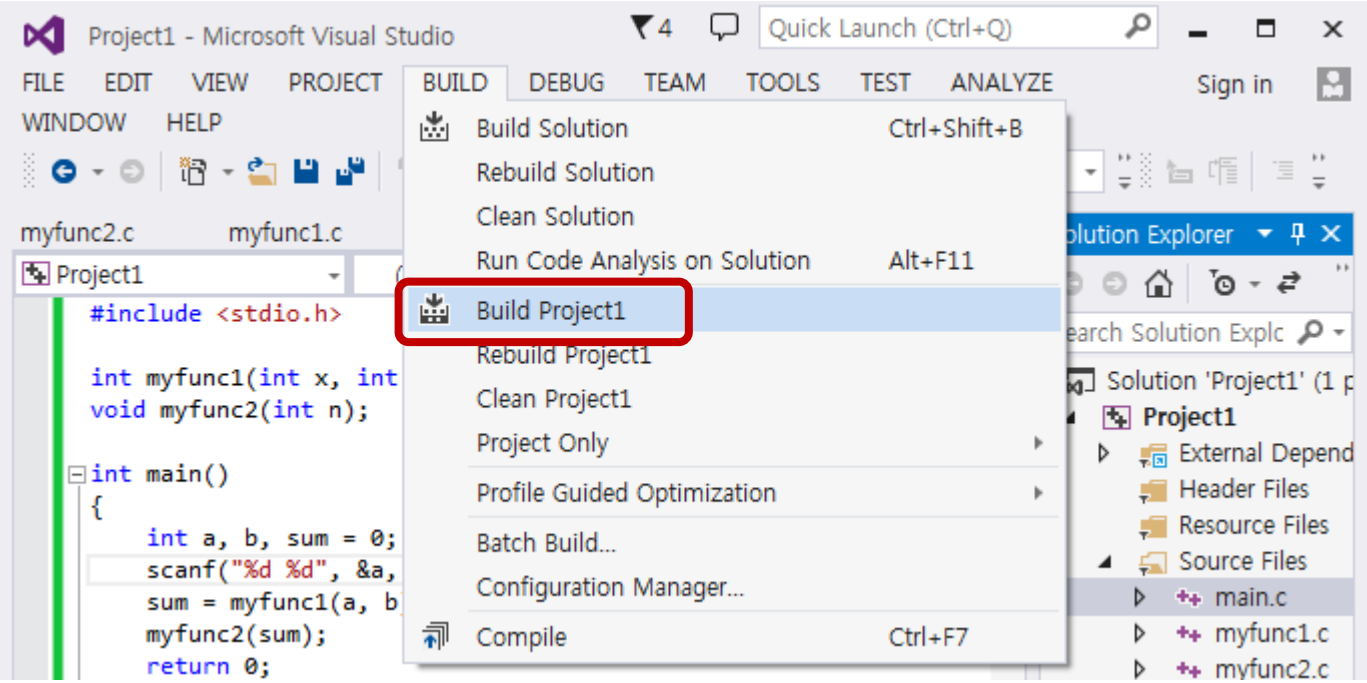
- **Visual studio**
  ① Create a project "Project1"
  ② Source file directory, create "main.c", "myfunc1.c", "myfunc2.c"
  Note) Add a file
  ① Create a file, insert a code, add to the project → add the pre-existing file
  ② In project, add a file, insert a code → create a new file



4) Separate Compilation?

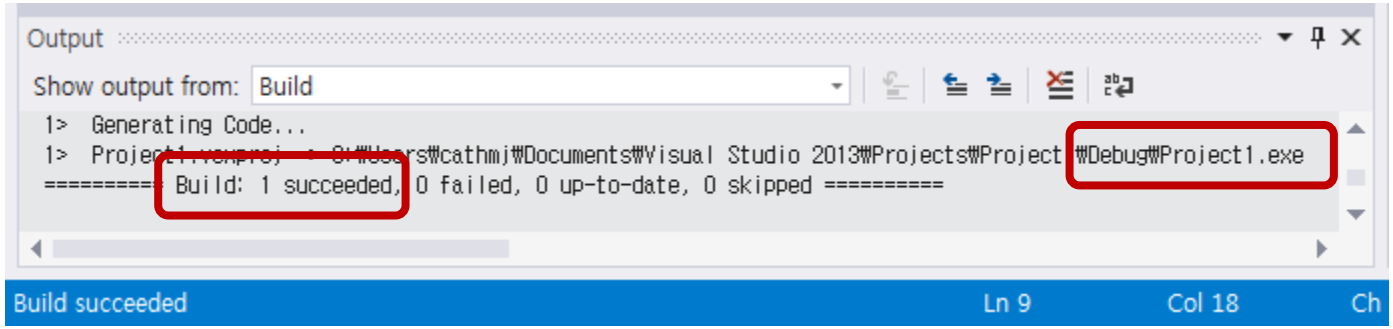
- **Visual studio**

  ③ Build Project1

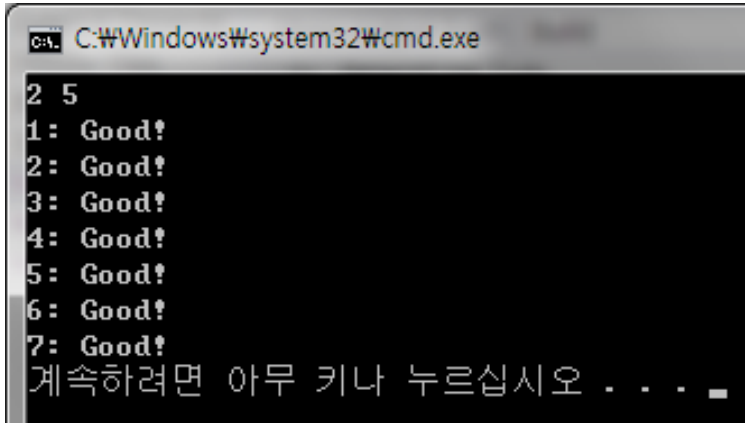


4) Separate Compilation?

- **In Visual studio**
  - Build → Project1.exe has been created



  - Result



4) Separate Compilation?

- **Divide into multiple files**
  - Variables and functions should be declared within each file
    - ✓ Each file is independently compiled
    - ✓ Compile and preprocess each file separately
- **Can we use the variables and functions declared or defined by other files?**
  - Need to know where they were declared or defined
    - ✓ Declare as extern
    - ✓ Use header files

4) Separate Compilation?

- **extern**
  - Use the variables that were declared or defined by other files
  - Let compiler know they were declared or defined by others
    - ✓ Do not need to know the specific file
  - Format
    - ✓ extern variable declaration/function declaration;
    - ✓ Function declaration, can avoid extern

4) Separate Compilation?

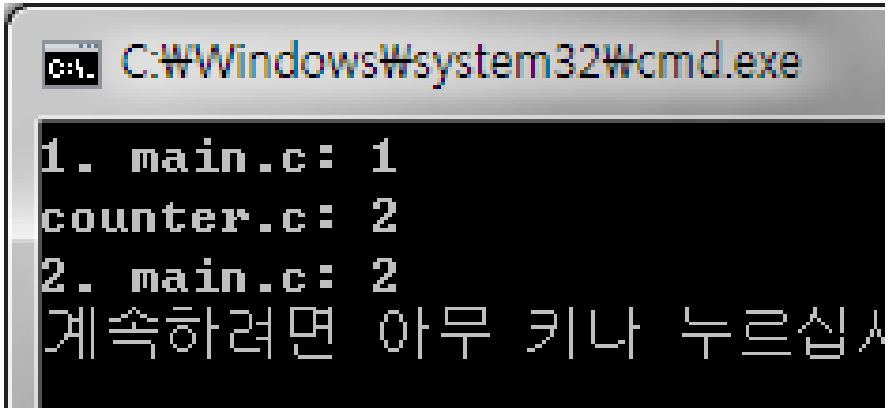
- **Ex: extern**

main.c

```
#include <stdio.h>
int num = 1;
int main()
{
    printf("1. main.c: %d\n", num);
    counter();
    printf("2. main.c: %d\n", num);
    return 0;
}
```

counter.c

```
#include <stdio.h>
extern int num;
void counter()
{
    num++;
    printf("counter.c: %d\n", num);
}
```



4) Separate Compilation?

Note) two types of static declaration!

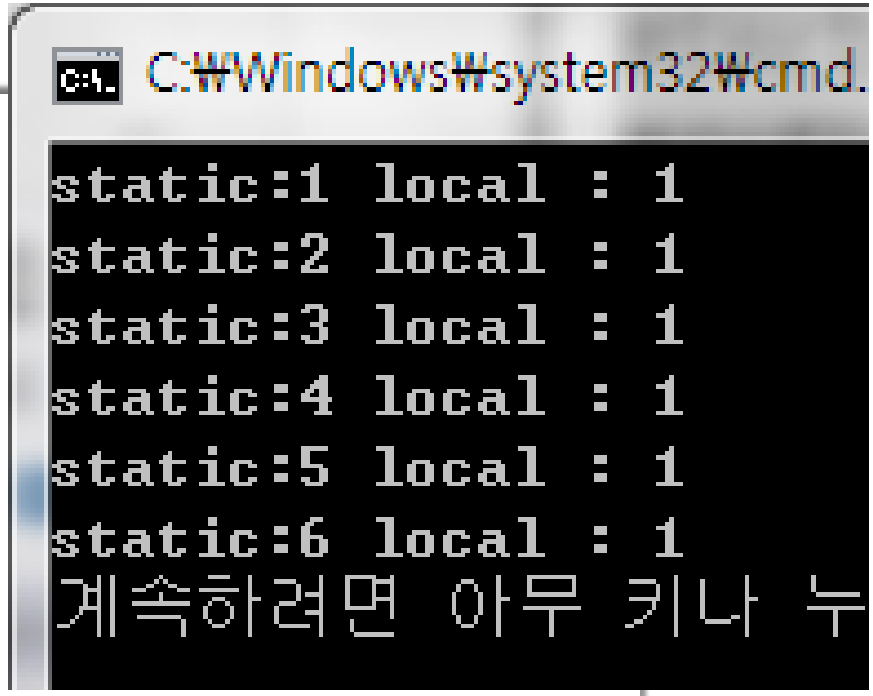
1) static local variable

- ✓ Like a global variable
- ✓ In a function, declare a static local variable…
  - ✓ Only accessible within a function (like a local variable)
  - ✓ Once initialized, exist in the memory space until program termination (like a global variable)

4) Separate Compilation?

- **Ex: static**

```
#include <stdio.h>
#define SIZE 3
void Func();
int main()
{
    int i;
    for (i=0; i<SIZE; i++)
        Func();
    return 0;
}
void Func()
{
    static int cnt1 = 0;
    int cnt2 = 0;
    cnt1++;
    cnt2++;
    printf("static:%d local:%d \n", cnt1, cnt2);
}
```



4) Separate Compilation?

Note) two types of static declaration!

2) static global variable/function

- ✓ Only valid within a file where variables/functions were declared
- ✓ Not accessible from the other files

4) Separate Compilation?

- **Ex: static global variable and extern declaration**
  - Due to static declaration of f(), only valid in func.c
  - In main.c, use exterm to call eval() defined in func.c

main.c

```c
#include <stdio.h>
extern int eval(int n);

int main()
{
    printf("%d\n", eval(5));
    return 0;
}
```

func.c

```c
static int f(int x)
{
    return x*4+3;
}

int eval(int n)
{
    return 3*f(n);
}
```

4) Separate Compilation?

- **To access and call the variables and functions from outside, store variable and function declarations in header files**
- **Contents in a header file (.h)**
  - Constants, struct definition, function prototype, extern variables, macro that are often used
- **Contents in a source file(.c)**
  - Global variables, function body
- **Header file: duplicate insertion**
  - Especially, struct definition → compilation error
  - Solution? Conditional compilation

4) Separate Compilation?

- **Header file: duplicate insertion**
  - Separate compilation in Visual studio?

myheader.h

```c
#include <stdio.h>
#define SIZE 3

struct student{
   int id;
   double score;
};
extern struct student st[SIZE];
```

main.c

```c
#include "myheader.h"
#include "myfunc.h"
struct student st[SIZE];

int main()
{
    int i;
    for (i=0; i<SIZE; i++)
       scanf("%d %lf", &st[i].id, &st[i].score);

    eval(st);

    return 0;
}
```

myfunc.h

```c
#include "myheader.h"

void eval(struct student *st1);
```

myfunc1.c

```c
#include "myheader.h"
void eval(struct student *st1)
{
    int i;
    double sum = 0.0;
    for (i=0; i<SIZE; i++)
        sum += st1[i].score;

    printf("%lf\n", sum/SIZE);
}
```

4) Separate Compilation?

- **Header file: duplicate insertion**

Project1 - Microsoft Visual Studio

myfunc.h* myfunc1.c myheader.h main.c

Project1 student

```
#include <stdio.h>
#define SIZE 3

struct student{
    int id;
    double score;
};
struct student st[SIZE];
```

Error List

1 Error 0 Warnings 0 Messages

	Description	File	Line	Column	Project
1	error C2011: 'student' : 'struct' type redefinition	myheader.h	4	1	Project1

Solution Explorer

Solution 'Project1' (1 pr

- Project1
  - External Depende
  - Header Files
    - myfunc.h
    - myheader.h
  - Resource Files
  - Source Files
    - main.c
    - myfunc1.c

Ready Ln 4 Col 16 Ch 16 INS

4) Separate Compilation?

- **Use conditional compilation**
  - Use #ifndef ~ #endif
  - In each header file, use #ifndef ~ #endif
  - Usage

```
#ifndef headerfilename
#define headerfilename

extern int num;
void print(int x);
…

#endif
```

4) Separate Compilation?

- **Header file: duplicate insertion (1/3)**

myheader.h

```
#ifndef __MYHEADER_H__
#define __MYHEADER_H__

#include <stdio.h>
#define SIZE 3

struct student{
   int id;
   double score;
};
struct student st[SIZE];

#endif
```

myfunc.h

```
#ifndef __MYFUNC_H__
#define __MYFUNC_H__

#include “myheader.h”

void eval(struct student *st1);

#endif
```

4) Separate Compilation?

- **Header file: duplicate insertion (2/3)**

Project1 - Microsoft Visual Studio

myfunc.h | myfunc1.c | myheader.h | main.c

```
#ifndef __MYHEADER_H__
#define __MYHEADER_H__

#include <stdio.h>
#define SIZE 3

struct student{
    int id;
    double score;
};
struct student st[SIZE];

#endif
```

Output

Show output from: Build

```
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

4) Separate Compilation?

- **Header file: duplicate insertion (3/3)**

myheader.h

```
#ifndef __MYHEADER_H__
#define __MYHEADER_H__

#include <stdio.h>
#define SIZE 3
struct student{
   int id;
   double score;
};
struct student st[SIZE];

#endif
```

main.c

```
① #include "myheader.h"
② #include "myfunc.h"
int main()
{
    int i;
    for (i=0; i<SIZE; i++)
       scanf("%d %lf",
              &st[i].id, &st[i].score);

    eval(st);
    return 0;
}
```

①

__MYHEADER_H__
has been defined

myfunc.h

```
#ifndef __MYFUNC_H__
#define __MYFUNC_H__

#include "myheader.h"
void eval(struct student *st1);

#endif
```

②

__MYHEADER_H__ has already been defined
Do not conduct #include "myheader.h"!

Outline

1) Preprocessor ?
2) Preprocessor Directives
3) const
4) Separate Compilation?
5) **Variable range and duration**

5) Variable range and duration

- **Variable range (Ref. function)**
  - Global variable
    - ✓ count
  - Local variable
    - ✓ sum, i, j

test.c

```
. . . . . .
int count = 10;
. . . . . .
int main()
{
    int sum = 0;
    . . . . . . . .
    if (sum < count + 3)
    {
        int i = 2;
        sum += i;
        . . . . . . . .
    }
    else
    {
        int j = 5;
        sum -= j;
        . . . . . . . .
    }
    . . . . . . . .
}
```

5) Variable range and duration

- **Variable range**

Type	Global	Local
Declaration	Outside a function	Within a function/block
Range	Within a program	Within a function/block
Duration	During program execution	During function/block execution
Access from other functions within the same source file	O	X
Access from other functions in a different source file	O (Use extern keyword)	X
Without initialization	Depending on data types, 0, '₩0', or NULL	Garbage value

5) Variable range and duration

- **Variable duration**
  - External variable(global variable)
  - Automatic variable(local variable)
  - Static variable

5) Variable range and duration

- **Variable range and duration in 3 separate files**

main.c

```
. . . . . .
int global1;
extern func2();

int main()
{
    int local1;
    func2();
    . . . . .
}
```

func1.c

```
. . . . . .
extern int global1;

static int func1()
{
    int local2;
    . . . . .
}
```

func1(): can call it only within func1.c

func2.c

global2: accessible only within func2.c

```
. . . . . .
static int global2;

int func2()
{
    static int local3;
    . . . . .
}
```

local3: accessible only within func2()

External | Automatic | Static

5) Variable range and duration

- **Variable range**

<table>
<tr><th colspan="2">Type</th><th>Keyword</th><th>Range</th><th>Duration</th><th>default value</th><th>Access from other functions within the same file</th><th>Access from other functions from other files</th></tr>
<tr><td colspan="2">External</td><td>extern</td><td>Within a program</td><td rowspan="3">Program execution</td><td rowspan="3">0/ '₩0'/NULL (depending on data types)</td><td>O</td><td>O</td></tr>
<tr><td rowspan="2">Static</td><td>Global</td><td rowspan="2">static</td><td>Within a file</td><td>O</td><td>X</td></tr>
<tr><td>Local</td><td rowspan="2">Within function/ Block</td><td>X</td><td>X</td></tr>
<tr><td colspan="2">Automatic</td><td>-</td><td>Function/block execution</td><td>Garbage</td><td>X</td><td>X</td></tr>
</table>