CS1010E TOPIC 2:
C BASIC & MODULAR DESIGN

Siau-Cheng KHOO

Block COM2, Room 04-11, +65 6516 6730

www.comp.nus.edu.sg/~khoosc

khoosc@nus.edu.sg

Semester II, 2017/2018

Lecture Outline

- **Better Understanding of Structure of C programs**
  - Constants and Variables
  - Input/Output statements: `printf` and `scanf`
- **Modular Design with User-defined Functions**
  - Function Definitions and Function Prototypes
  - Function Application/Call
  - Scoping Rules and Pass-by-Value Parameter Passing
  - Execution model for function calls

Let’s review C program basic

```c
#include <stdio.h>

int main(void) {
        int a, b, rem;           // declaring variables

        printf("Enter two non-negative integers: ");
        scanf("%d %d", &a, &b) ;

        while (b>0) {
                rem = a % b;   // "a % b" is "a modulo b"
                a = b;
                b = rem;
        }

        printf(“The result of gcd is %d.\n", a) ;
        return 0;
}
```

Preprocessor Directives

- Provide instructions that are performed before the program is compiled
- Begins with a #
- #include inserts additional statements in the program

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

- `<stdio.h>` -- info related to input/output statements used in the program
- .h – “file extension” specifies that they are header files.
- < … > -- the file within comes from the Standard C Library that comes with ANSI C compiler

```
#include <stdio.h>

int main(void) {
        int a, b, rem;              // decla

        printf("Enter two non-negative i
        scanf("%d %d", &a, &b) ;

        while (b>0) {
                rem = a % b;    // "a % b"
                a = b;
                b = rem;
        }

        printf(“The result of gcd is %d.
        return 0;
}
```

Main Function

- Every C program contains a set of statements forming a main function
- Only one main function available
- Keyword int – function returns an integer value to the operating systems (OS)
- Keyword void – the function is not receiving any info from the OS
- Symbols { } – function body is enclosed by curly braces, { and }
- Function body contains two types of commands: declarations and statements
    - They are all indented for clarity

```
#include <stdio.h>

int main(void) {
    int a, b, rem;          // d

    printf("Enter two non-negativ
    scanf("%d %d", &a, &b) ;

    while (b>0) {
        rem = a % b;    // "a %
        a = b;
        b = rem;
    }

    printf(“The result of gcd is
    return 0;
}
```

Declarations

- Defines the memory locations that will be used by the statement in the function body
- Must appear before statements
- Each declaration ends with a “;”
- Each needed memory location is given a name – variable
  - Variables are separated by “,”
- Each variable is declared with a memory size and the kind of values it will store inside that memory – data type

```
int a, b, rem ;
double size_1, size_2, size_3 = 10.5,
           size_4 ;
```

```
#include <stdio.h>

int main(void) {
        int a, b, rem;            // d

        printf("Enter two non-negativ
        scanf("%d %d", &a, &b) ;

        while (b>0) {
                rem = a % b;    // "a %
                a = b;
                b = rem;
        }

        printf(“The result of gcd is
        return 0;
}
```

Statements

- Specify the operations to be performed
- printf prints information to the monitor

```
printf(“hello \n”) ;

printf(“hello”

        “world”

        “\n”) ;

printf(“result is %d \n”, rem) ;

printf(“result is %5.2f \n”, x);
```

- scanf reads input from keyboard and stores into memory referred to by variables

```
scanf(“%d %d”, &a, &b) ;

scanf(“%lf %lf”, &x, &y) ;
```

- They both ends with “;”

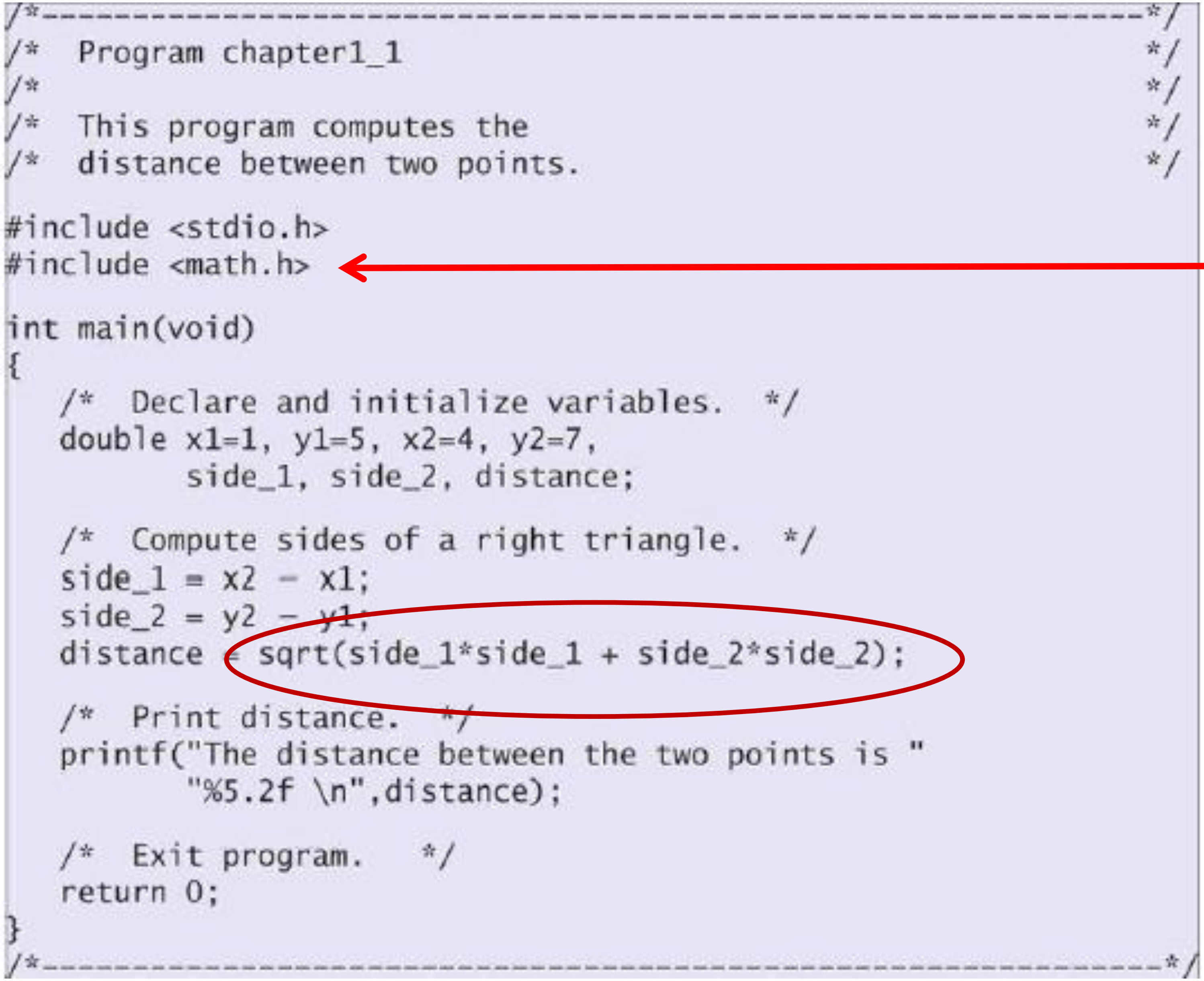
```
#include <stdio.h>

int main(void) {
        int a, b, rem;              // declarin

        printf("Enter two non-negative integ
        scanf("%d %d", &a, &b) ;

        while (b>0) {
                rem = a % b;    // "a % b" is
                a = b;
                b = rem;
        }

        printf(“The result of gcd is %d.\n",
        return 0;
}
```

White Space

- We also include blank spaces, tabs, blank lines, etc. to make the program more readable.
- Declarations and statements are indented to show the structure of the program.

```
#include <stdio.h>

int main(void) {
        int a, b, rem;           // declari

        printf("Enter two non-negative inte
        scanf("%d %d", &a, &b) ;

        while (b>0) {
                rem = a % b;   // "a % b" is
                a = b;
                b = rem;
        }

        printf("The result of gcd is %d.\n'
        return 0;
}
```

Another sample program



```
/*-----------------------------------------------------------*/
/*  Program chapter1_1                                        */
/*                                                            */
/*  This program computes the                                 */
/*  distance between two points.                              */

#include <stdio.h>
#include <math.h>

int main(void)
{
   /*  Declare and initialize variables.  */
   double x1=1, y1=5, x2=4, y2=7,
          side_1, side_2, distance;

   /*  Compute sides of a right triangle.  */
   side_1 = x2 - x1;
   side_2 = y2 - y1;
   distance = sqrt(side_1*side_1 + side_2*side_2);

   /*  Print distance.  */
   printf("The distance between the two points is "
          "%5.2f \n",distance);

   /*  Exit program.  */
   return 0;
}
/*-----------------------------------------------------------*/
```

`<math.h>` contains mathematics functions such as `sqrt` that can be used in this program.

Constants

- **Specific values that we use in the program, such as**

1 5 4 7	**integers**
3.14159 -1.5	**reals/floating-point numbers**
‘a’ ‘x’	**character**
“This is a test”	**constant string**

- **They are constants because you (your program) can’t change them**

Variables

- **Memory locations that are assigned a name or identifier**
- **Rules for selecting a valid identifier are:**
  - **It must begin with an alphabetic character or the underscore character (_)**
  - **An alphabetic character in an identifier can be lowercase or uppercase**
  - **An identifier can contain digits, but not as the first character; and**
  - **An identifier can be of any length**

Case Sensitive

- **C is case sensitive, thus uppercase letters are different from lowercase letters**

- `Total`, `TOTAL` **and** `total` **represent three different variables**

- **C also includes keywords with special meaning to the C compiler that cannot be used for identifiers**

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Identifier Names

- Identifier names should be carefully selected
- It must reflect the contents of the variable
- It should also indicate the unit of measurement
- Eg: a variable represents a temperature measurement in Celsius, use an identifier such as `temp_C` or `degree_C`.

Variables: Initialization

```
int count;

count = count + 12;
```

What value does ‘count’ contain hold after this statement?

```
int count;

count = 0;

count = count + 12;
```

Assignment

```
int count = 0;

count = count + 12;
```

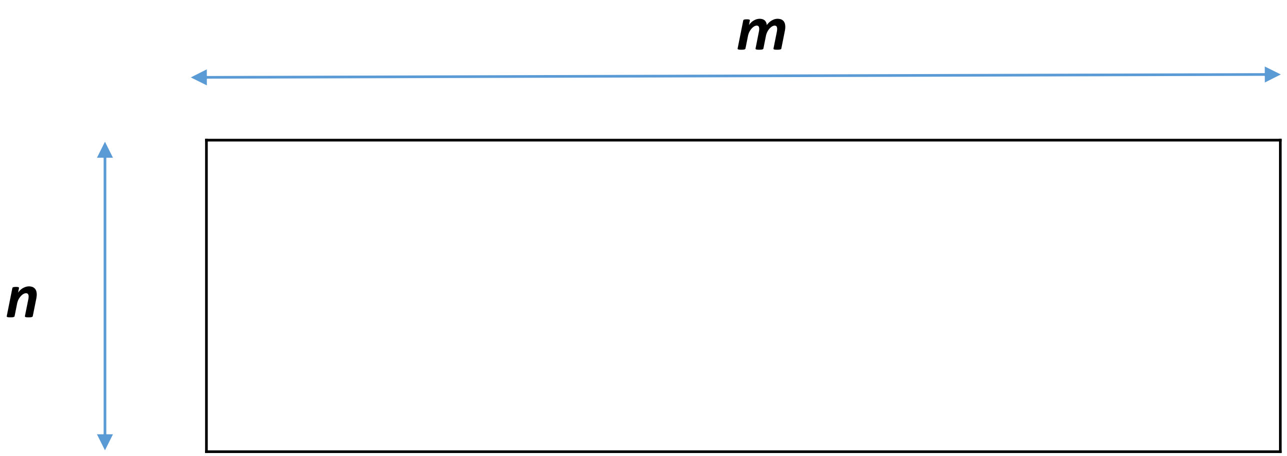
Initialization

Modular Design

- How to analyse, design and implement a program ?
  - Problem Solving Process (Computational Thinking)
- How to break a problem into sub-problems with step-wise refinement ?
  - Modular design; step-wise refinement
- How to create your own (user-defined) functions to address sub-problems?
  - Function definition
  - Function prototype

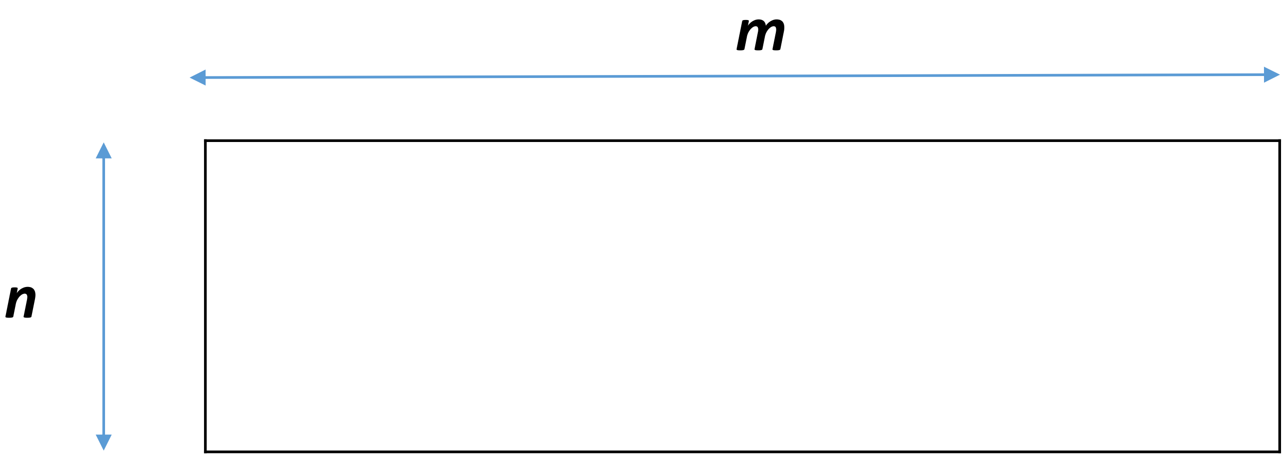
Problem #3

- Adam would like to fully cover a rectangular-sized floor of length *m* metres and breadth *n* metres by square tiles of arbitrary size of integer length, what is the minimum number of square tiles needed to fully cover the floor? (*m* and *n* are integers)



Problem #3 (Analysis & Design)

- **Adam would like to fully cover a rectangular-sized floor of length *m* metres and breadth *n* metres by square tiles of arbitrary size of integer length, what is the minimum number of square tiles needed to fully cover the floor? (*m* and *n* are integer)**
-



Think: ***Can there really be a solution to this problem?***

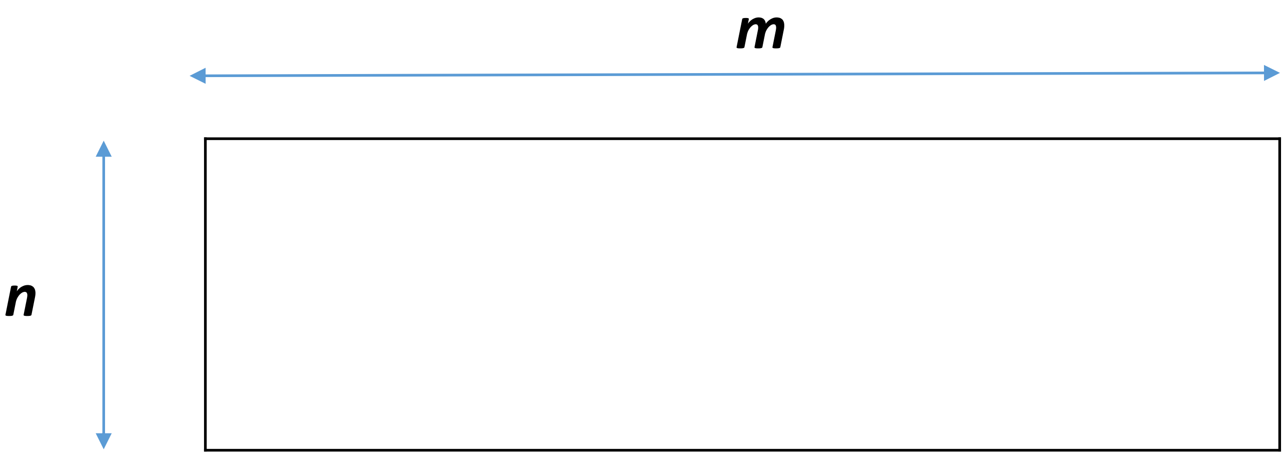
Can the floor really be covered with square tiles?

YES!!!

Problem #3 (Analysis & Design)

- **Adam would like to fully cover a rectangular-sized floor of length *m* metres and breadth *n* metres by square tiles of arbitrary size of integer length, what is the minimum number of square tiles needed to fully cover the floor?**

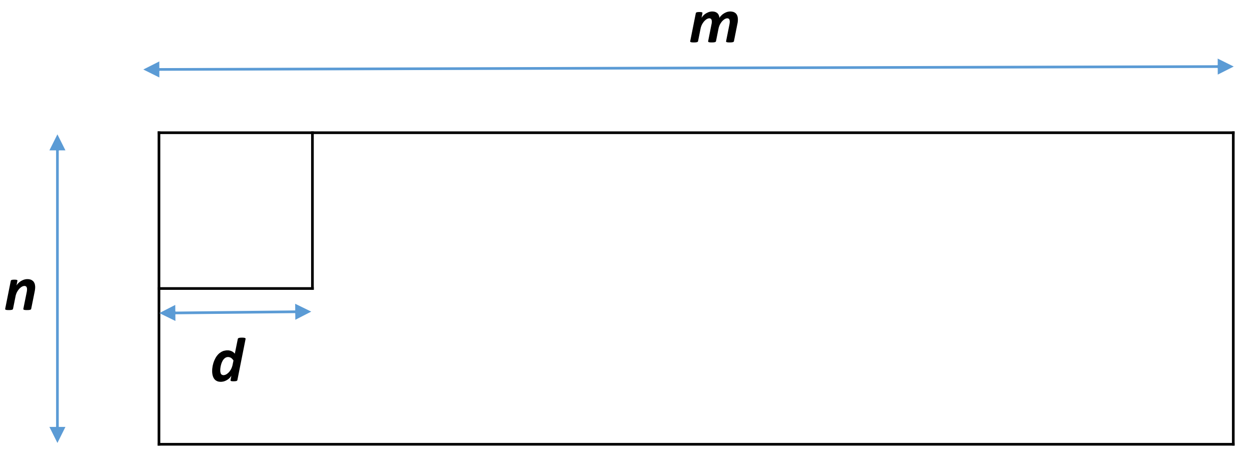
Find the length of largest square tile, result is *d*



Problem #3 (Analysis & Design)

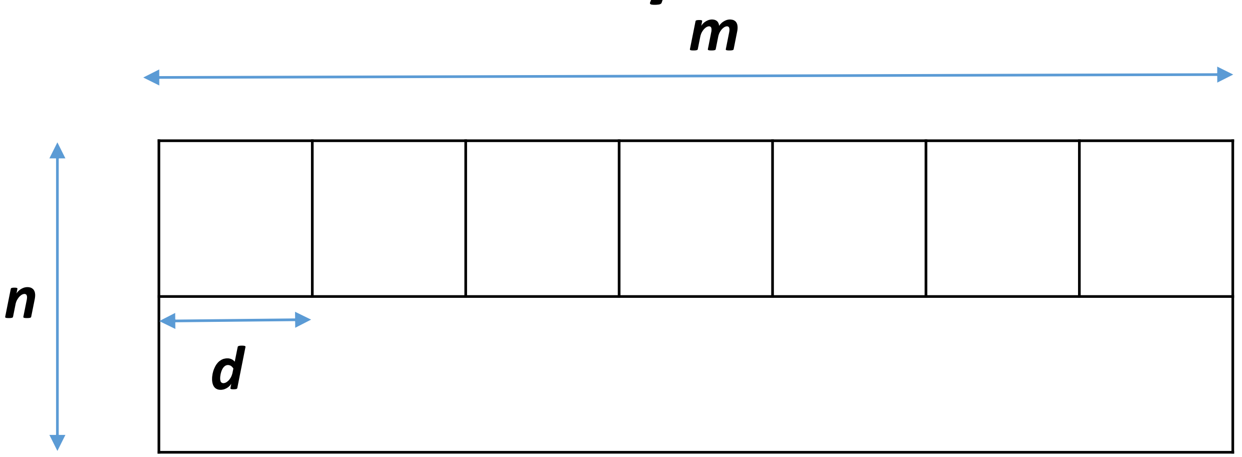
- **Adam would like to fully cover a rectangular-sized floor of length *m* metres and breadth *n* metres by square tiles of arbitrary size of integer length, what is the minimum number of square tiles needed to fully cover the floor?**

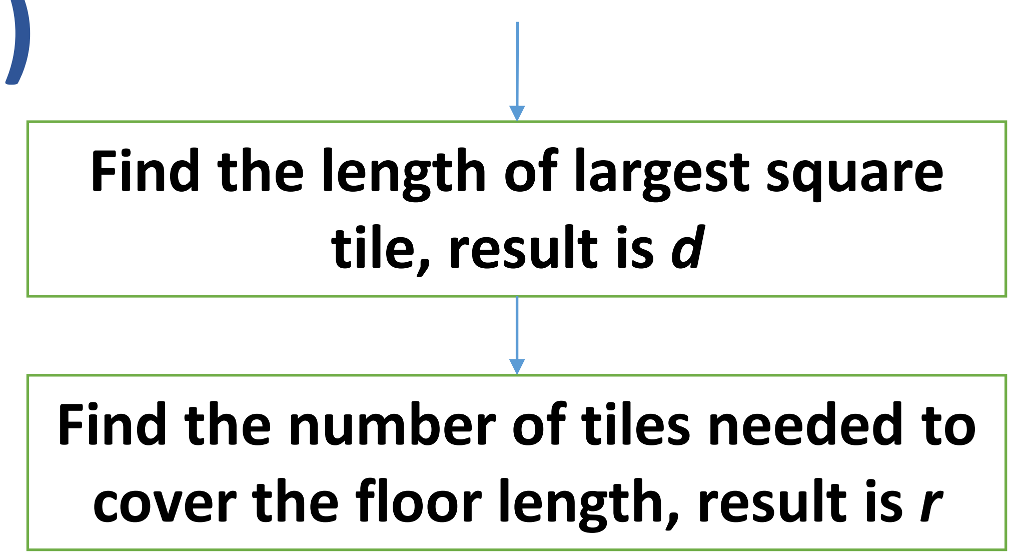
Find the length of largest square tile, result is *d*



Problem #3 (Analysis & Design)

- Adam would like to fully cover a rectangular-sized floor of length *m* metres and breadth *n* metres by square tiles of arbitrary size of integer length, what is the minimum number of square tiles needed to fully cover the floor?



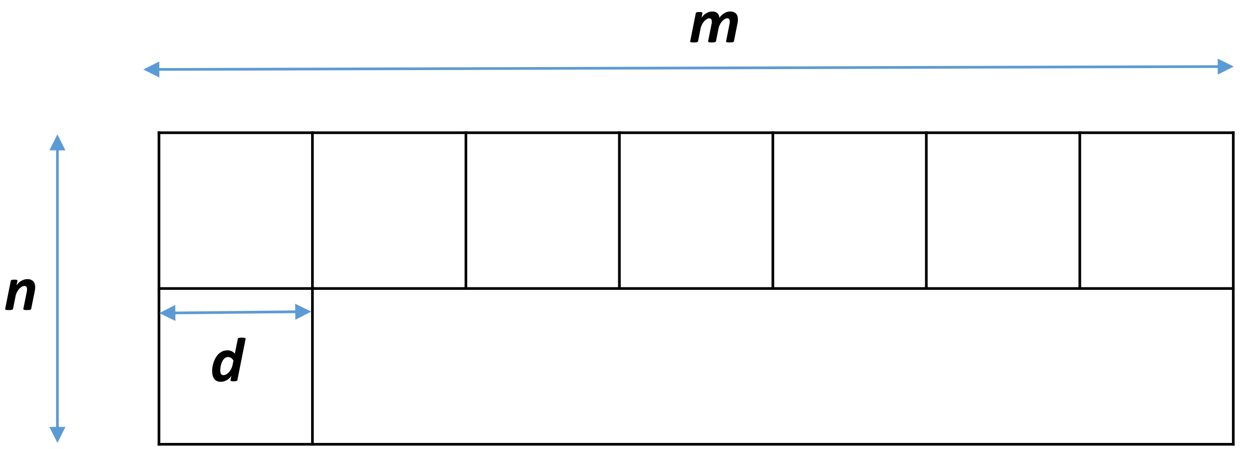


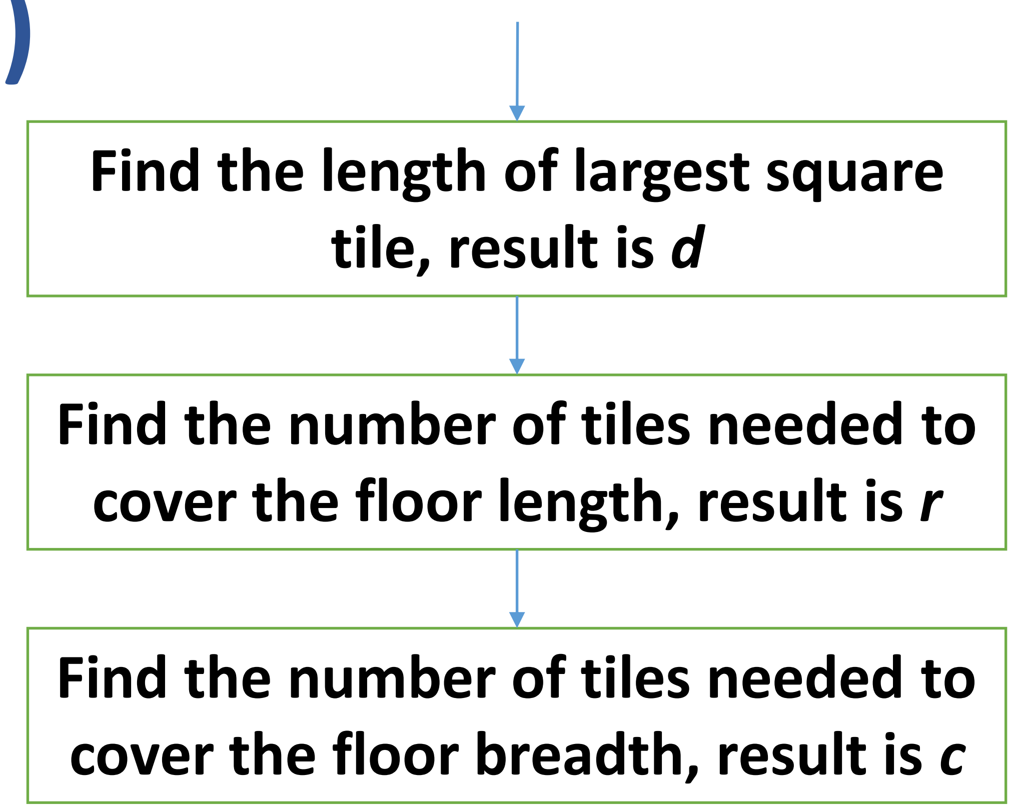
Find the length of largest square tile, result is *d*

Find the number of tiles needed to cover the floor length, result is *r*

Problem #3 (Analysis & Design)

- Adam would like to fully cover a rectangular-sized floor of length *m* metres and breadth *n* metres by square tiles of arbitrary size of integer length, what is the minimum number of square tiles needed to fully cover the floor?





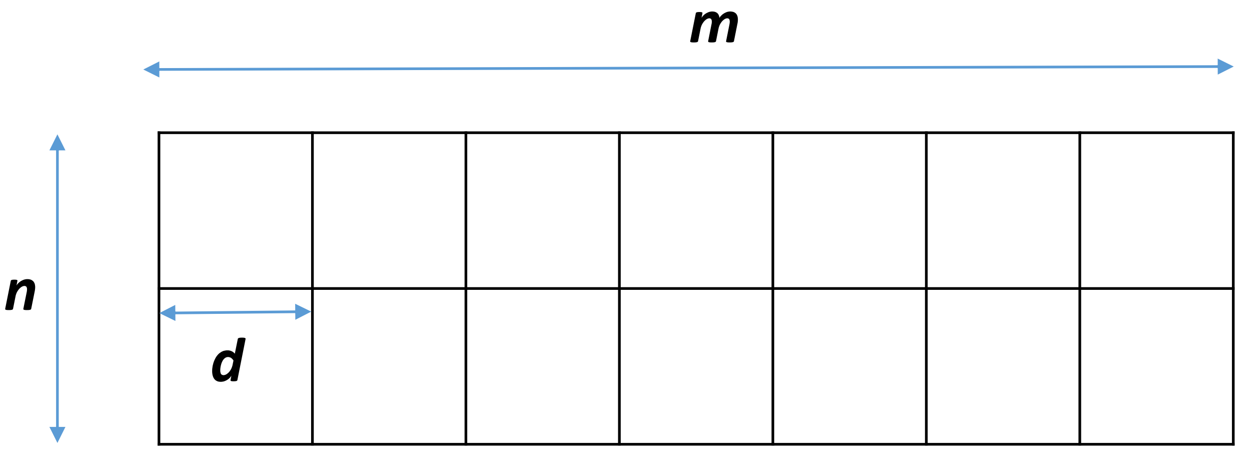
Find the length of largest square tile, result is *d*

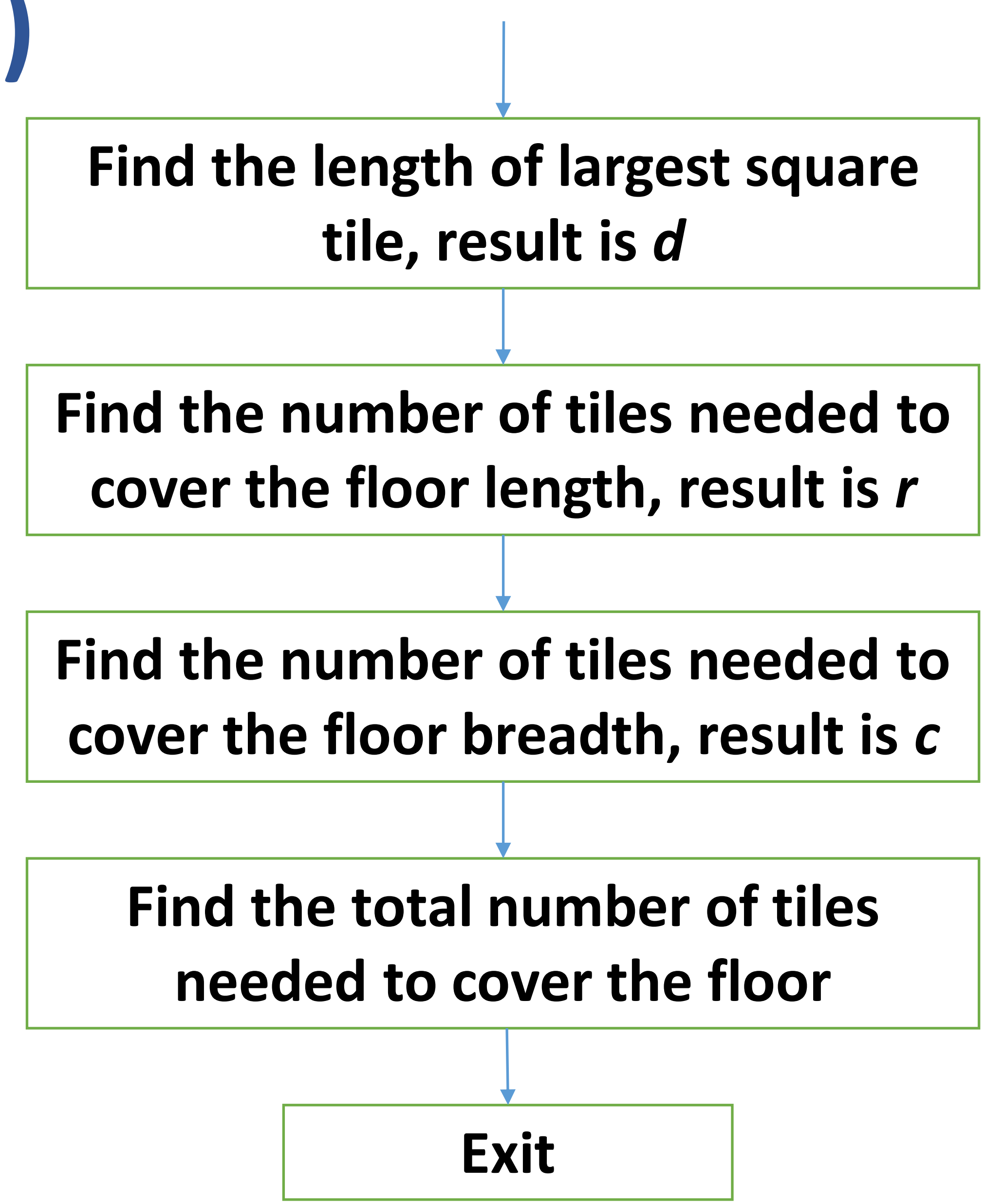
Find the number of tiles needed to cover the floor length, result is *r*

Find the number of tiles needed to cover the floor breadth, result is *c*

Problem #3 (Analysis & Design)

- Adam would like to fully cover a rectangular-sized floor of length *m* metres and breadth *n* metres by square tiles of arbitrary size of integer length, what is the minimum number of square tiles needed to fully cover the floor?





Find the length of largest square tile, result is *d*

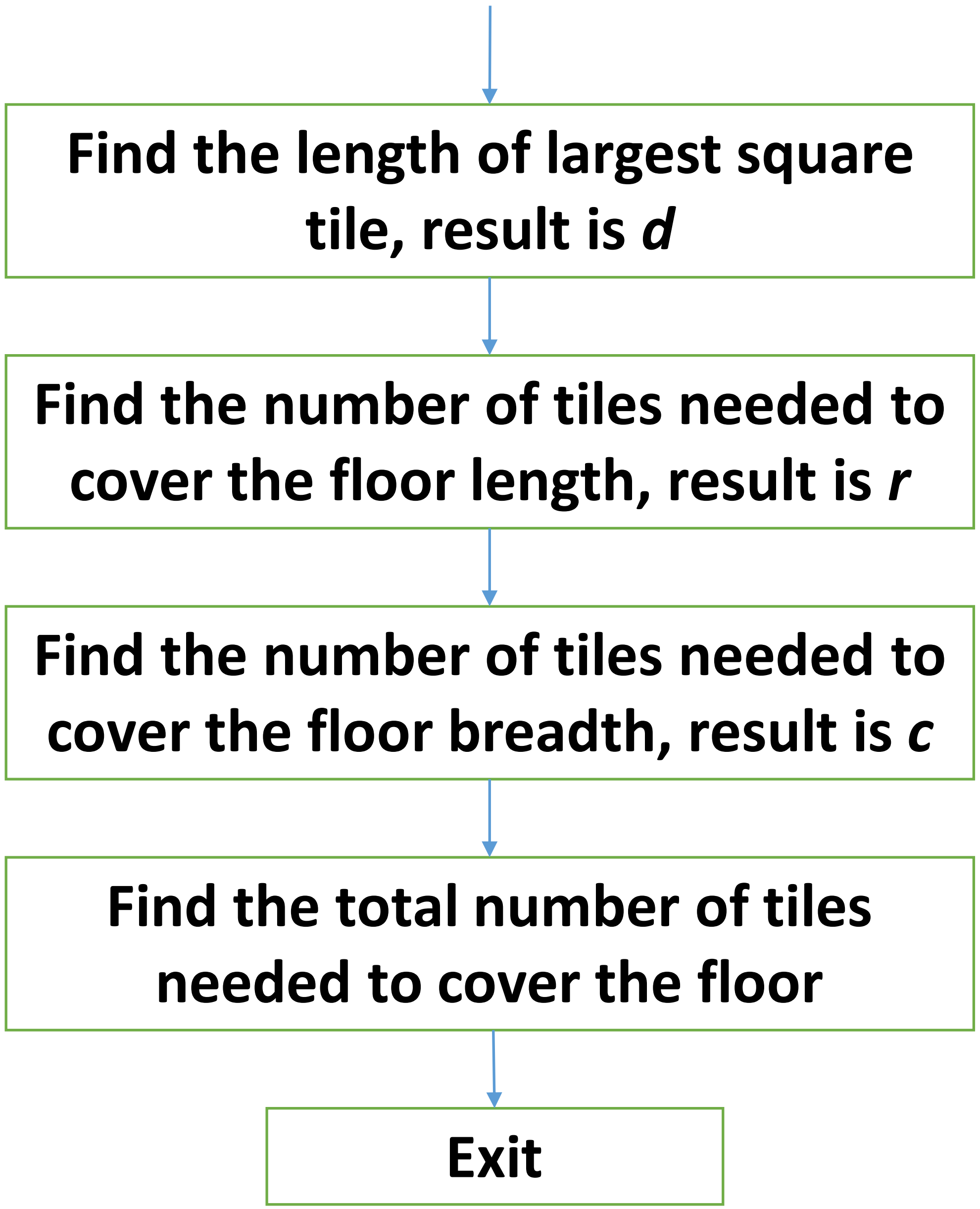
Find the number of tiles needed to cover the floor length, result is *r*

Find the number of tiles needed to cover the floor breadth, result is *c*

Find the total number of tiles needed to cover the floor

Exit

Problem #3 (Step-wise refinement)



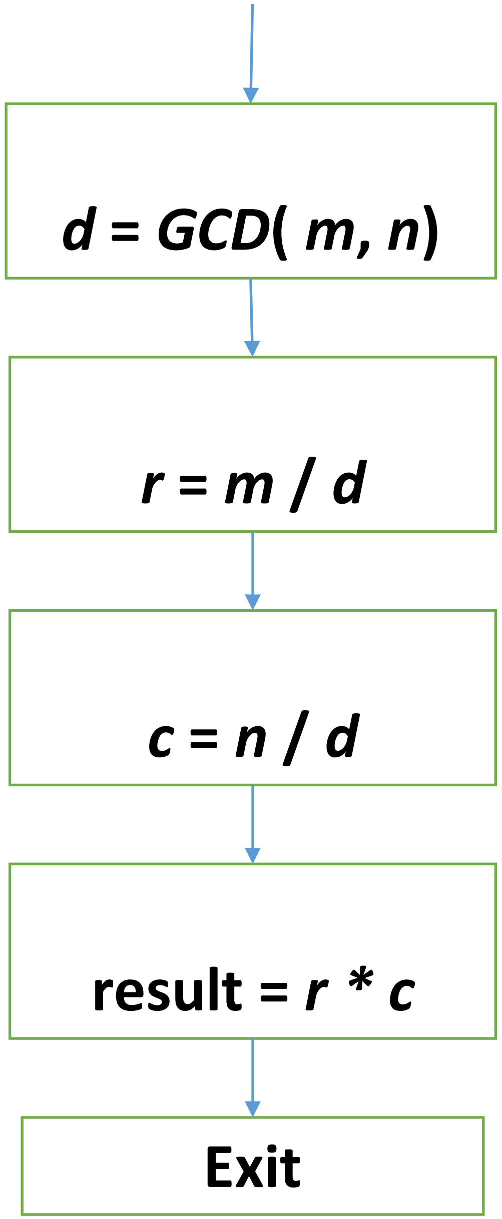
Find the length of largest square tile, result is *d*

Find the number of tiles needed to cover the floor length, result is *r*

Find the number of tiles needed to cover the floor breadth, result is *c*

Find the total number of tiles needed to cover the floor

Exit



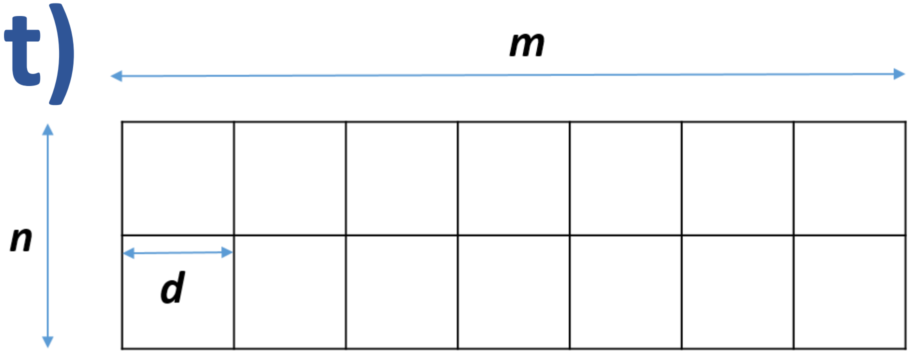
$d = GCD(m, n)$

$r = m / d$

$c = n / d$

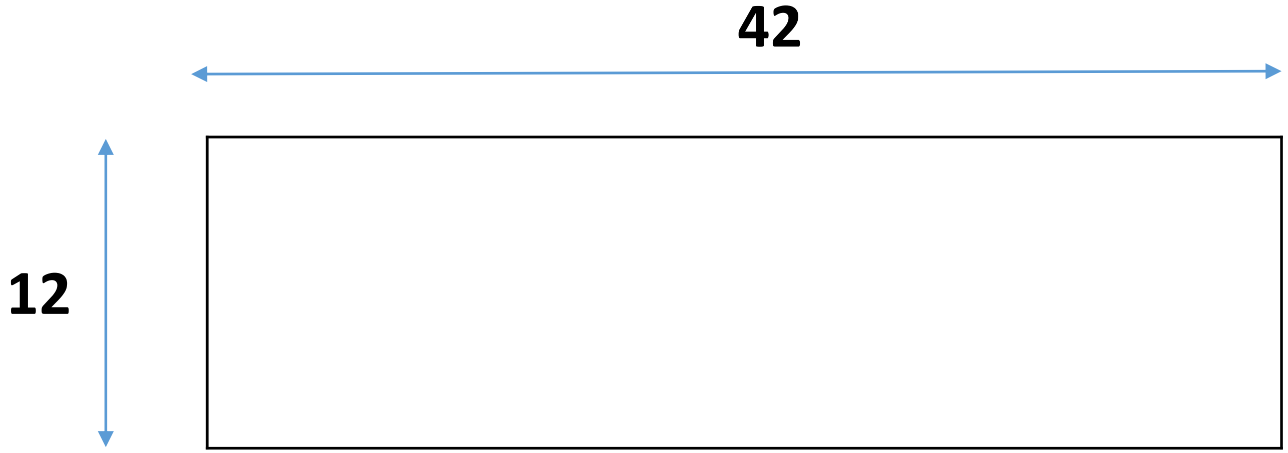
result = $r * c$

Exit



Problem #3 (testing with an instance)

- **Adam would like to fully cover a rectangular-sized floor of length 42 metres and breadth 12 metres by square tiles of arbitrary size of integer length, what is the minimum number of square tiles needed to fully cover the floor?**



Problem #3 (testing with an instance)

1. Find the GCD of 42 and 12, resulting is 6

2. Find the number of tiles need to cover the length,

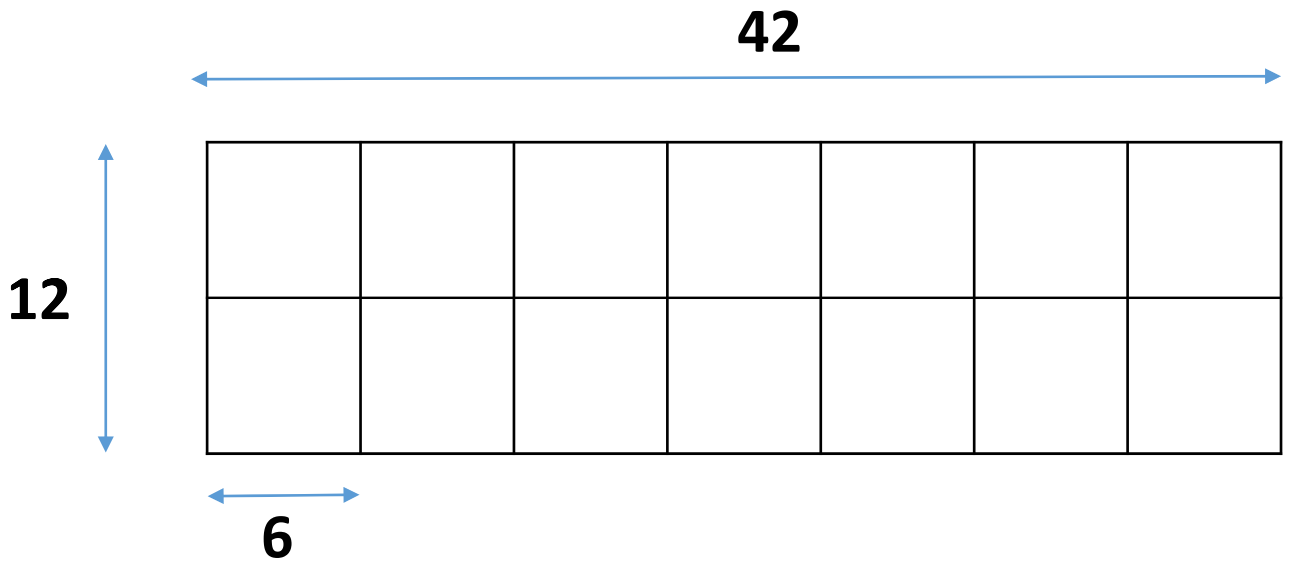
   $r = 42 / 6 = 7$

3. Find the number of tiles needed to cover the breadth,

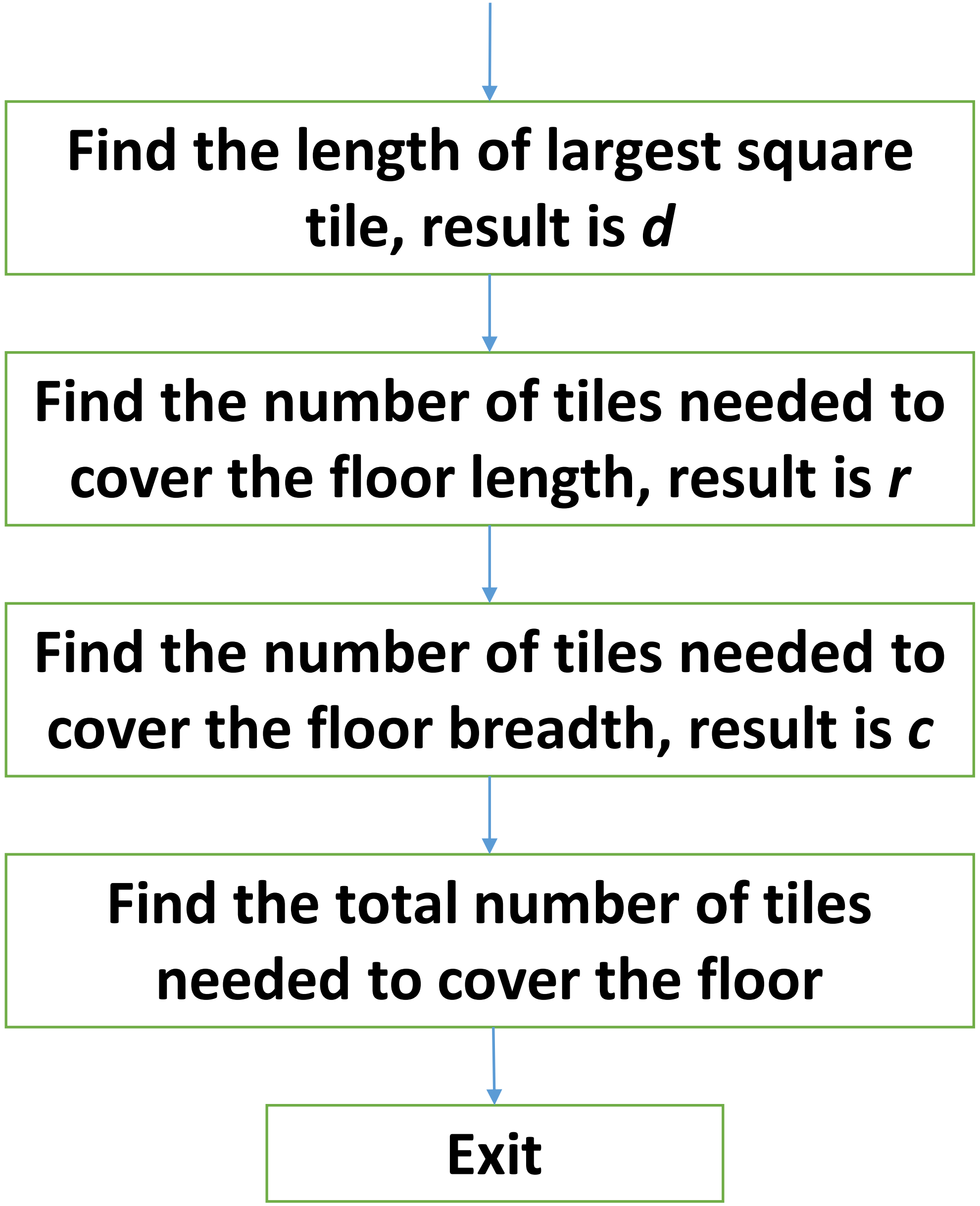
   $c = 12 / 6 = 2$

4. The number of tiles needed is

   $r * c = 14$.



Problem #3 (Step-wise refinement)



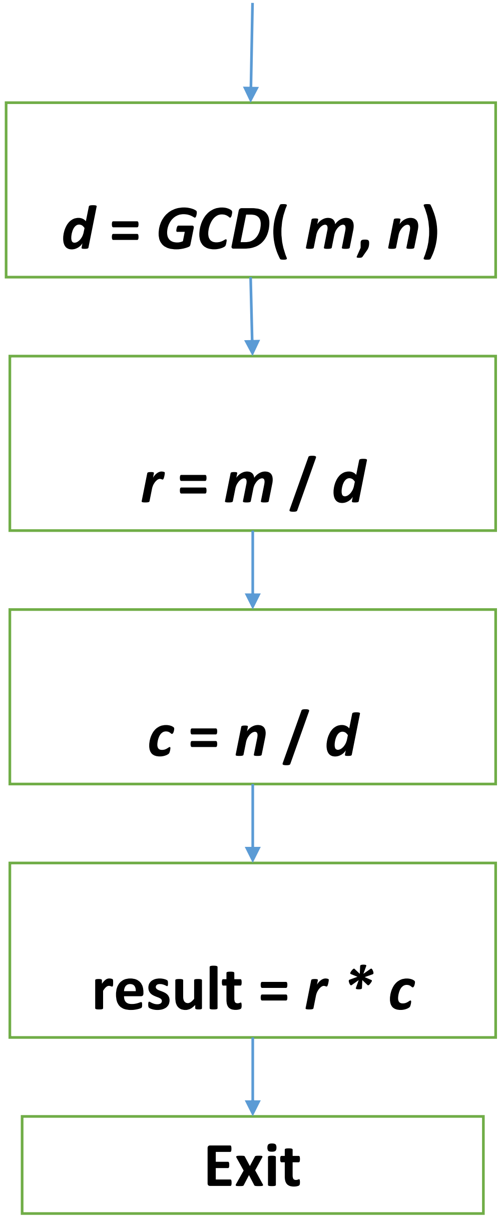
Find the length of largest square tile, result is *d*

Find the number of tiles needed to cover the floor length, result is *r*

Find the number of tiles needed to cover the floor breadth, result is *c*

Find the total number of tiles needed to cover the floor

Exit



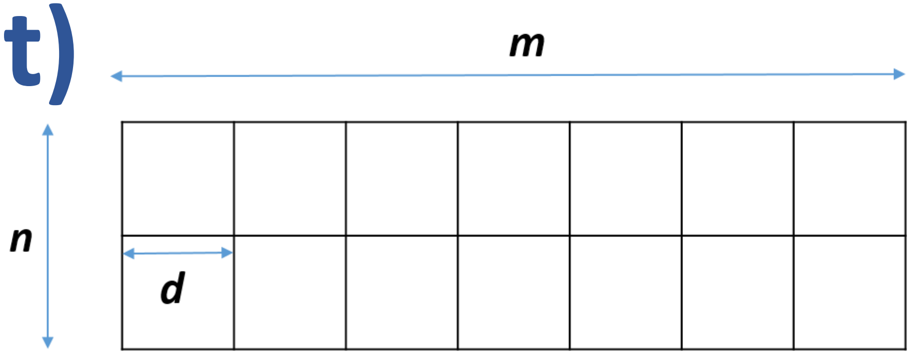
$d = GCD(m, n)$

$r = m / d$

$c = n / d$

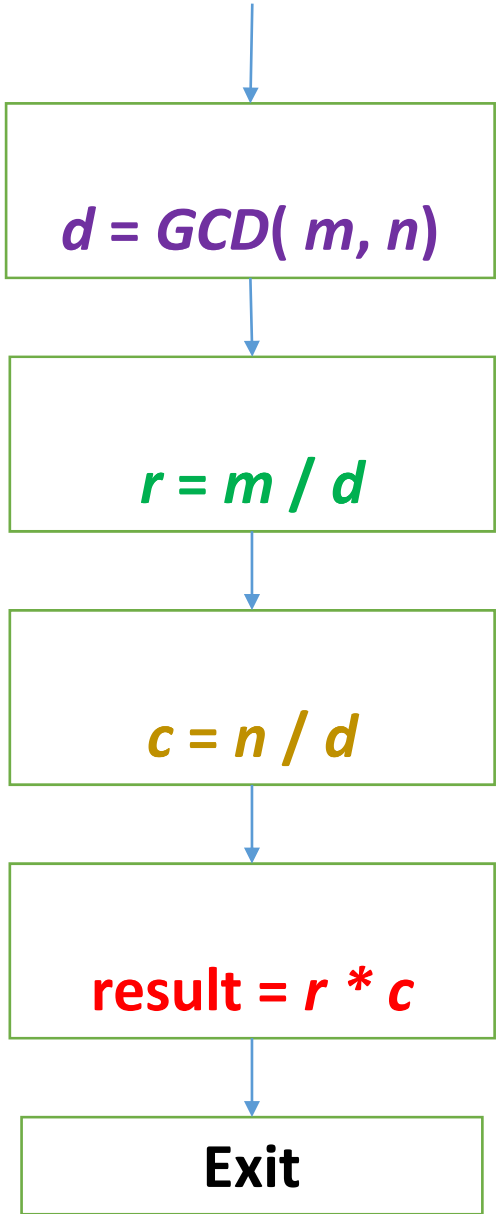
result = $r * c$

Exit



Take note of **how this solution has been designed in a modular fashion**

Problem #3 (Direct Implementation)



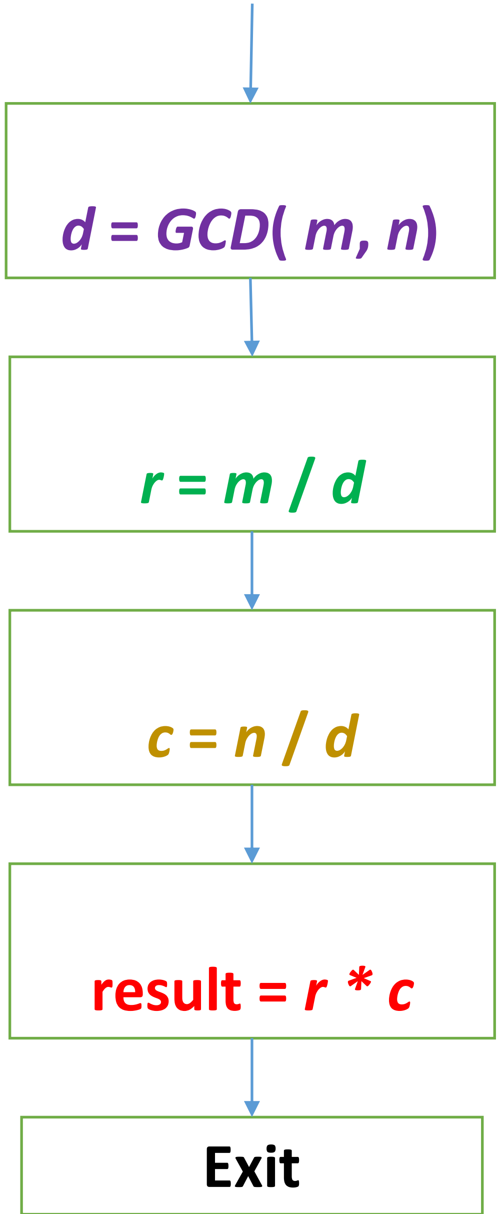
```
#include <stdio.h>

int main(void) {
       int m, n, d, r, c, result;            // declaring variables

       printf("Enter the floor length and breadth: ");
       scanf("%d %d", &m, &n) ;

       while (n>0) {       // Compute GCD(m,n)
              d = m % n;
              m = n;
              n = d;
       }
       r = m / d ;         // find # of tiles across length
       c = n / d ;         // find # of times across breadth
       result = r * c ;    // compute # of tiles in total
       printf(“You need %d tiles.\n", result) ;
       return 0;
}
```

This solution is incorrect. Why?

Problems with Direct implementation

- **The implementation code is cluttered, harder to understand, harder to maintain**
  - **Is it possible to subject the code to minimal change when the problem varies a little?**
- **GCD computation is commonly used, why must we always write GCD implementation whenever we want to use it?**
  - **Can we write it once and use it many times?**

Problem #3 (Modular Implementation)



```
#include <stdio.h>

int main(void) {
        int m, n, d, r, c, result;              // declaring variables

        printf("Enter the floor l
        scanf("%d %d", &m, &n) ;

        d = gcd(m,n)            // compute GCD of m and n

        r = m / d ;             // find # of tiles across length
        c = n / d ;             // find # of times across breadth
        result = r * c ;        // compute # of tiles in total
        printf("You need %d tiles.\n", result) ;
        return 0;
}
```

Calling a GCD function

Modular Design supports variants of problems

```c
#include <stdio.h>
// Covering the floor of sizes m and n
int main(void) {
        int m, n, d, r, c, result;           // declaring variables

        printf("Enter the floor length and breadth: ");
        scanf("%d %d", &m, &n) ;

        d = gcd(m,n)        // compute GCD of m and n

        r = m / d ;         // find # of tiles across length
        c = n / d ;         // find # of times across breadth
        result = r * c ;    // compute # of tiles in total
        printf(“You need %d tiles.\n", result) ;
        return 0;
}
```

Modular Design supports variants of problems

```
#include <stdio.h>
// Partially Cover floor of m and n
// Leave a pathway of 1 metre around the covered area
int main(void) {
      int m, n, d, r, c, result;            // declaring variables

      printf("Enter the floor length and breadth: ");
      scanf("%d %d", &m, &n) ;

      d = gcd(m-2,n-2)
      r = (m-2) / d ;                      // find # of tiles across length
      c = (n-2) / d ;                      // find # of times across breadth
      result = r * c ;    // compute # of tiles in total
      printf("You need %d tiles.\n", result) ;
      return 0;
}
d
```

```
#include <std
// Covering
int main(void)
      int m,

      printf(
      scanf("

      d = gcd

      r = m /
      c = n /
      result
      printf(
      return
}
```

Modular Design su

```
#include <std
// Covering
int main(void
      int m,

      printf(
      scanf("

      d = gcd

      r = m /
      c = n /
      result
      printf(
      return
}
```

```
#include <stdi
// Partially
// Leave a p
int main(void)
      int m, n

      printf("
      scanf("%

      d = gcd(
      r = (m-2
      c = (n-2
      result =
      printf("
      return 0
}
d
```

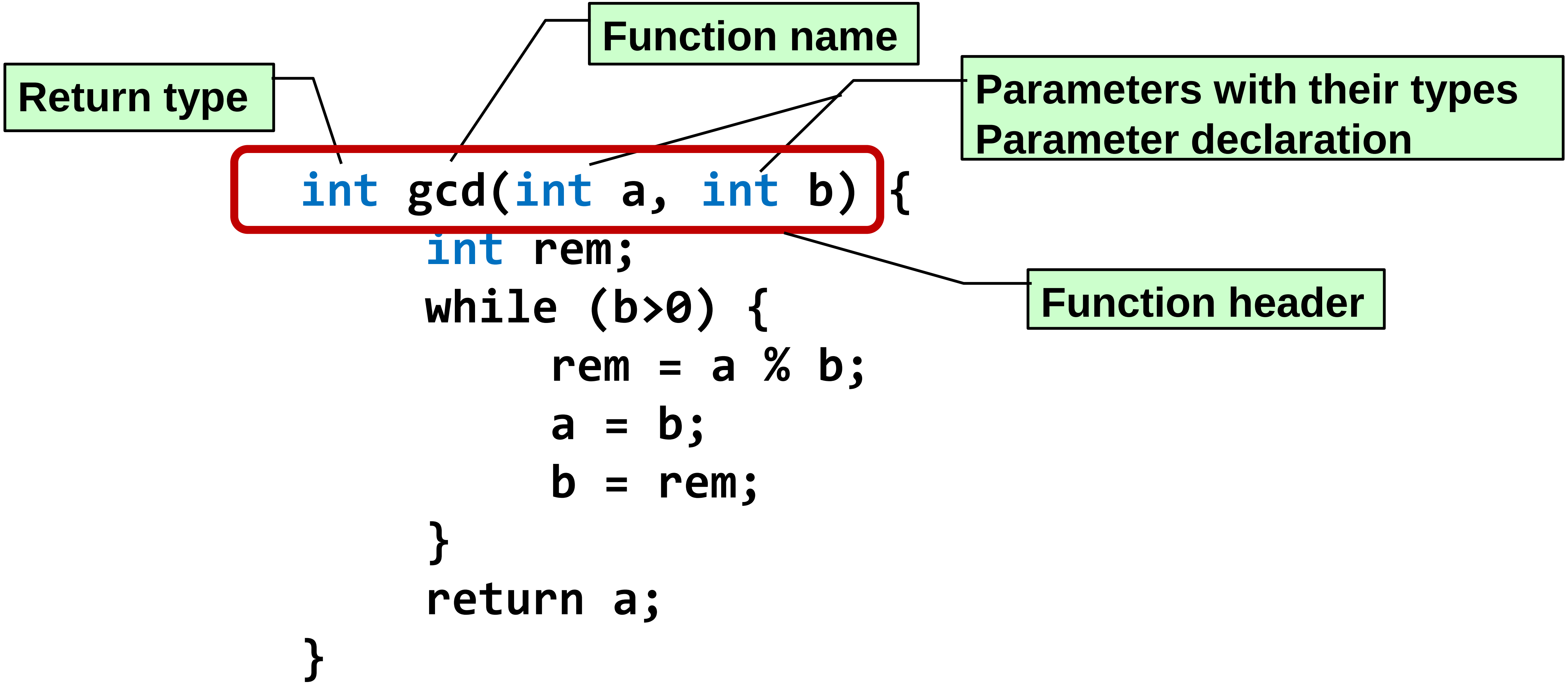
```
#include <stdio.h>
// Partially Cover floor of m and n
// at most half of the floor be covered
int main(void) {
      int m, n, d, r, c, result;           // declaring

      printf("Enter the floor length and breadth: ");
      scanf("%d %d", &m, &n) ;

      d = gcd(m / 2,n)
      r = (m / 2) / d ;    // find # of tiles across leng
      c = n / d ;          // find # of times across brea
      result = r * c ;     // compute # of tiles in total
      printf("You need %d tiles.\n", result) ;
      return 0;
}
d
```

Modular Design – User-defined Functions

```
#include <stdio.h>

int main(void) {
        int m, n, d, r,

        printf("Enter t
        scanf("%d %d",

        d = gcd(m,n)

        r = m / d ;
        c = n / d ;
        result = r * c
        printf("You nee
        return 0;
}
```

```
// Euclid's Algorithm for GCD Computation
int gcd(int a, int b) {
     int rem;
     while (b>0) {
          rem = a % b;
          a = b;
          b = rem;
     }
     return a;
}
```

Components of a User-defined Function Definition



Function name

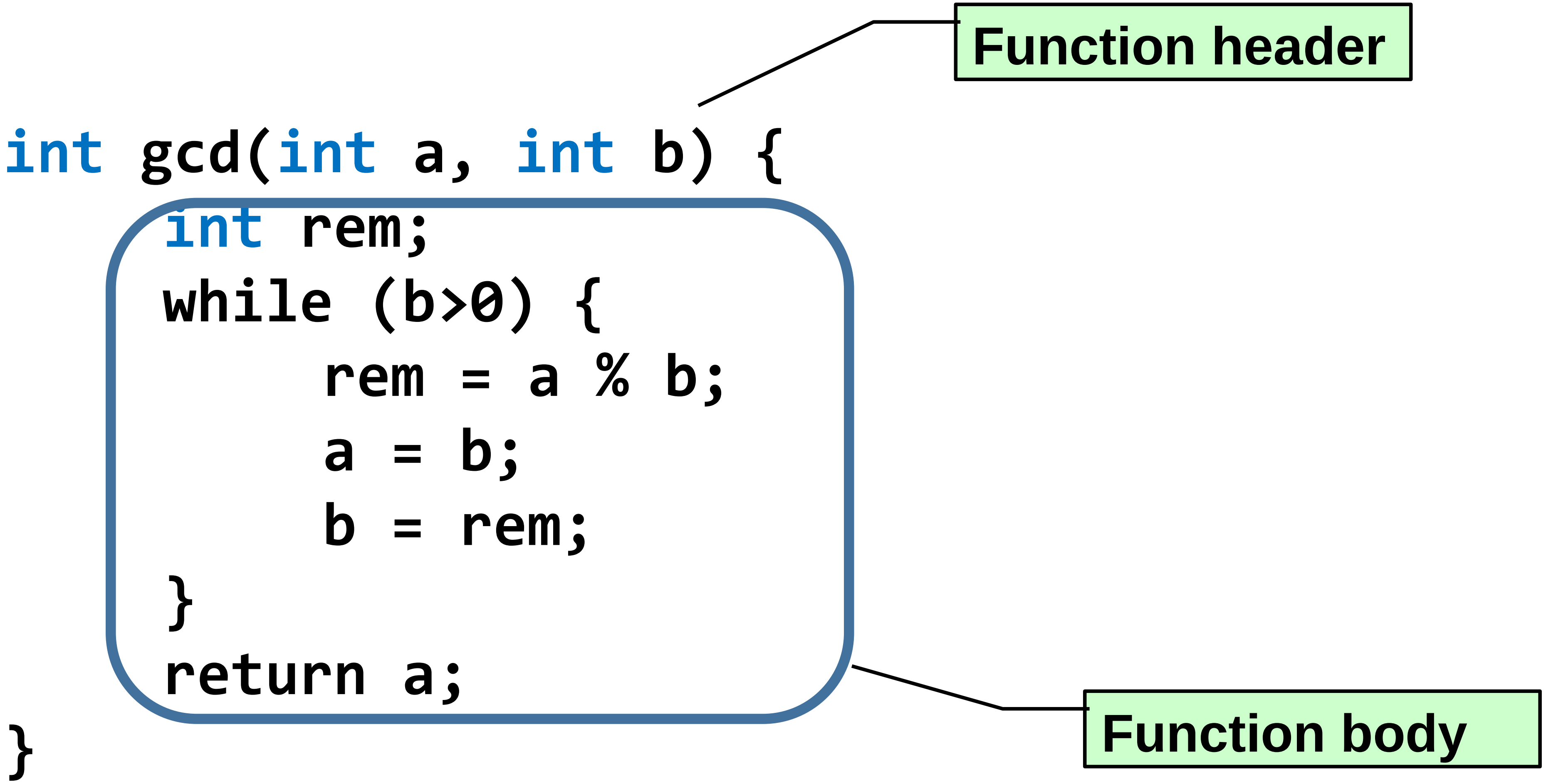
Return type

Parameters with their types
Parameter declaration

Function header

```
int gcd(int a, int b) {
        int rem;
        while (b>0) {
                rem = a % b;
                a = b;
                b = rem;
        }
        return a;
}
```

Components of a User-defined Function Definition

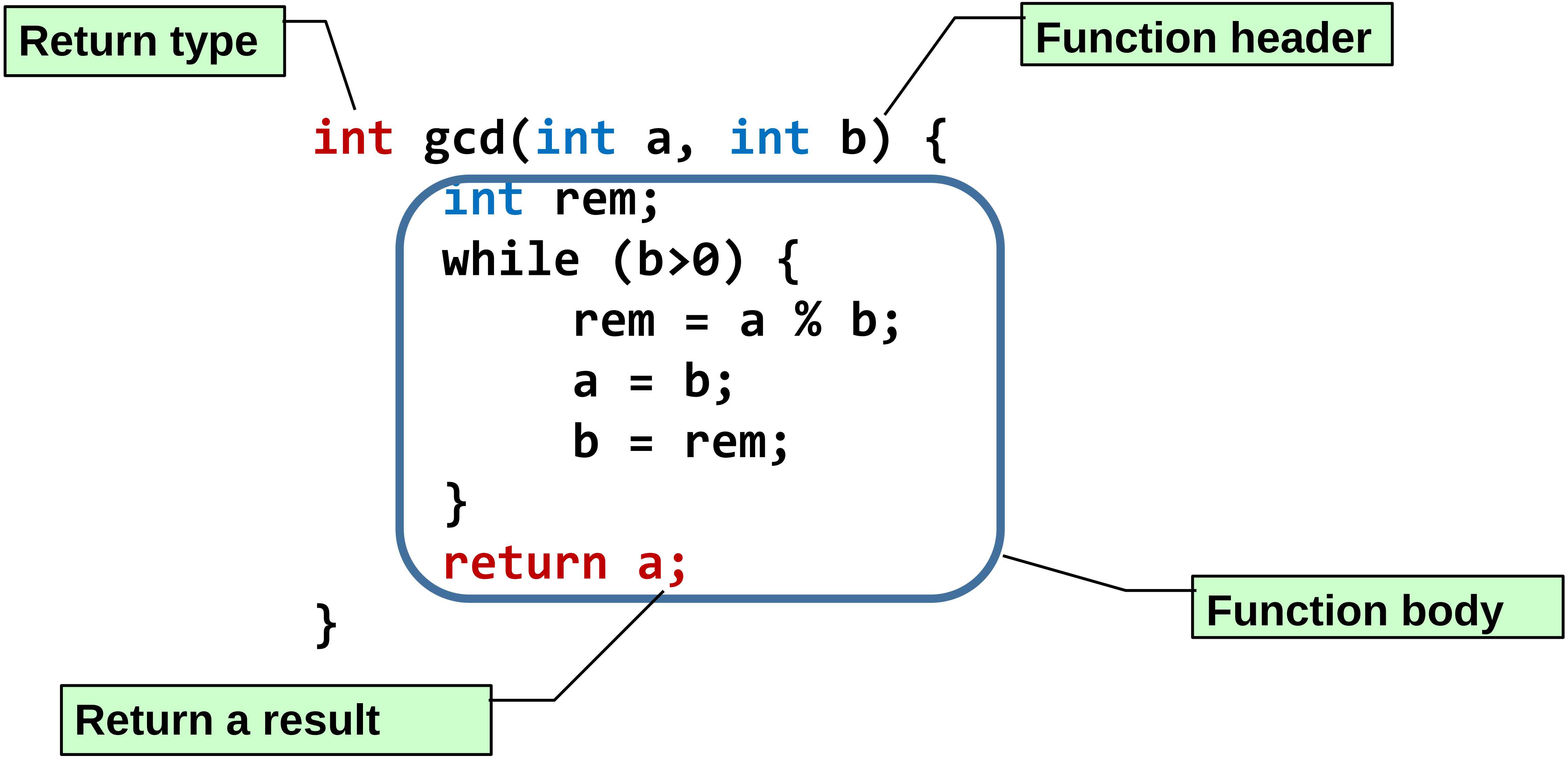


```
int gcd(int a, int b) {
    int rem;
    while (b>0) {
            rem = a % b;
            a = b;
            b = rem;
    }
    return a;
}
```

Function header

Function body

Components of a User-defined Function Definition



```
int gcd(int a, int b) {
    int rem;
    while (b>0) {
          rem = a % b;
          a = b;
          b = rem;
    }
    return a;
}
```

Function Prototype: A Summary of a User-defined Function Definition

```
int gcd(int, int)
```

```
int gcd(int a, int b) {
    int rem;
    while (b>0) {
        rem = a % b;
        a = b;
        b = rem;
    }
    return a;
}
```

C Program Structure with User-defined functions

```
#include <stdio.h>
…
Function prototypes

int main (void) {
    . . .
}

User-defined function
definitions
```

```
#include <stdio.h>

int gcd(int, int);

int main(void) {
      . . .
      d = gcd(m,n) ;
      . . .
}

int gcd(int a, int b) {
      . . .
}
```

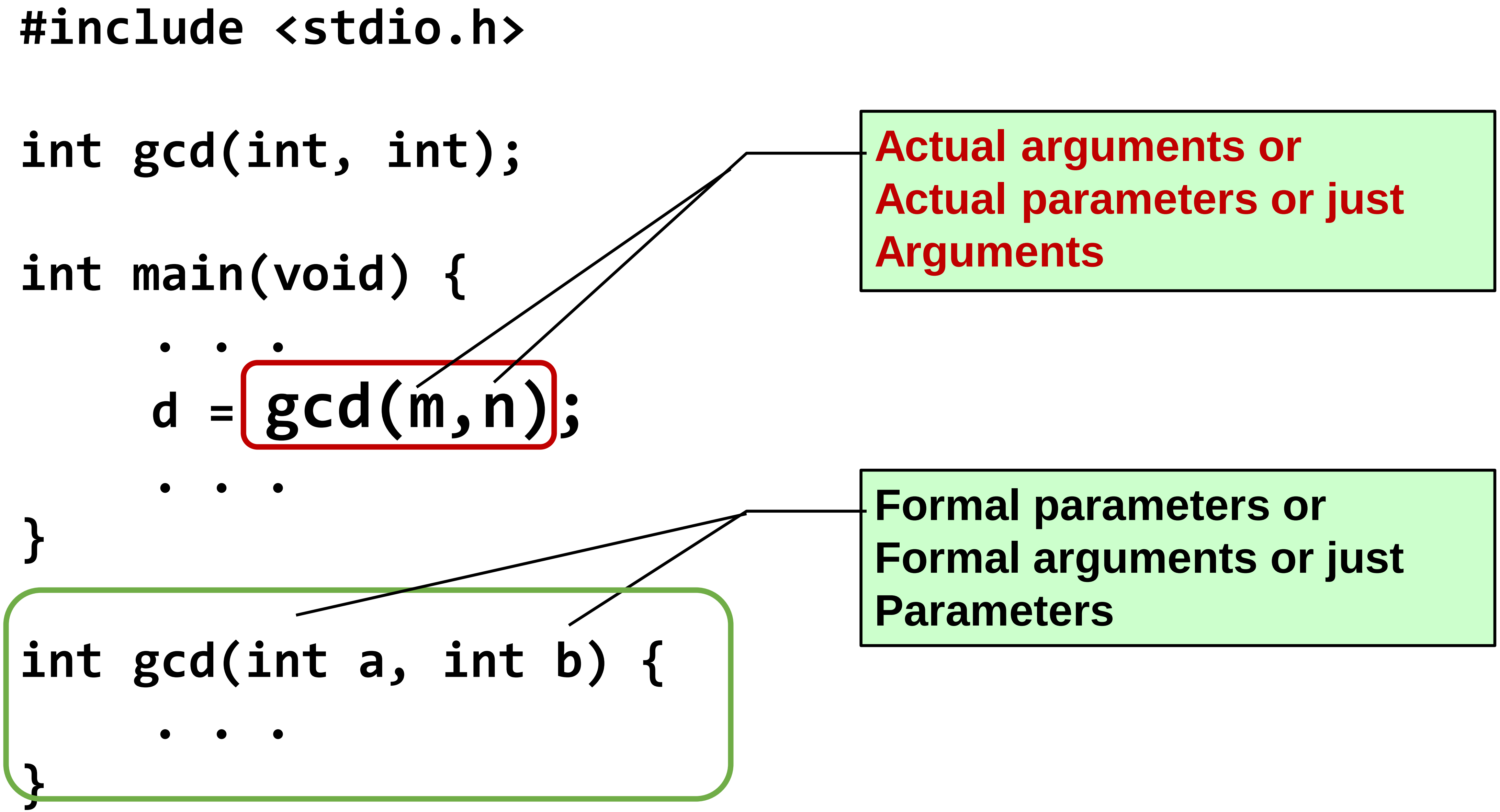
Functions – Define and Use

```
#include <stdio.h>

int gcd(int, int);

int main(void) {
     . . .
     d = gcd(m,n) ;
     . . .
}

int gcd(int a, int b) {
     . . .
}
```

Function application or
Function call or
Function invocation

Function Definition

Functions – Define and Use



```
#include <stdio.h>

int gcd(int, int);

int main(void) {
     . . .
     d = gcd(m,n);
     . . .
}

int gcd(int a, int b) {
     . . .
}
```

Actual arguments or Actual parameters or just Arguments

Formal parameters or Formal arguments or just Parameters

Functions – Notice the Differences

```
#include <stdio.h>

int gcd(int, int);

int main(void) {
     . . .
     d = gcd(m,n) ;
     . . .
}

int gcd(int a, int b) {
     . . .
}
```

Function Prototype:

- Only one identifier exists – the function name
- Shows all parameter types and return type

Function Header:

- Declare function name and parameter names
- Declare all parameter types and return type

Functions – Notice the Differences

```
#include <stdio.h>

int gcd(int, int);

int main(void) {
     . . .
     d = gcd(m,n) ;
     . . .
}

int gcd(int a, int b) {
     . . .
}
```

Function Call:

- **Only function name and argument**
- **Do not show any type information**

Function Header:

- **Declare function name, parameter names**
- **Declare all parameter types and return type**

Formal and Actual Parameters

```
int main(void) {
    int m, n, d1, d2 ;
    . . .
    d1 = gcd(m,n) ;
    d2 = gcd(405,827);
    d1 = d1 + gcd(m,827)
         + gcd(405,n);
    . . .
}
int gcd(int a, int b) {
    int rem ;
    . . .
}
```

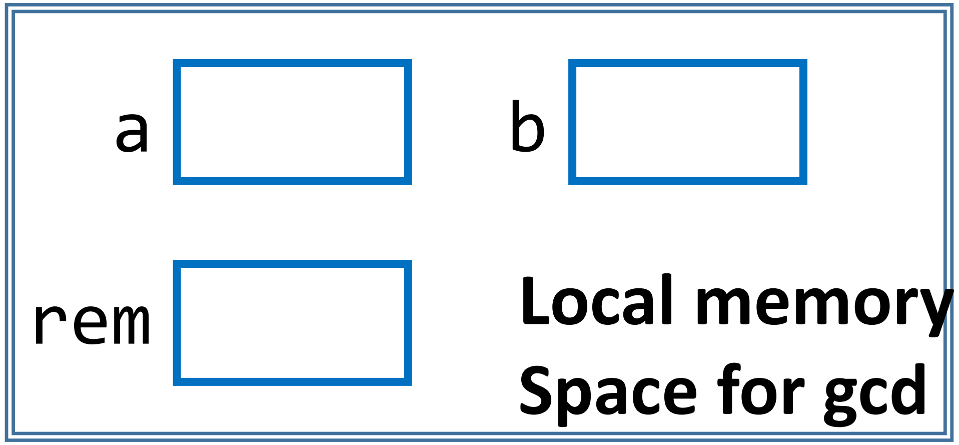
- **One definition, multiple calls**
- **One set of formal parameters (a and b) per function definition**
- **Actual arguments vary from one call to another**

Scope Rules – Local variables

- Formal parameters are local to the function they are declared in.
- Variables declared within a function are local to that function too
- Local variable are only accessible in the body of the function they are declared – Scope rule.

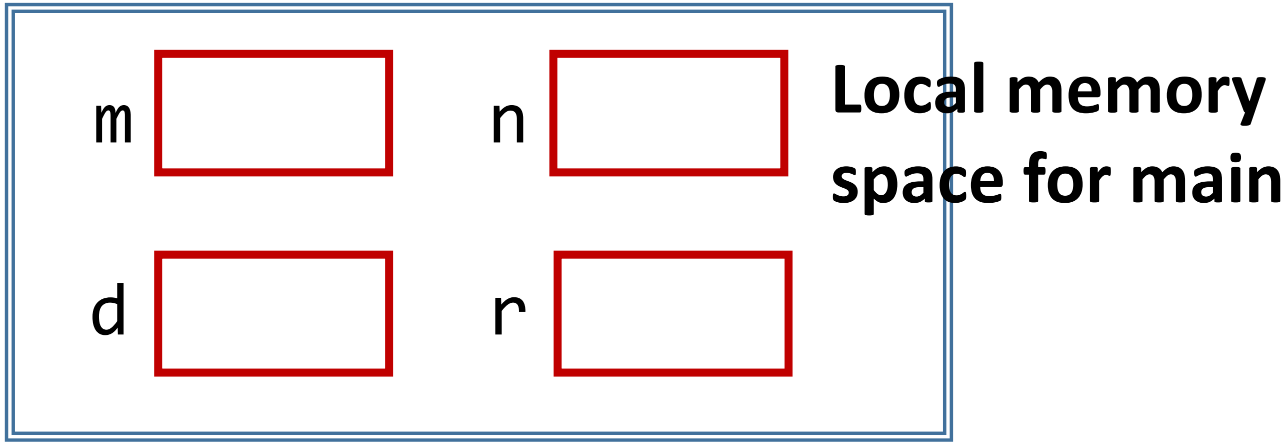
```
int gcd(int a, int b) {
    int rem;
    while (b>0) {
        rem = a % b;
        a = b;
        b = rem;
    }
    return a;
}
```

```c
int gcd(int a, int b) {
    int rem;
    while (b>0) {
        rem = a % b;
        a = b;
        b = rem;
    }
    return a;
}
```



Local memory
Space for gcd

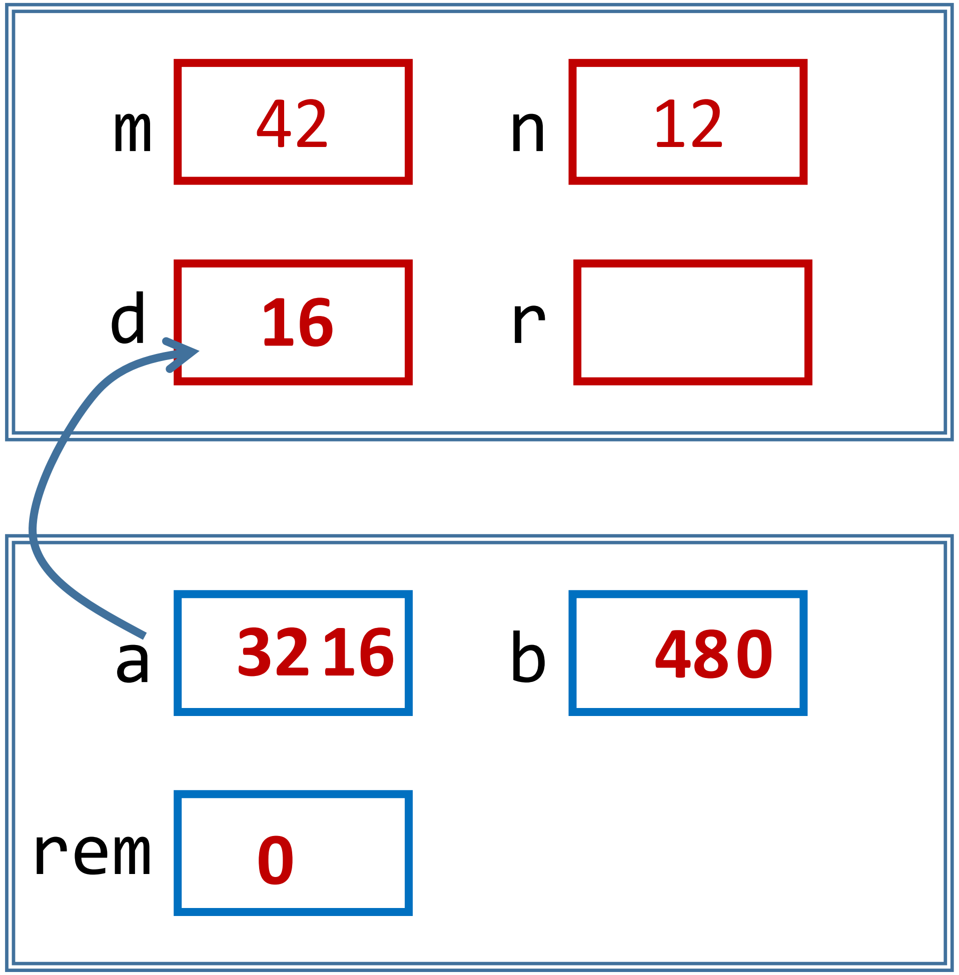
```c
int main(void) {
    int m, n, d, r ;

    . . .
    d = gcd(32,48) ;
    r = gcd(m,n) ;
    . . .
    return 0;
}
```



Local memory
space for main

Execution model

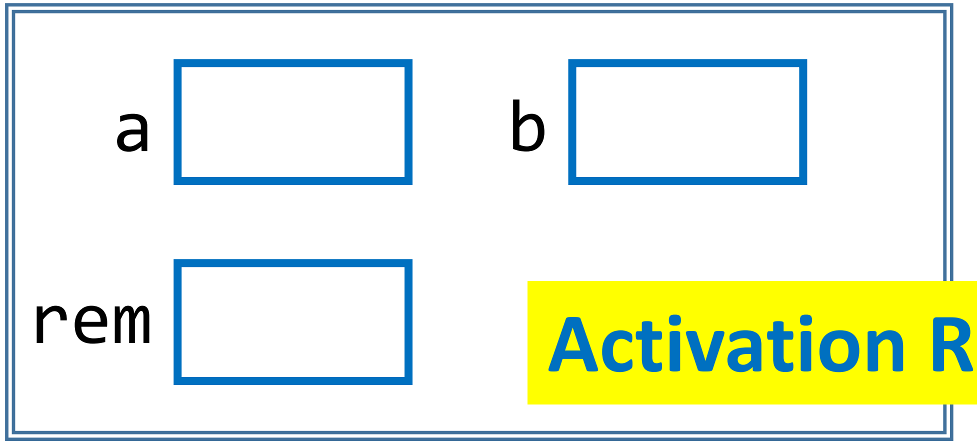
```
int main(void) {
      int m, n, d, r ;

      . . .
      d = gcd(32,48) ;
      r = gcd(m,n) ;
      . . .
      return 0;
}

int gcd(int a, int b) {
      int rem;
      while (b>0) {
```



Execution model

```
int main(void) {
    int m, n, d, r ;

    . . .
    d = gcd(32,48) ;
    r = gcd(m,n) ;
    . . .
    return 0;
}

int gcd(int a, int b) {
    int rem;
    while (b>0) {
```

Local variables for a function is kept in a place called Activation Record. This record is created when the function is called, and deleted when the function execution is completed.

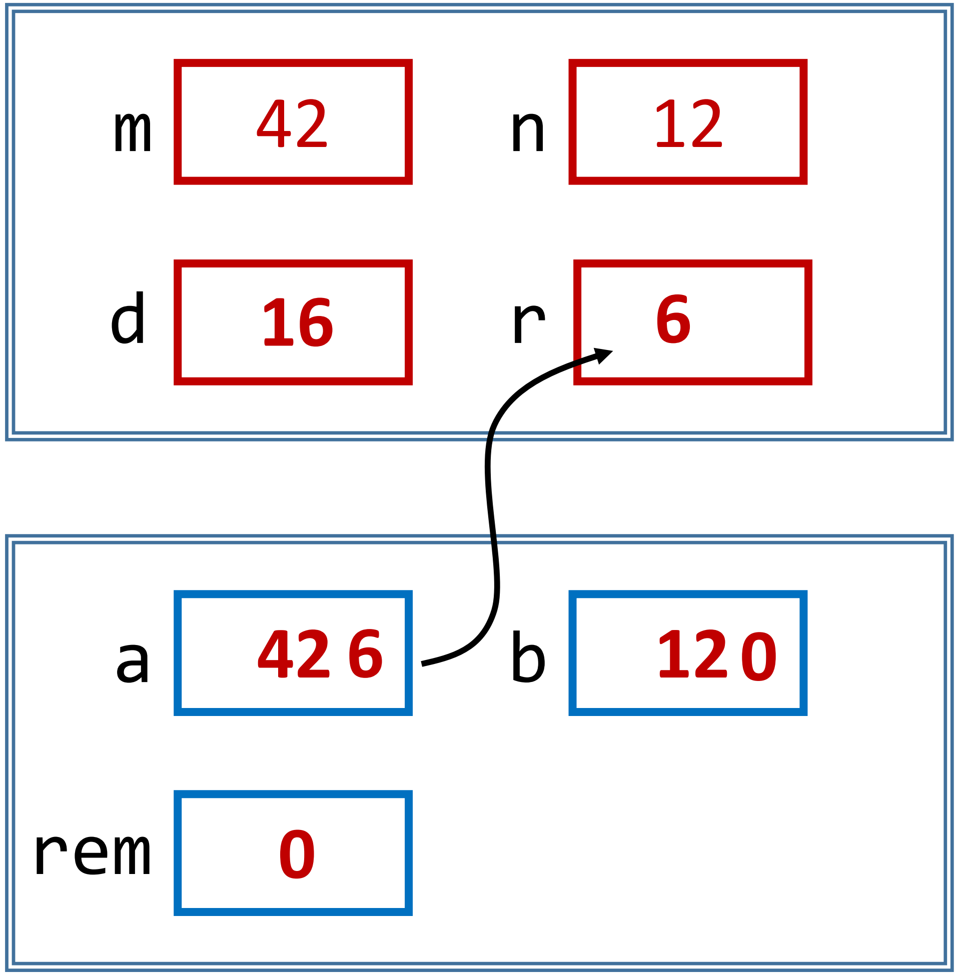


Activation Record

Every time a function is called, a new activation record is created.

Execution model

```
int main(void) {
      int m, n, d, r ;

      . . .
      d = gcd(32,48) ;
      r = gcd(m,n) ;
      . . .
      return 0;
}

int gcd(int a, int b) {
      int rem;
      while (b>0) {
```

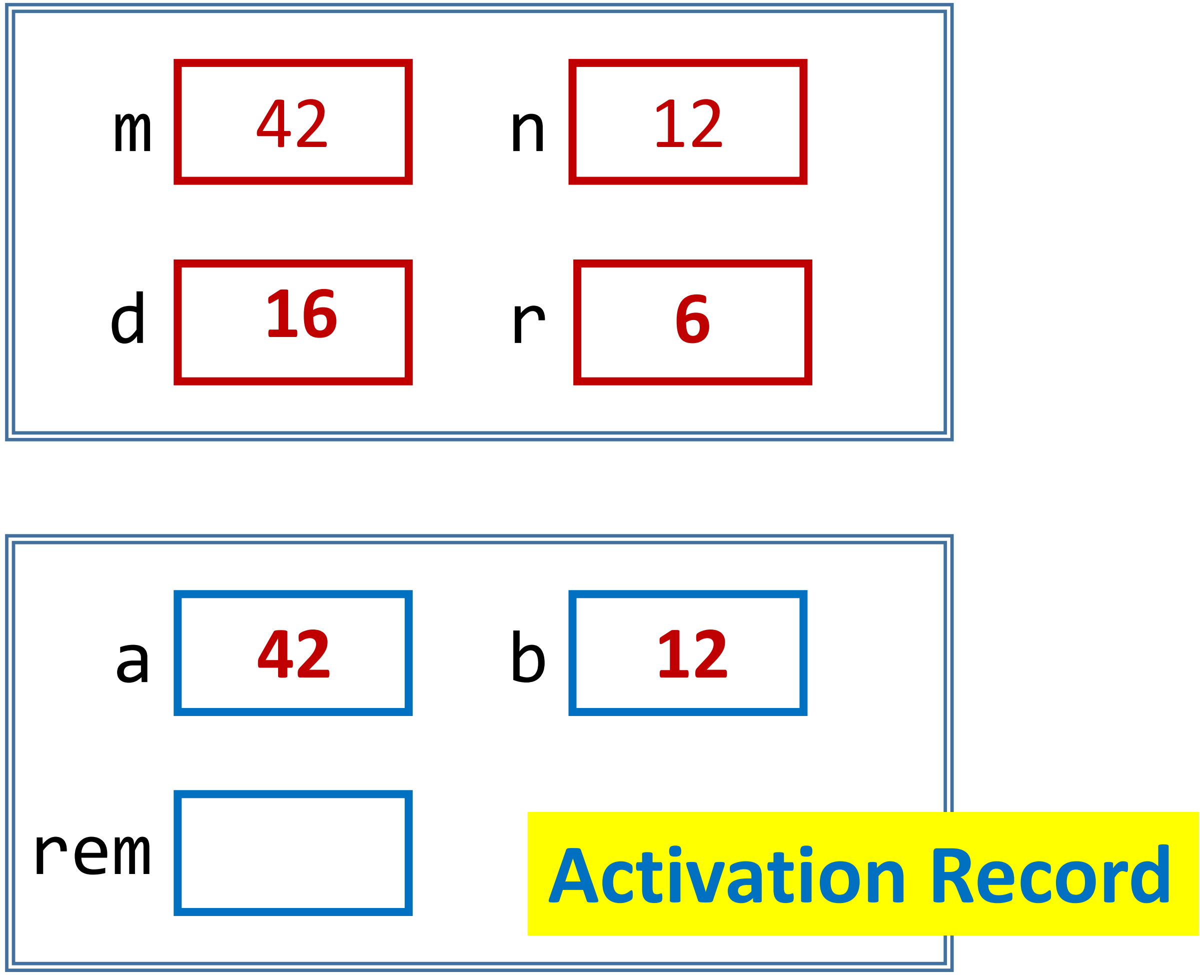


Arguments are Passed-by-Value

```
int main(void) {
     int m, n, d, r ;

     . . .
     d = gcd(32,48) ;
     r = gcd(m,n) ;
```

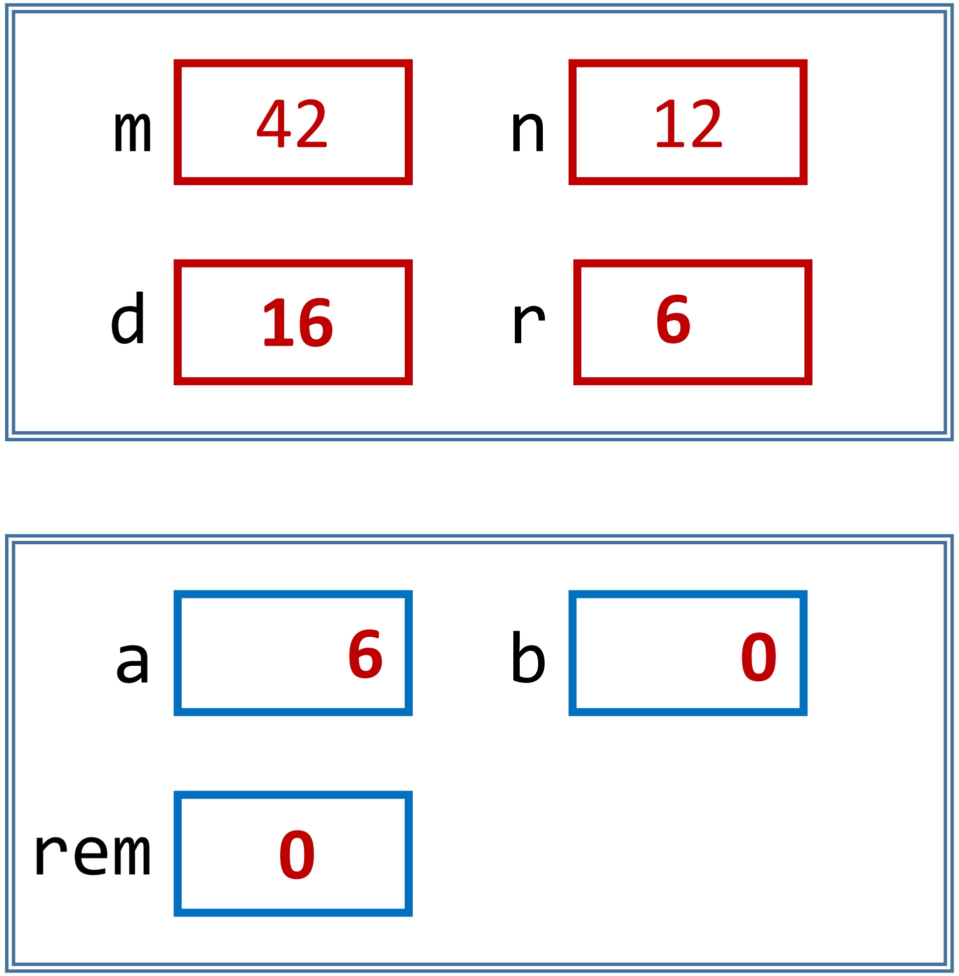
The value of arguments are passed from caller to the callee “gcd”; the result from the callee is “returned” to the caller.

```
     while (b>0) {
```



What’s the big deal?

```
int main(void) {
     int m, n, d, r ;

     . . .
     d = gcd(32,48) ;
     r = gcd(m,n) ;
     . . .
     return 0;
}

int gcd(int a, int b) {
     int rem;
     while (b>0) {
          rem = a % b;
```



m 42 n 12

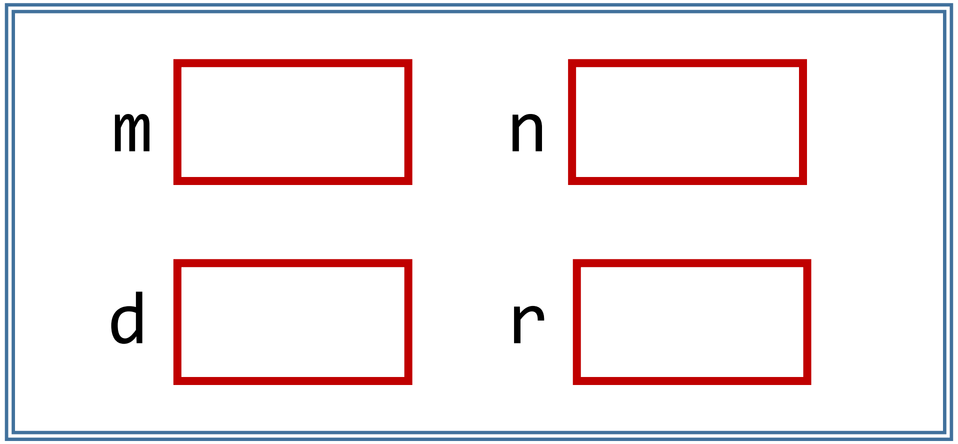
d 16 r 6

a 6 b 0

rem 0

After executing gcd(m,n), what are the values of m and n?

```
int main(void) {
    int m, n, d, r ;

    . . .
    d = gcd(32,48) ;
    r = gcd(m,n) ;
    . . .
    return 0;
}
```



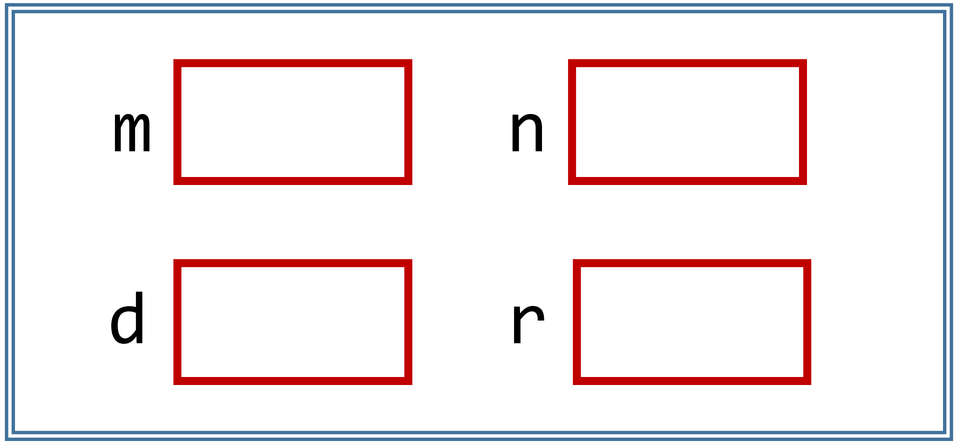
```
int gcd(int a, int b) {
    int rem;
    while (n>0) {
        rem = m % n;
        m = n;
        n = rem;
    }
    return m;
}
```

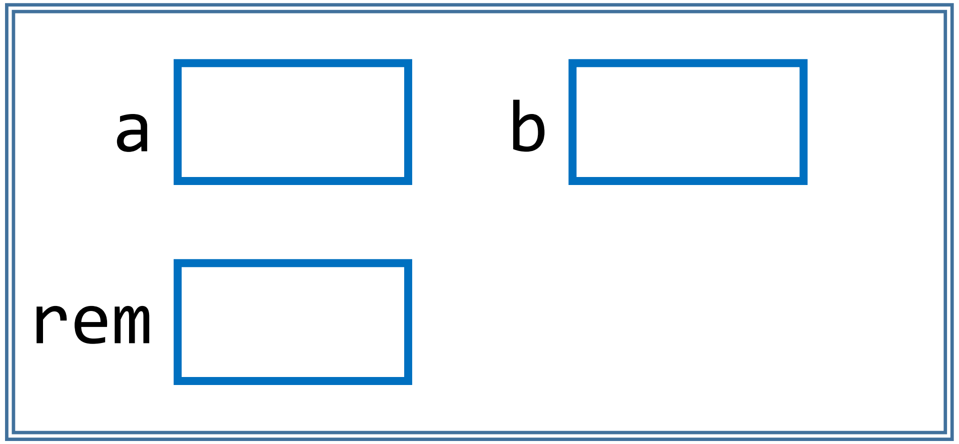
If I use variable m and n in the gcd function, then after executing gcd(m,n), what are the values of m and n in the main function?

```
int main(void) {
      int m, n, d, r ;

      . . .
      d = gcd(32,48) ;
      r = gcd(m,n) ;
      . . .
      return 0;
}
```

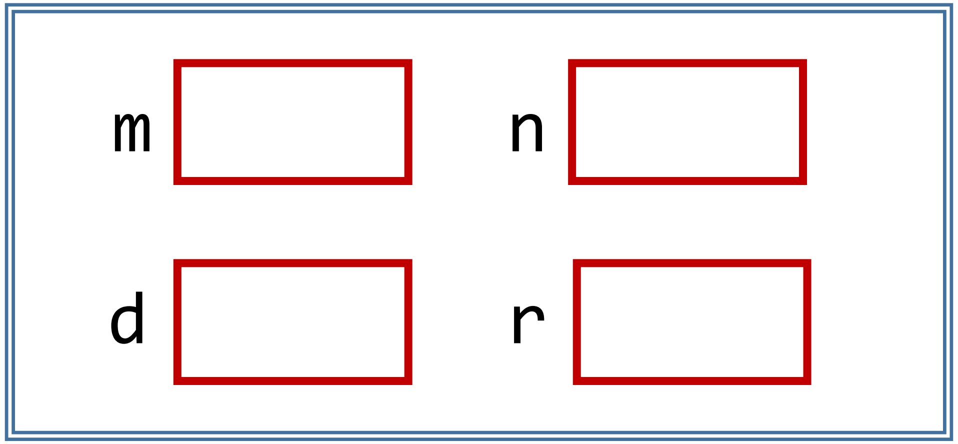
```
int gcd(int a, int b) {
      int rem;
      while (n>0) {
            rem = m % n;
            m = n;
            n = rem;
      }
      return m;
}
```

Access to m and n are illegal because of scope rule!





```c
int main(void) {
    int m, n, d, r ;

    . . .
    d = gcd(32,48) ;
    r = gcd(m,n) ;
    . . .
    return 0;
}
```



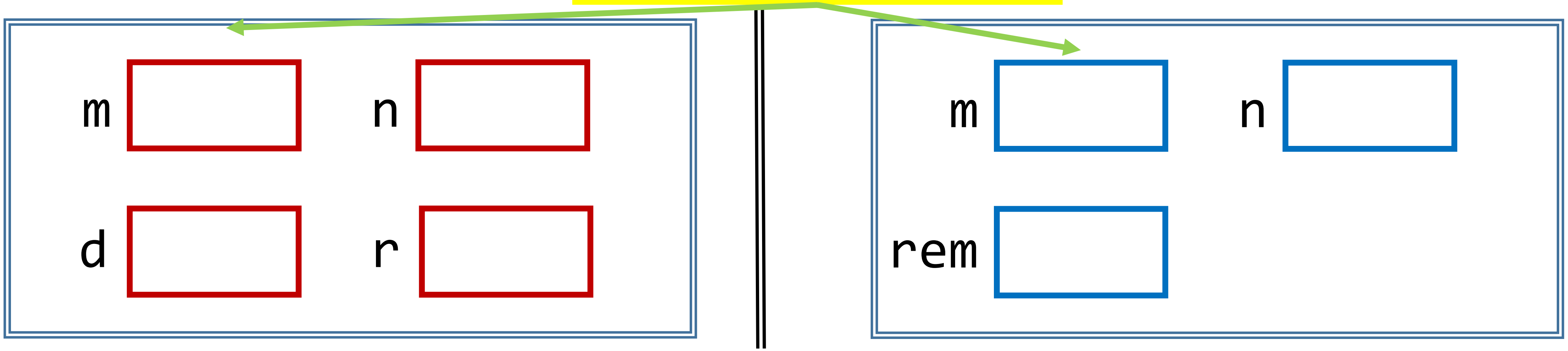
```c
int gcd(int m, int n) {
    int rem;
    while (n>0) {
        rem = m % n;
        m = n;
        n = rem;
    }
    return m;
}
```

If I change the parameters of gcd to m and n, then after executing gcd(m,n), what are the values of m and n in the main function?

```
int main(void) {
    int m, n, d, r ;

    . . .
    d = gcd(32,48) ;
    r = gcd(m,n) ;
    . . .
    return 0;
}
```

```
int gcd(int m, int n) {
    int rem;
    while (n>0) {
        rem = m % n;
        m = n;
        n = rem;

    eturn m;
```

Call by Value:
We have two distinct versions of m and n, living in different spaces.



Summary

- **Better Understanding of Structure of C programs**
  - Constants and Variables
  - Input/Output statements: `printf` and `scanf`
- **Modular Design with User-defined Functions**
  - Function Definitions and Function Prototypes
  - Function Application/Call
  - Scoping Rules and Pass-by-Value Parameter Passing
  - Execution model for function calls