

4. Functions

[ECE10002] C Programming

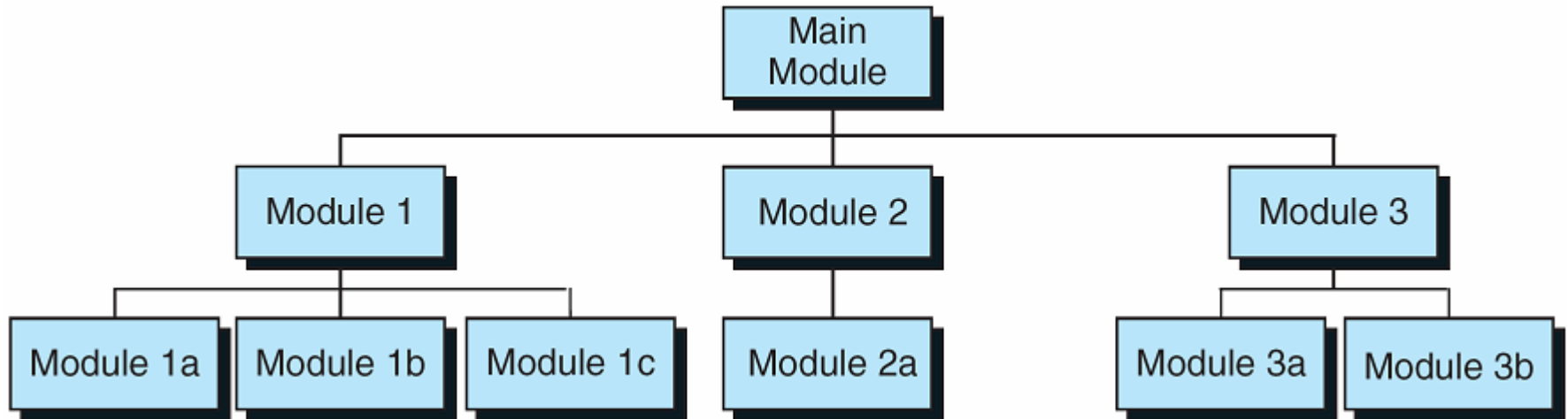
Agenda



- Designing Structured Programs
- Functions in C
- Inter-function Communication
- Standard Functions
- Scope

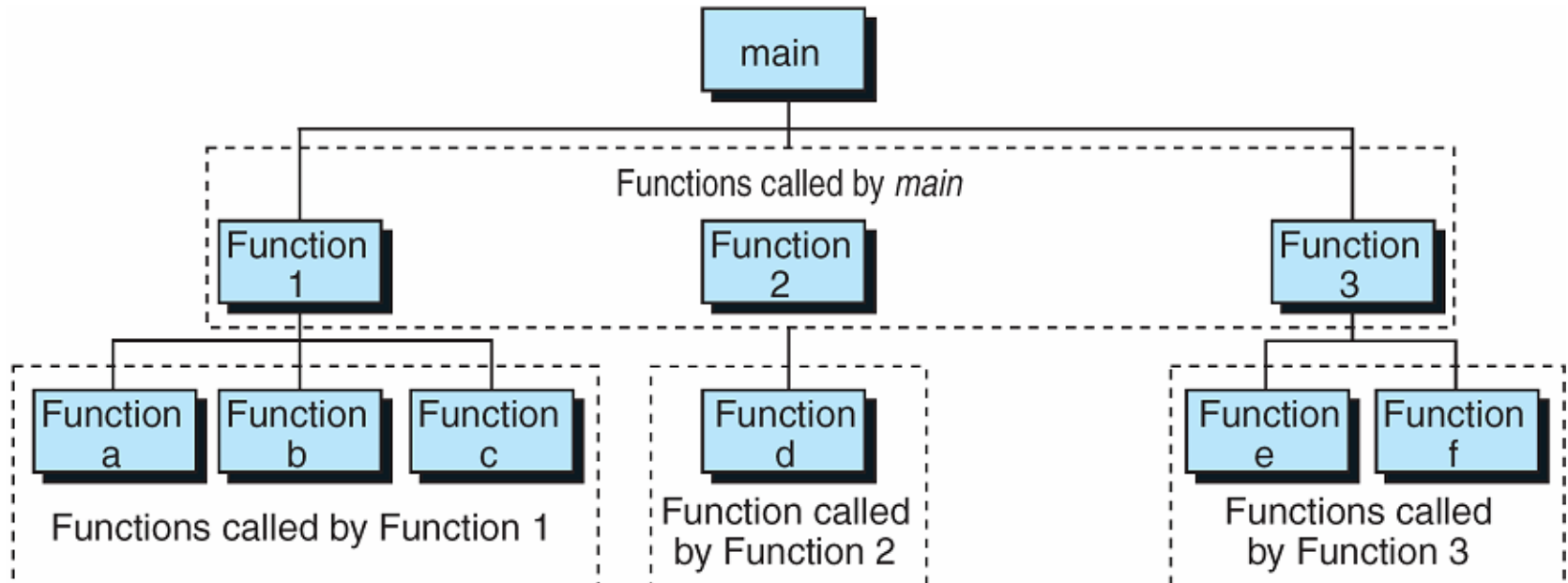
Designing Structured Programs

- **Top-down approach** for a complex problem
 1. Understand the problem as a whole
 2. Break it into simpler understandable parts
 3. Write subprograms for each of broken parts (**module**)



Functions in C

- C program is made of one or more functions
 - Idea of top-down design is supported by functions.
 - Each function can **call** other functions.
 - A program should have an **entry function**, “**main**”.
 - Every program starts from main function



Example: Elephant.c



```
// This program prints the instructions to  
    put an elephant into a refrigerator
```

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

```
// function declarations
```

```
void OpenDoor();
```

```
void PushElephantIntoRefrigerator();
```

```
void CloseDoor();
```

```
int main()  
{
```

```
    // function calls
```

```
    OpenDoor();
```

```
    PushElephantIntoRefrigerator();
```

```
    CloseDoor();
```

```
    return 0;
```

```
}
```

```
// function definitions
```

```
void OpenDoor()
```

```
{
```

```
    printf("Open the door.\n");
```

```
}
```

```
void PushElephantIntoRefrigerator()
```

```
{
```

```
    printf("Push the elephant into the  
    refrigerator.\n");
```

```
}
```

```
void CloseDoor()
```

```
{
```

```
    printf("Close the door.\n");
```

```
}
```

Example: Circle.c



```
#include <stdio.h>

#define PI 3.141592F

// function declarations
float GetCircleSize(float radius);
float GetCircleCircumstance(float radius);

int main()
{
    float r = 0.F, s = 0.F, c = 0.F;

    printf("Input radius of a circle : ");
    scanf("%f", &r);

    s = GetCircleSize(r);
    c = GetCircleCircumstance(r);

    printf("radius = %.2f\n", r);
    printf("size = %.2f\n", s);
    printf("circumstance = %.2f\n", c);

    return 0;
}

// function definitions
float GetCircleSize(float radius)
{
    float size = radius * radius * PI;

    return size;
}

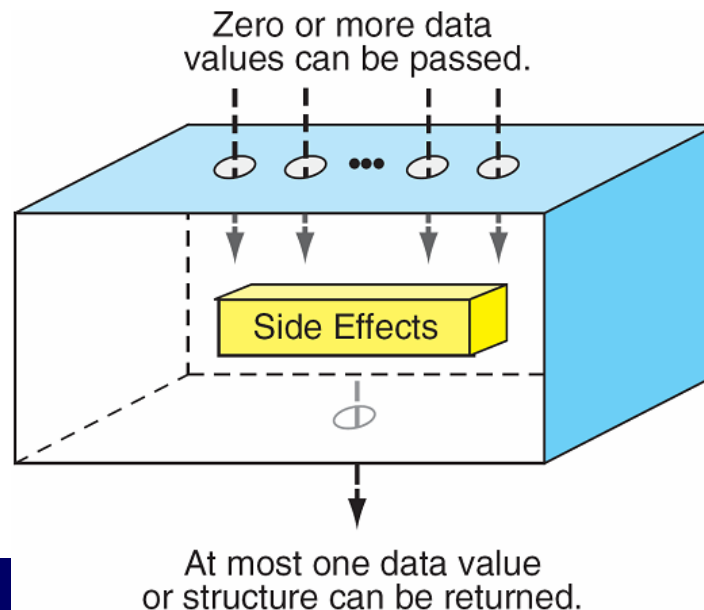
float GetCircleCircumstance(float radius)
{
    float circumference = 2 * PI * radius;

    return circumference;
}
```

Concept of Function

■ What a function does?

1. Receive zero or more pieces of data (parameters)
2. Perform some actions
 - Operate on the parameters
 - Additional actions (side effect)
3. Return at most one piece of data



Functions in C



■ Using functions

- Function definition
- Function call
- Function declaration

■ More about functions

- Parameter passing
- Return value
- Bi-directional communication

Function Definition

- Function can be defined by **header** and **body**
 - **Header**: specification for return type, function name, formal parameters
 - **Body**: program codes to be executed
 - Consists of **local declarations** and **statements**

Function Header

```
return_type function_name (formal parameter list)
```

```
{  
  // Local Declarations  
  ...  
  // Statements  
  ...  
} // function_name
```

Function Body

```
// an example of function def.  
void greetings()  
{  
    // no local declarations  
  
    // statements  
    printf("Hello, World!\n");  
}
```

Function Call

■ Function call (invocation)

- Called function receives **execution control** from calling function
- After execution, called function **returns** control to the calling function

```
#include <stdio.h>
void greetings();    // declaration
```

```
int main()
{
    // local declarations

    // statements
    ...
    greetings();    // function call
    ...

    return 0;
}
```

```
// function definition
void greetings()
{
    // no local declarations

    // statements
    printf("Hello, World!\n");
}
```

Function with Parameters

- **Parameters (arguments)**: information passed from calling function to called function

```
#include <stdio.h>
void Report(int num1, int num2, int sum);
int main()
{
    // local declarations
    int a = 10, b = 20;
    int c = 0;

    // statements
    c = a + b;
    Report(a, b, c);

    return 0;
}
```

num1 = a;
num2 = b;
sum = c;

// function definition

```
void Report(int num1, int num2, int sum)
{
    // no local declarations

    // statements
    printf("%d + %d = %d\n",
           num1, num2, sum);
}
```

formal parameter list

Calling A Function with Parameters



■ Syntax of function call

- function_name (**actual_parameter_list**);
 - Actual parameter list: list of values (or expressions) to send to called function

multiply (6, 7)

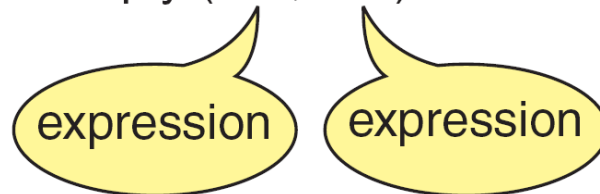
multiply (6, b)

multiply (multiply (a, b), 7)

multiply (a, 7)

multiply (a + 6, 7)

multiply (... , ...)



Formal Parameter and Actual Parameter

- Formal parameters: **variables** declared in function header
- Actual parameters: **values (or expressions)** in calling statement
- Formal and actual parameters must **match exactly** in **type**, **order** and **number**.
- Value of an actual parameter is **copied** to the corresponding formal parameters

```
#include <stdio.h>
void Report(int num1, int num2, int sum);
int main()
{
    // local declarations
    int a = 10, b = 20;
    Report(a, b, a + b);    // function call

    return 0;
}
```

Actual parameters

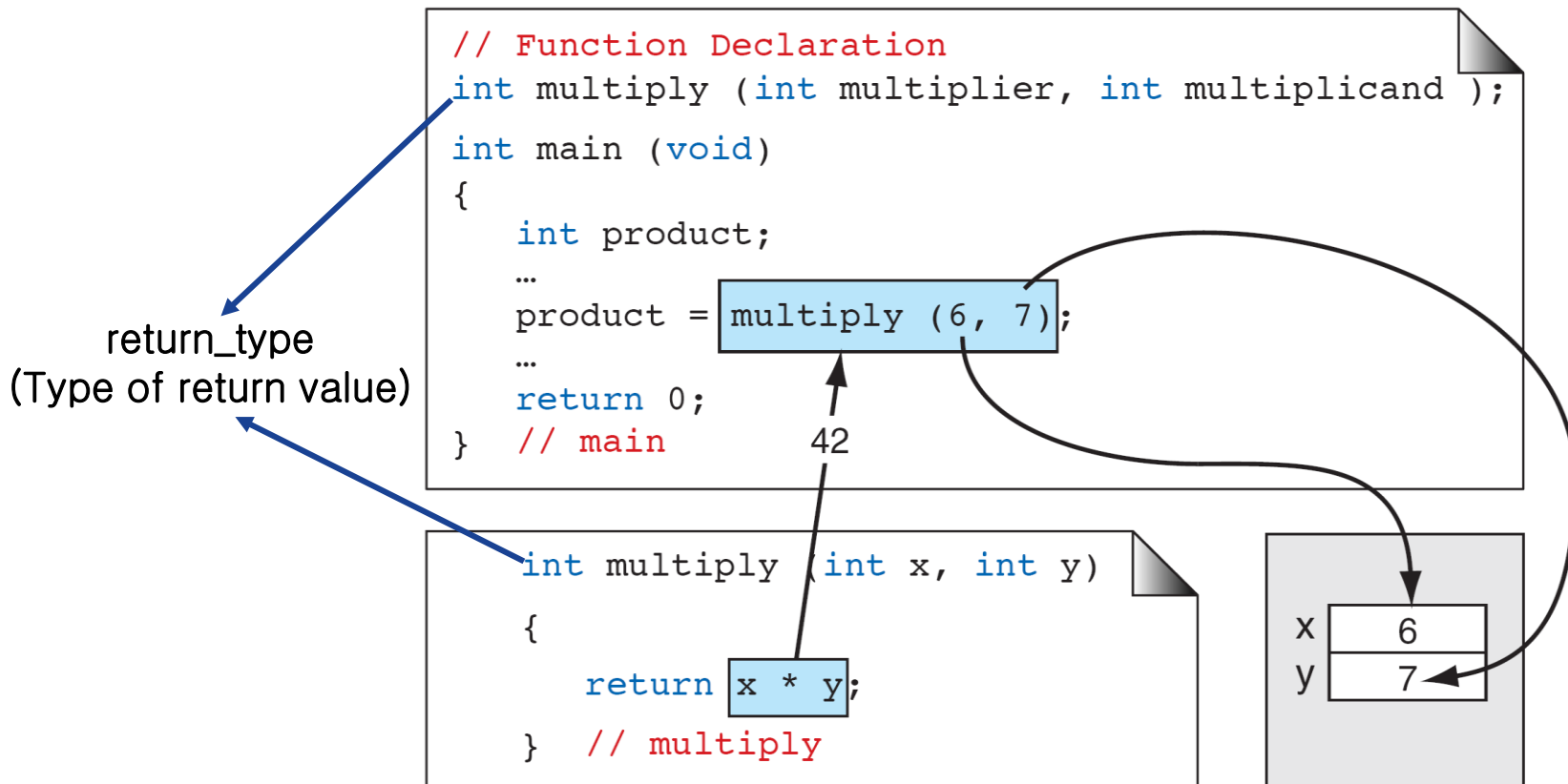
Formal parameters

```
// function definition
void Report(int num1, int num2, int sum)
{
    // no local declarations

    // statements
    printf("%d + %d = %d\n",
           num1, num2, sum);
}
```

Function With Return Value

- **Return value:** information passed from called function to calling function



Function Declaration

- Syntax of function declaration is similar to function header but...
 - Terminates with semicolon
 - Identifier names for parameters can be omitted

Ex) `int Multiply(int, int);` // also OK, but not desirable

```
#include <stdio.h>
int Multiply(int num1, int num2);    // declaration
int main()
{
    int a = 10, b = 20;

    printf("%d * %d = %d\n", a, b, Multiply(a, b));

    return 0;
}

// definition of Multiply
int Multiply(int num1, int num2)
{
    return num1 * num2;
}
```

Example: Print With Comma



- Print a number with comma (Ex: 123456 → 123,456)

```
#include <stdio.h>

void printWithComma (long num);

int main (void)
{
    long number = 0;

    printf("\nEnter a number with up to 6 digits: ");
    scanf ("%ld", &number);
    printWithComma (number);

    return 0;
} // main
```

```
void printWithComma (long num)
{
    int thousands = 0;
    int hundreds = 0;

    thousands = num / 1000;
    hundreds = num % 1000;

    printf("\nThe number you entered is %03d,%03d",
           thousands, hundreds);
    return;
} // printWithComma
```


Two Aspects of Functions in C



■ Mapping from parameters to return value

- `double sin(double x);` // declared in `math.h`
- `double gaussian(double x);` // user-defined function
- `char LowerToUpper(char c);` // ex) 'a' → 'A'
- ETC.

■ Subroutine

- Subprogram (module) to perform a subtask
- Side effect or main job of a function?
- Return value contains any information from called function
Ex) error code ...

➔ C functions have both aspects

Why Function?



- Advantages of using function
 - Problem factoring
 - Code reuse
 - System library functions
 - Ex) standard I/O function (stdio.h),
math functions (math.h)
 - Protect data
 - Local variable

Agenda



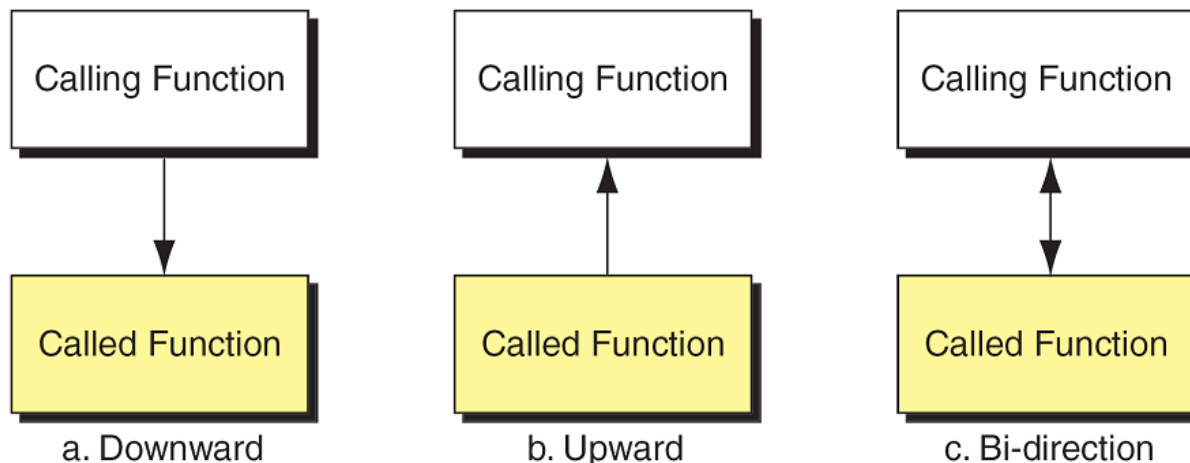
- Designing Structured Programs
- Functions in C
- Inter-function Communication
- Standard Functions
- Scope

Inter-Function Communication

■ Types of inter-function communication

- Downward communication: parameters
- Upward communication: return value
- Bi-directional communication: pointers

Ex) Modifying a variable in calling function from called function



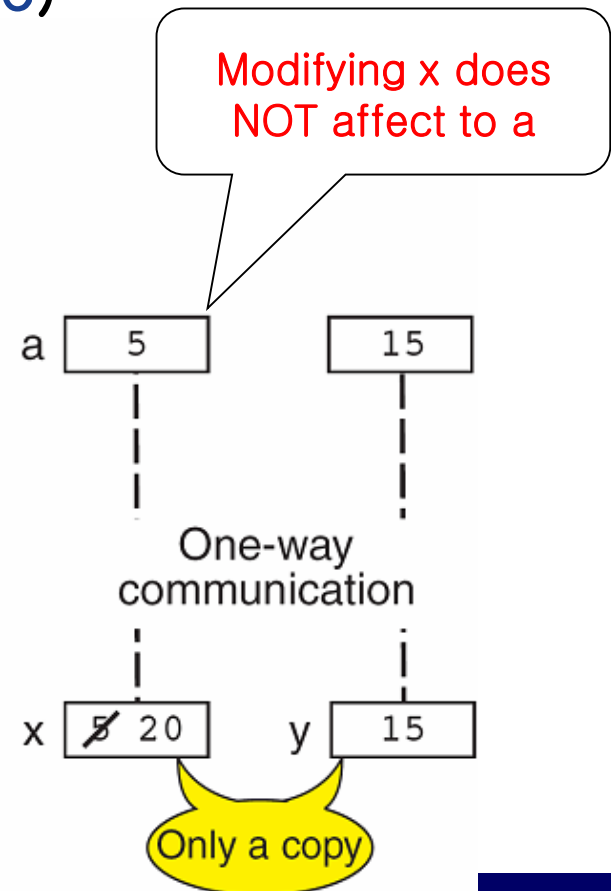
Downward Communication

- In C function call, actual parameters are **copied** to formal parameters (**Call-by-value**)

```
// Function Declaration
void downFun (int x, int y);
int main (void)
{
    // Local Definitions
    int a = 5;
    // Statements
    downFun (a, 15);
    printf ("%d\n", a);
    return 0;
} // main
```

prints 5

```
void downFun (int x, int y)
{
    // Statements
    x = x + y;
    return;
} // downFun
```



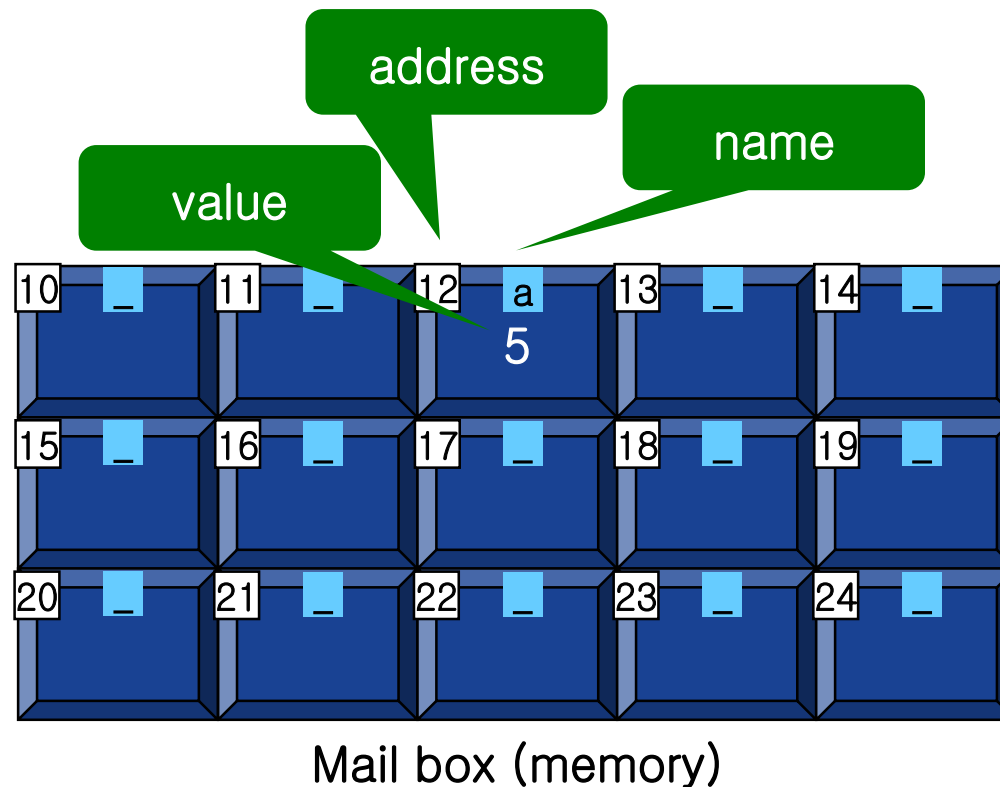
Exercises



- Read two integers a , b . Then, exchange the two integers. Write a function `swap(x, y)`.
- Read three integers a , b , and c . Then, shift right the three integers. Write a function `shift3(x, y, z)`.

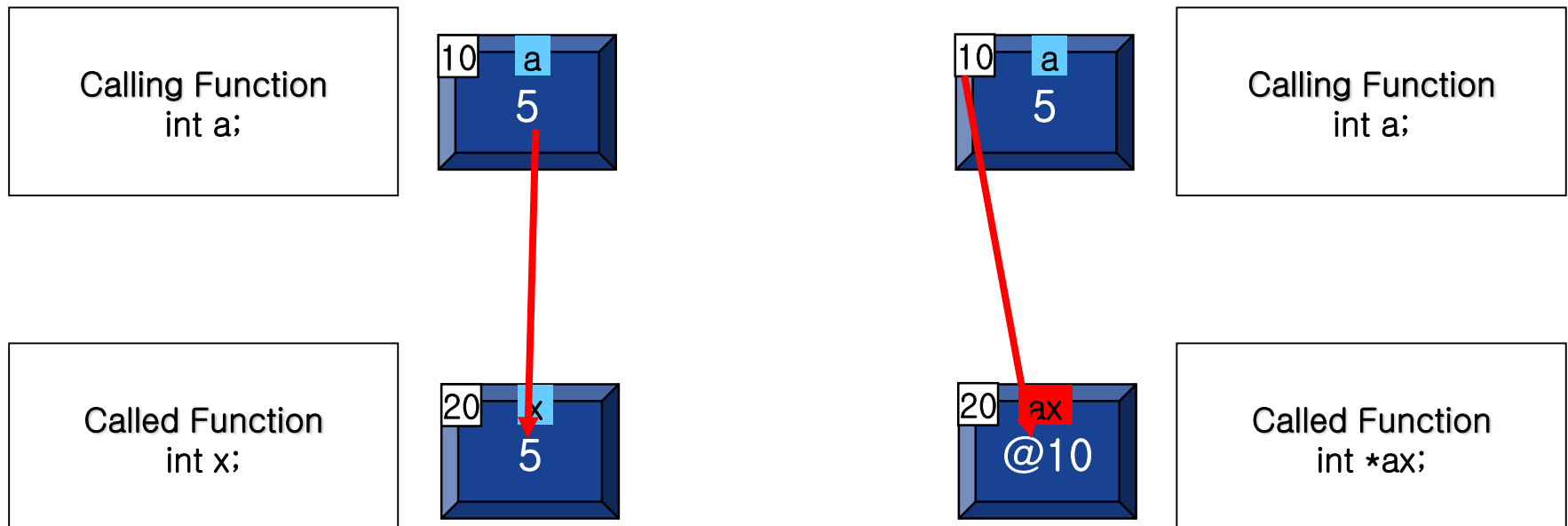
Bi-Directional Communication

- Three aspects of a variable: name, value, **address**
 - **Address**: location of variable in memory



Bi-Directional Communication

- Bi-directional communication by passing address
 - Calling function does not send value of a variable, but **address** of it
 - Called function receives the address using **pointer variable**
 - **Pointer variable**: variables to store **address** of other variables



Pointer Variables

- To modify value of a variable in calling function, we need ...
 1. To extract the address of a variable.
 2. To use a pointer variable as a parameter.
 3. To access value pointed by a pointer variable.

```
int main (void)
{
    int a;
    int b;
    ...
    upFun (&a, &b);
    ...
} // main
```

1

```
void upFun (int* ax, int* ay)
{
    *ax = 23;
    *ay = 8;
    return;
} // upFun
```

2

3

Bi-Directional Communication



- Extracting address of a variable: **address operator &**

Ex) int a, b;
 upFun(&a, &b);

- Syntax of pointer variables: **<type> * <identifier>**

Ex) int *pi; // pointer for integer variables
 float *pf; // pointer for float variables
 char *pc; // pointer for char variables

- Accessing value pointed by pointer variable:
indirection operator *

Ex) *ax = 23;
 *ay = 8;

Example: Exchange Function



■ Exchanging two variables

■ Incorrect example

```
int x = 10, y = 20;  
x = y;           // value of x is lost!  
y = x;           // value of x is 20
```

■ Correct example

```
int x = 10, y = 20, temp = 0;  
temp = x;        // save value of x  
x = y;  
y = temp;        // set y by old value of x
```

Example: Exchange Function

■ Calling function

```
int main()
{
    int a = 10, b = 20;
    ...
    Exchange(a, b);
    ...
}
```

x and y are exchanged,
but a and b are not

■ Called function

```
void Exchange(int x, int y)
{
    int temp = 0;
    temp = x;
    x = y;
    y = temp ;
}
```

■ Calling function

```
int main()
{
    int a = 10, b = 20;
    ...
    Exchange(&a, &b);
    ...
}
```

■ Called function

```
void Exchange(int *x, int *y)
{
    int temp = 0;
    temp = *x;
    *x = *y;
    *y = temp ;
}
```

Example: Quotient and Remainder

- Get two numbers and print their quotient and remainder

```
#include <stdio.h>
void divide (int dividend, int divisor, int* quotient, int* remainder);

int main()
{
    int num1 = 0, num2 = 0;
    int quo = 0, rem = 0;
    scanf("%d %d", &num1, &num2); // missed in your slides
    divide(num1, num2, &quo, &rem);
    printf("%d / %d = %d\n", num1, num2, quo);
    printf("%d %% %d = %d\n", num1, num2, rem);

    return;
}

void divide (int dividend, int divisor, int* quotient, int* remainder)
{
    *quotient = dividend / divisor;
    *remainder = dividend % divisor;
    return;
}
```

Typical Procedure to Use a Function



- **Define a function.**
 - Design function header
 - Defines the input and the output of the function
 - Implement function body
- **Declare the function before the first call.**
 - Just copy the function header and attach semicolon.
 - Usually, functions are declared at before the first function definition.
- **Use the function.**
 - Function call (function invocation)

Exercise



- Write a program that prints the least significant (rightmost) digit of any integer read from the keyboard.
 - Write a function “firstDigit” that extracts the rightmost digit.
- Ex) Input a number: **532**
Least significant digit of 532 is 2.

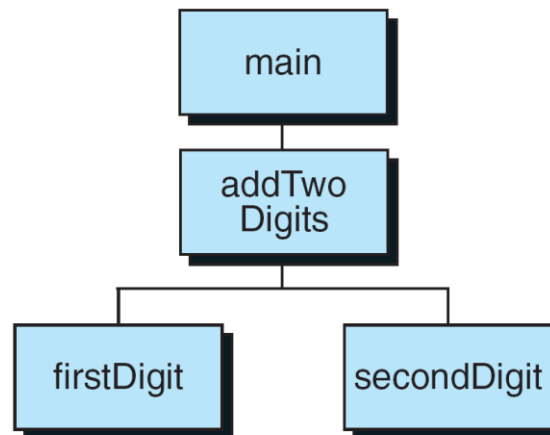
Exercise

- Extend the previous program to extract and add the two least significant digits of any integer number.

Ex) Input a number: 528

Sum of the two least significant digits is 10.

- Add a function to extract two digits and adds the two least significant digits.
- Add a function to extract the second least significant digits.



Exercise



- Write a program `sumprod.c` that reads two integers and prints their sum and product.
 - Implement and use a function “ReadTwoNumbers” to read the two numbers.
 - Implement and use a function “GetSumAndProduct” to compute the sum and the product.

Agenda



- Designing Structured Programs
- Functions in C
- Inter-function Communication
- Standard Functions
- Scope

Standard Functions



- **Standard functions:** built-in functions provided by C language itself
 - **Function declaration:** **system header files**
 - To use standard functions, proper header files should be included
Ex) `stdio.h` for `printf`, `scanf`
 - Locations of system header files are vary with system.
Ex) `C:\Program Files\Microsoft Visual Studio 10.0\VC\include`
`C:\Dev-Cpp\include`
`/usr/include`
Cf) `#include < >` vs. `#include " "`
 - **Function definition:** **system library**
 - Integrated by **linker**

Standard Function in Hello.c



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

```
int main()
```

```
{
```

```
    printf("Hello, World!\n");
```

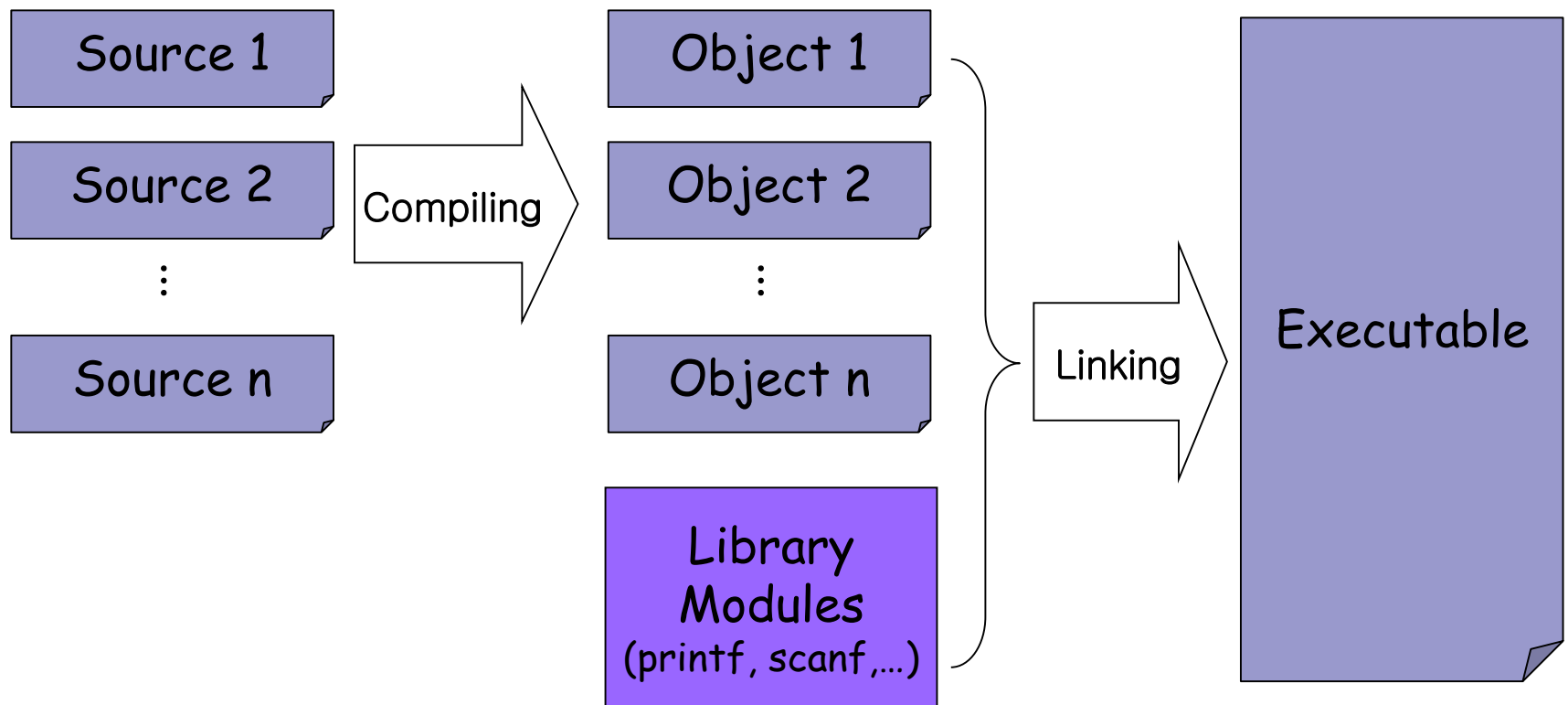
```
    return 0;
```

```
}
```

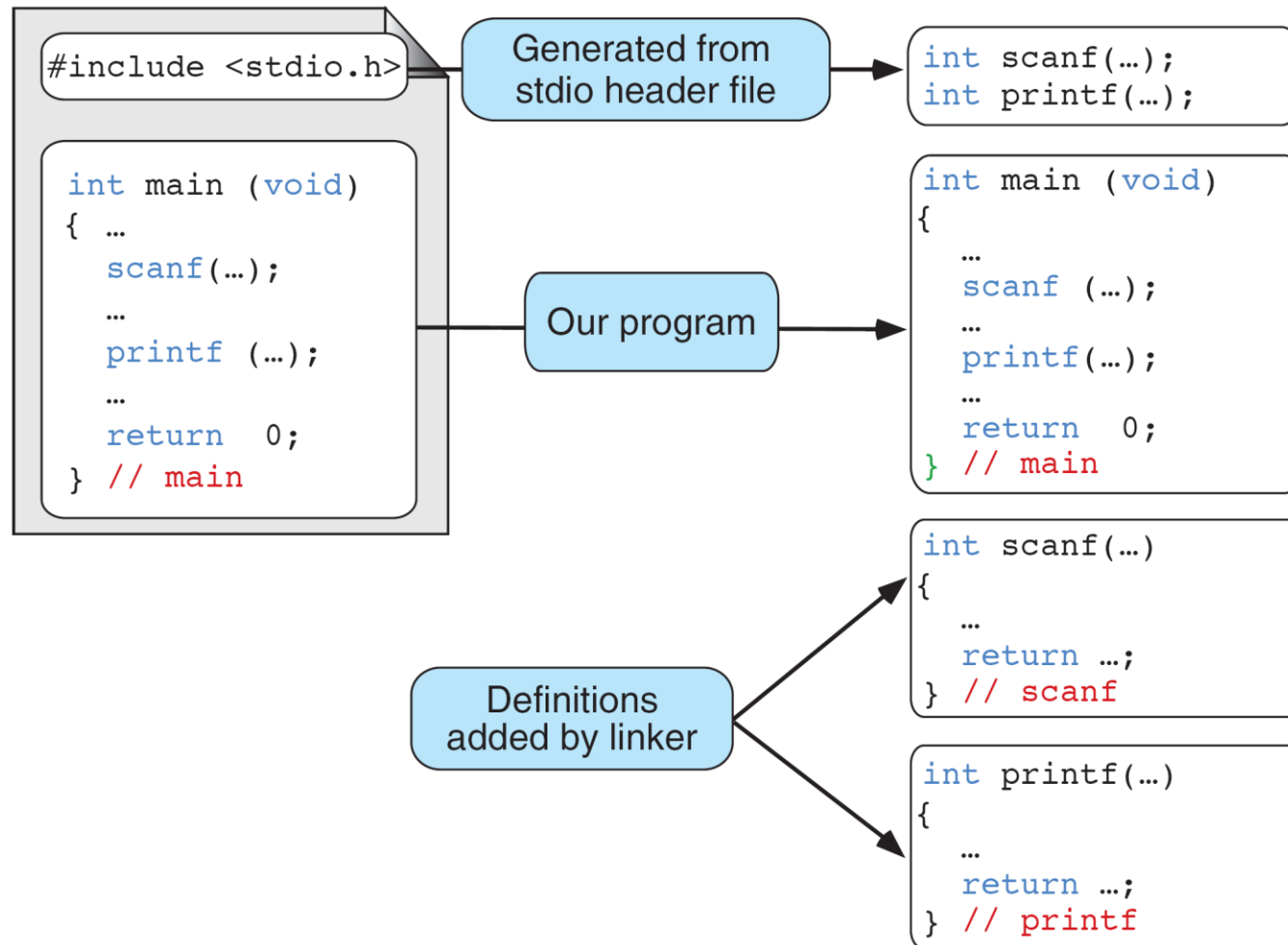
Linking

■ Linking

- Integrating objects and **library modules** required to execute



Standard Functions



Standard Functions



■ C provides a rich collection of standard functions

- Standard I/O (**stdio.h**)
 - printf, scanf, getchar, fprintf, fscanf, ...
- Math library (**math.h**)
 - abs, sin, cos, exp, log, pow, rand, ...
- Type library (**ctype.h**)
 - isalpha, isdigit, ...
- String manipulation (**string.h**)
 - strcpy, strcat, strcmp, ...
- ETC.

■ References

- C/C++ reference sites
 - <http://www.cppreference.com>
 - <http://msdn.microsoft.com>
- Manual page on UNIX (incl. cygwin)
 - Ex) \$ man -s3 printf // -s3 specifies section for library functions

Examples of Standard Functions



■ Absolute value (math.h)

- `int abs(int);` // `abs(-1)` // = 1;
- `double sin(double), double cos(double), double exp(double), ...`

■ System command (stdlib.h)

- `int system(char *command);`
Ex) `system("dir");` // displays files in current directory

■ Current time (time.h)

- `time_t time(time_t *);` // get current time
 - Return the time as seconds elapsed since midnight, Jan 1st, 1970.

■ Random number generation (stdlib.h)

- `void srand(unsigned int seed);` // initialize random seed
- `int rand(void);` // generate a random range 0 to RAND_MAX
 - RAND_MAX is defined in stdlib.h

Random Number

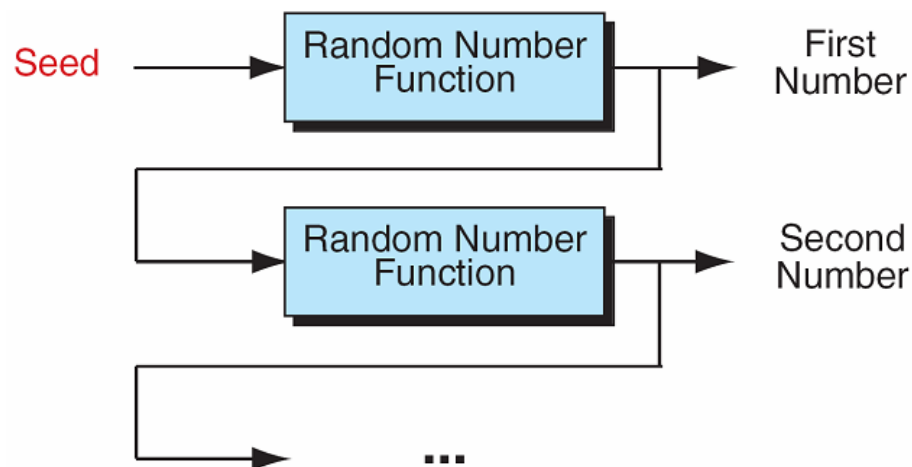
- C language cannot generate truly random number but **pseudo random number** from previous number (seed)

- Pseudo random numbers depend on previous number, but seems to be random

Ex) $\text{next_random} = (18394 * \text{seed} + 2567) \% 32768$

$\text{seed} = \text{next_random}$

Just an example!



Random Number



■ Specifying random seed

Ex) initializing seed with a constant → generate same sequence for all executions

```
srand(997);  
printf("rand() = %d\n", rand());  
printf("rand() = %d\n", rand());  
printf("rand() = %d\n", rand());
```

Ex) initializing seed according to current time → generate different sequence in every run

```
srand(time(NULL));  
printf("rand() = %d\n", rand());  
printf("rand() = %d\n", rand());  
printf("rand() = %d\n", rand());
```

Random Number



- If we want to a random number in a specific range, return value of rand() should be **scaled**.
 - `rand() % range + mininum;`
 - `range = (maximum - minimum) + 1` // [mininum, maximum]
- Ex) random number between 3 and 7
- ```
rand_num = rand() % 5 + 3;
```

# Exercise



- Write functions for the following mathematical expressions. Assume all variables are double type.

1.  $\sqrt{u + v} w^2$

2.  $\sqrt{(u - v)^3}$

3.  $\log_e(x^y)$

4.  $|xy - w / z|$

# Agenda

---



- Designing Structured Programs
- Functions in C
- Inter-function Communication
- Standard Functions
- Scope

# Scope

- **Scope**: region of program in which a defined object is visible
  - **Global scope**: object visible from its declaration to the end of program
  - **Local scope**: object that exists only from its declaration to the end of function or **block** (compound statement)

```
#include <stdio.h>
int sum = 0; // global declaration
int main(void)
{
 int a = 0, b = 0; // local declaration
 // some codes
 return 0;
}
```

# Scope

```
/* This is a sample to demonstrate scope. The techniques
 used in this program should never be used in practice.
*/
```

```
#include <stdio.h>
int fun (int a, int b);
```

Global area

```
int main (void)
```

```
{
```

```
 int a;
```

```
 int b;
```

```
 float y;
```

```
 ...
```

```
 { // Beginning of nested block
```

```
 float a = y / 2;
```

```
 float y;
```

```
 float z;
```

```
 ...
```

```
 z = a * b;
```

```
 ...
```

```
 } // End of nested block
```

```
 ...
```

```
} // End of main
```

main's area

Nested block  
area

```
int fun (int i, int j)
```

```
{
```

```
 int a;
```

```
 int y;
```

```
 ...
```

```
} // fun
```

fun's area

# Scope

- What's the result of the following program?

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

```
int main()
```

```
{
```

```
 int x = 100;
```

```
 {
```

```
 float y = x;
```

```
 float x = 50;
```

```
 printf("y = %f\n", y);
 }
```

```
 return 0;
```

```
}
```



# Scope

---



- Smaller scope gets higher priority

Ex) `float a` and `float y` overrides `int a` and `int y` in previous code

- Recommendation

- It is poor programming style to reuse identifiers within the same scope.