Ch.9 Functions and Variables

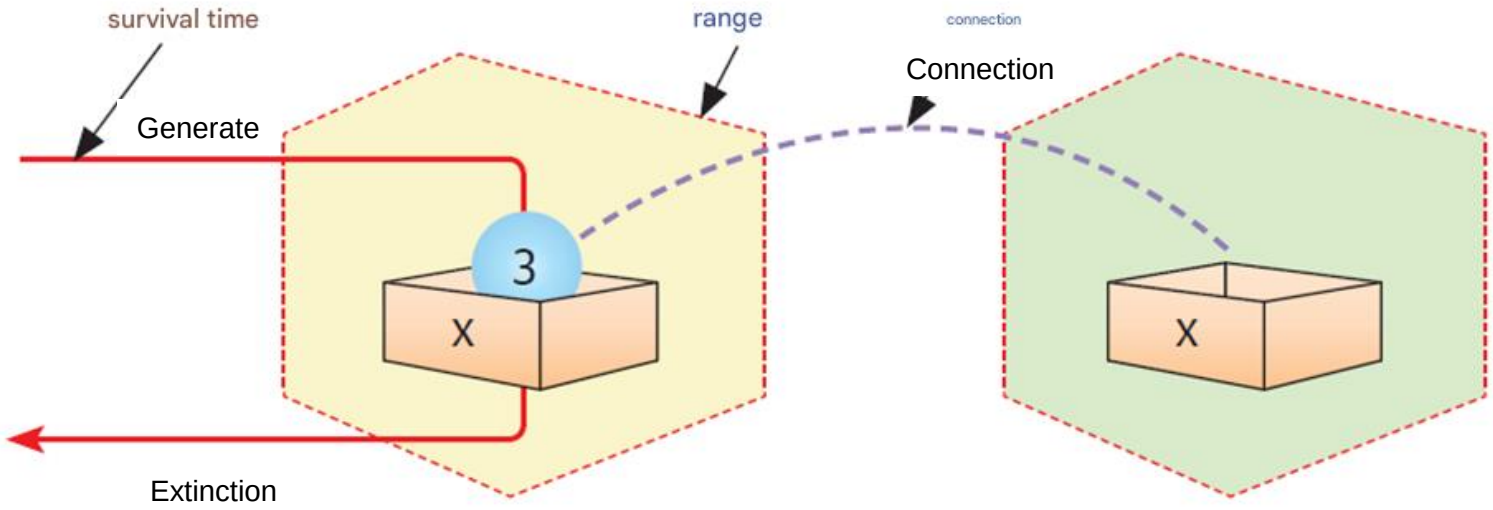
What you will learn in this chapter

- Understanding the concept of repetition
- Variable properties
- Global and local variables
- Automatic and static variables
- Recursive call

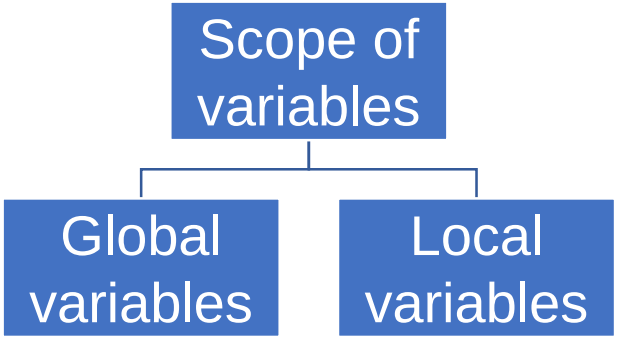
In this chapter, we will focus on the relationship between functions and variables. We will also look at recursive calls, where a function calls itself .

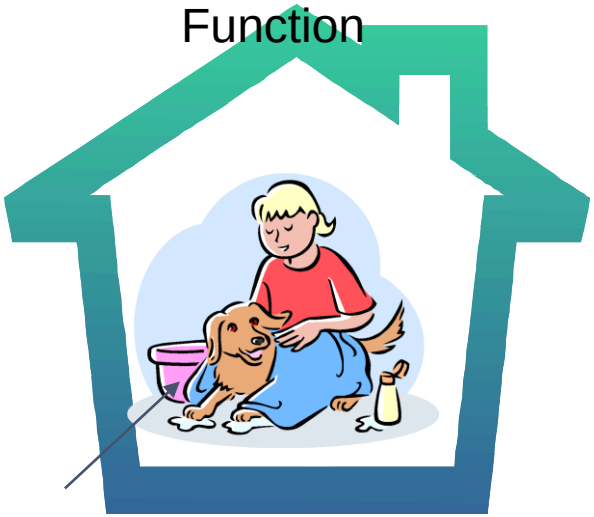
Variable properties

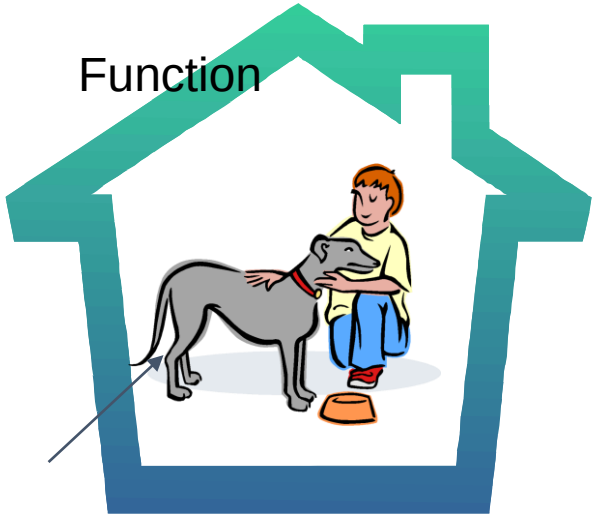
- **Variable properties :** name , type , size , value + range , life time , linkage
  - **Scope :** The region of a program where a variable is accessible — its *visibility*.
  - **Lifetime :** The duration for which a variable exists in memory.
  - **Linkage :** Indicates whether a variable is accessible from other files or translation units.



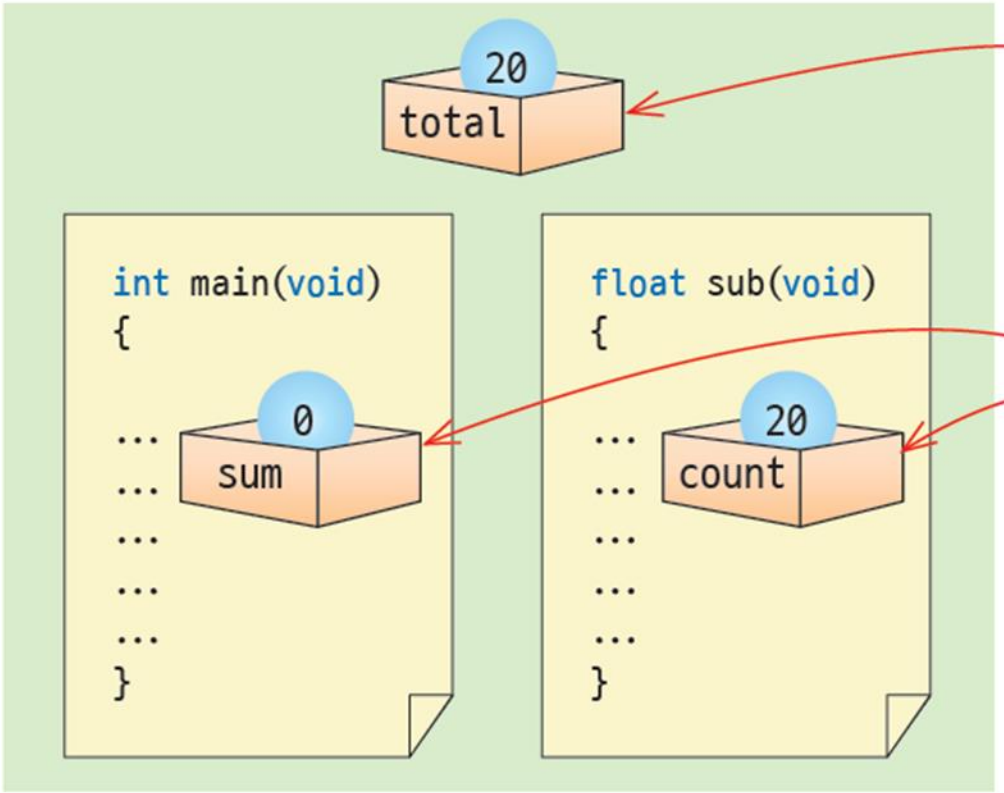
Scope of variables







Global Variables and local variables



Global variables:
defined outside of a function

Local variables:
defined inside of a function

Local variables

- **A local variable** is a variable declared within a block.

```
int sub(void)
{
        int x = 0;

        while(flag!= 0){
            int y;
            ...
        }

        y = 0; // Error!!
        ...
}
```

The scope in which local variable x can be used

The scope in which global variable x can be used

Error because y was used outside the block where it was declared

Local variables must not leave the block in which they are declared.

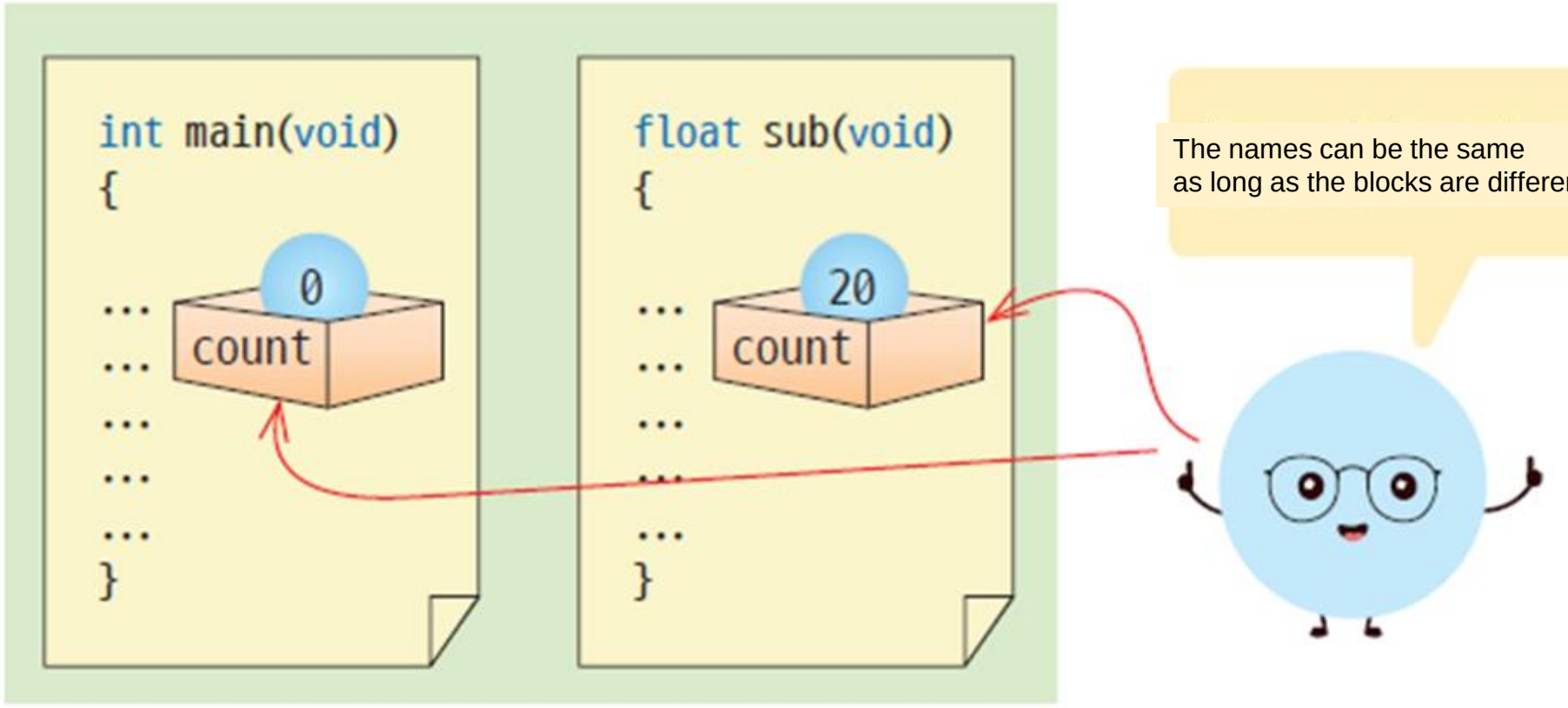
Local variable declaration location

- In C , it can be declared anywhere inside a block !!

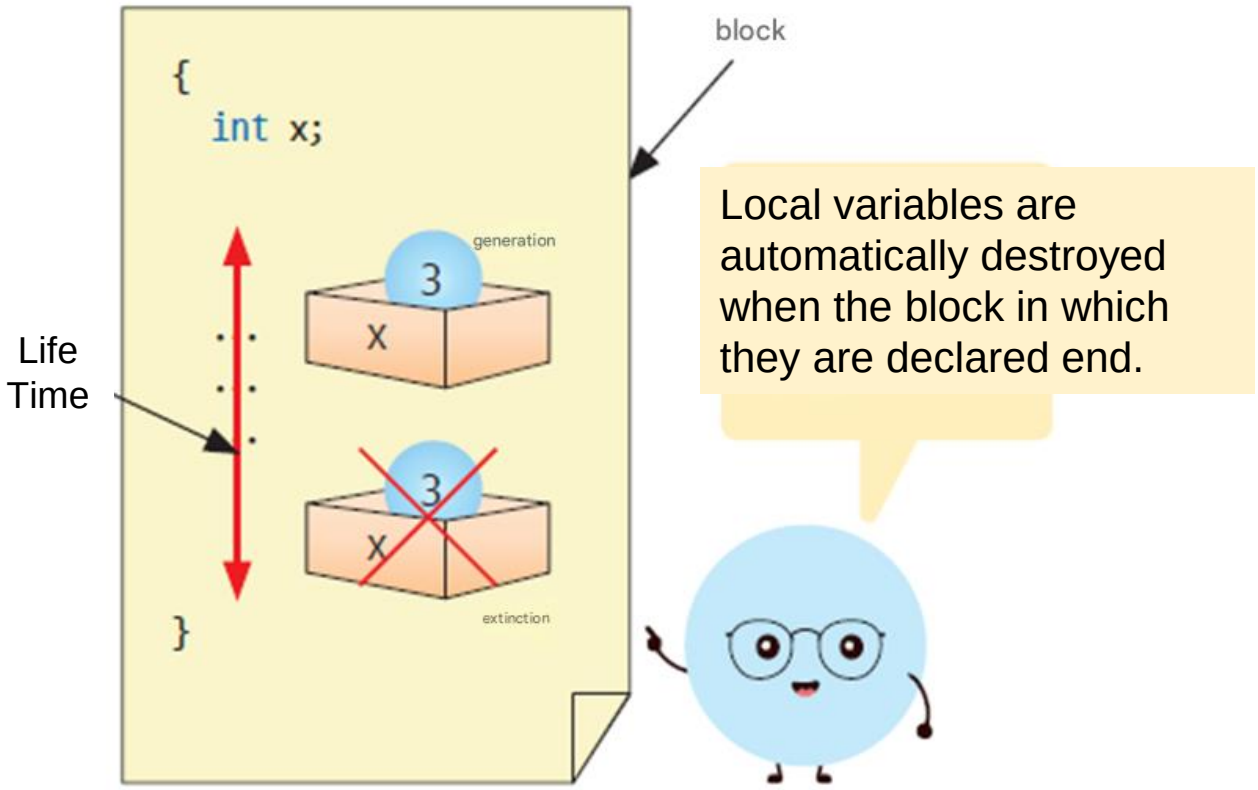
```
while(1) {
        ...
        ...
        int sum = 0;
        ...
}
```

You can declare any number of local variables, even in the middle of a block

Local variables with the same name



Life time of local variables

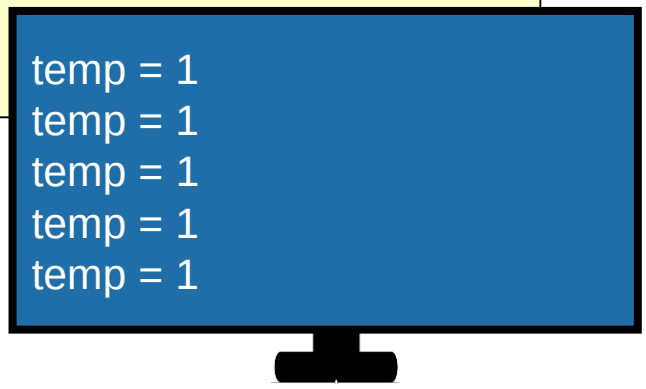


Local variable example

```
#include < stdio.h >

int main( void )
{
        int i ;

        for ( i = 0; i < 5; i ++)
        {
                int temp = 1;
                printf ( "temp = %d\n" , temp);
                        temp++;
        }
       return 0;
}
```

Whenever each block is called, temp is created and initialized (Different address)

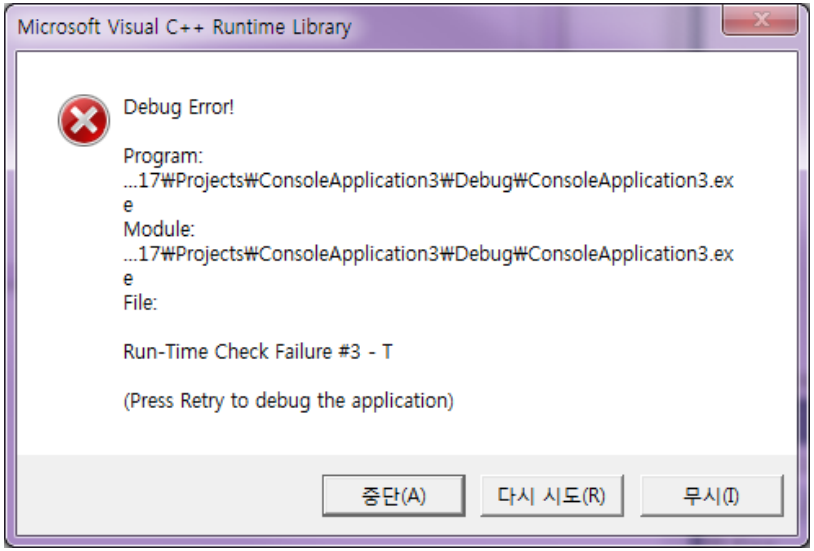
* The variable seems to have the same address each time because the compiler optimizes memory usage by reusing the same stack space for variables with the same scope



Initial value of local variable

```
#include < stdio.h >
int main( void )
{
        int temp;
        printf ( "temp = %d\n" , temp);
        return 0;
}
```

Since it is not initialized, it has a garbage value.



Function parameters

- Parameters defined in the header part of a function are also a type of local variable. That is, they have all the characteristics of local variables.
- What makes it different from local variables is that they are initialized with the argument values when the function is called.

```
int inc ( int counter )
{
    counter++;
    return counter;
}
```

Parameters are also a kind of local variable

Function parameters

```
#include < stdio.h >
int inc ( int counter);

int main( void )
{
        int i ;

        i = 10;
        printf ( " Before calling the function i =%d\n" , i );
        inc ( i );
        printf ( " After calling the function i =%d\n" , i );
        return 0;
}
void  inc ( int counter)
{
        counter++;
}
```

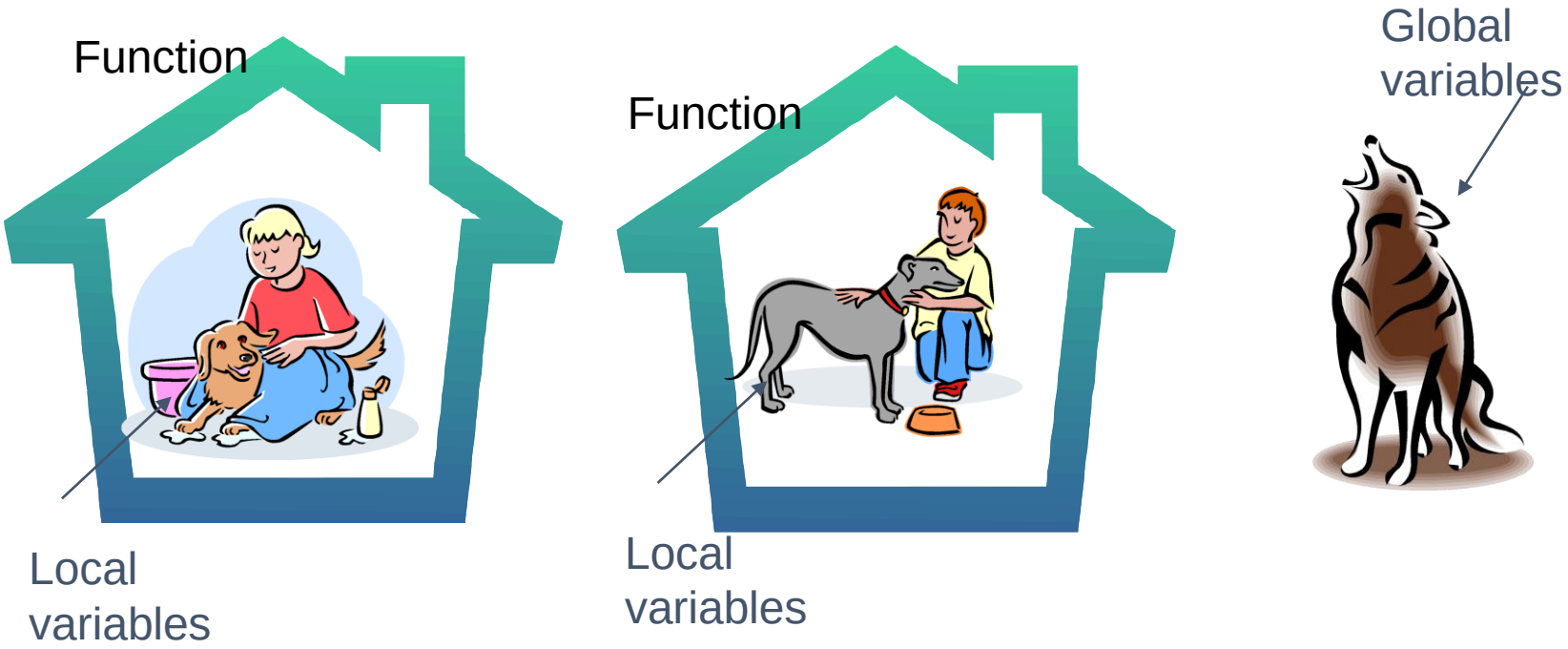
Call by value

Parameters are also a type of local variable

```
Before calling a function i =10
After calling the function i =10
```

Global variables

- **A global variable** is a variable declared outside any function.
- The scope of a global variable is the entire source file.

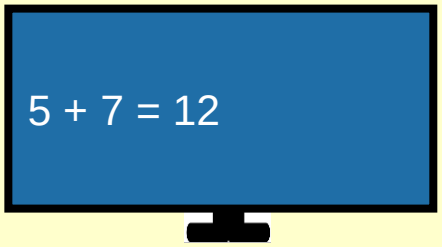


Initial values and life time of global variables

Scope of global variables

```
#include <stdio.h>
int A;
int B;
int add()
{
        return A + B;
}

int main( void )
{
        int answer;
        A = 5;
        B = 7;
        answer = add();
        printf (“ % d + % d = % d\n”, A, B, answer);
        return 0;
}
```

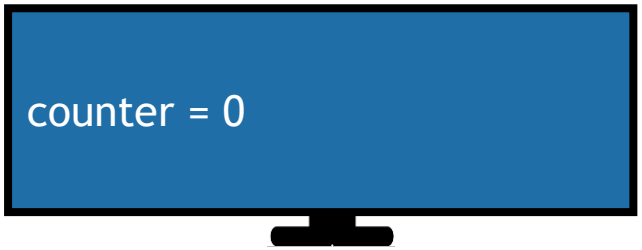
Global variables
The initial value is 0



Global Initial value of variable

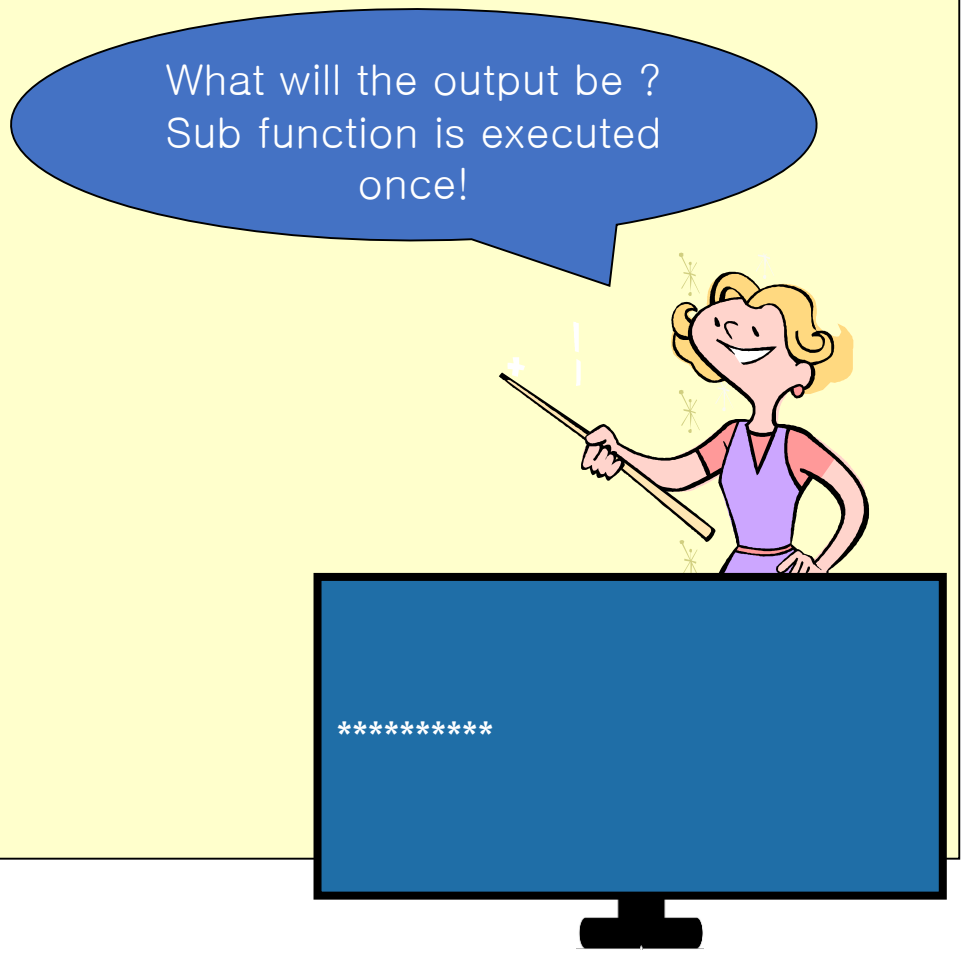
```
#include < stdio.h >

int counter;

int main( void )
{
        printf ( "counter = % d\n" , counter);
        return 0;
}
```

Global variables are initialized to 0 by the compiler when the program runs.



Use of global variables

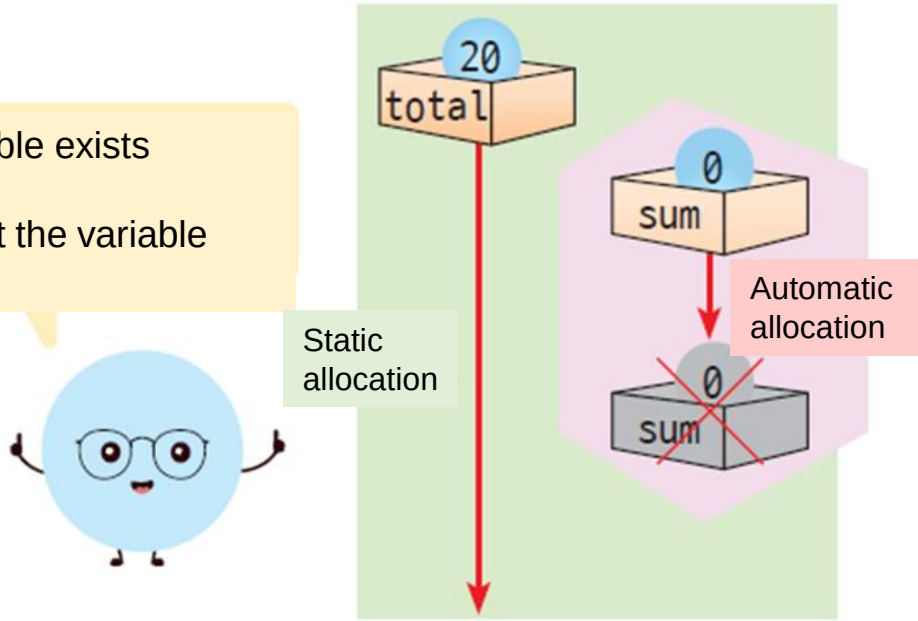
```
#include < stdio.h >

int x;
void sub();

int main( void )
{
        for (x = 0; x < 10; x++)
                sub();
}

void sub()
{
        for (x = 0; x < 10; x++)
                printf ( "*" );
}
```



Lifetime of a variable

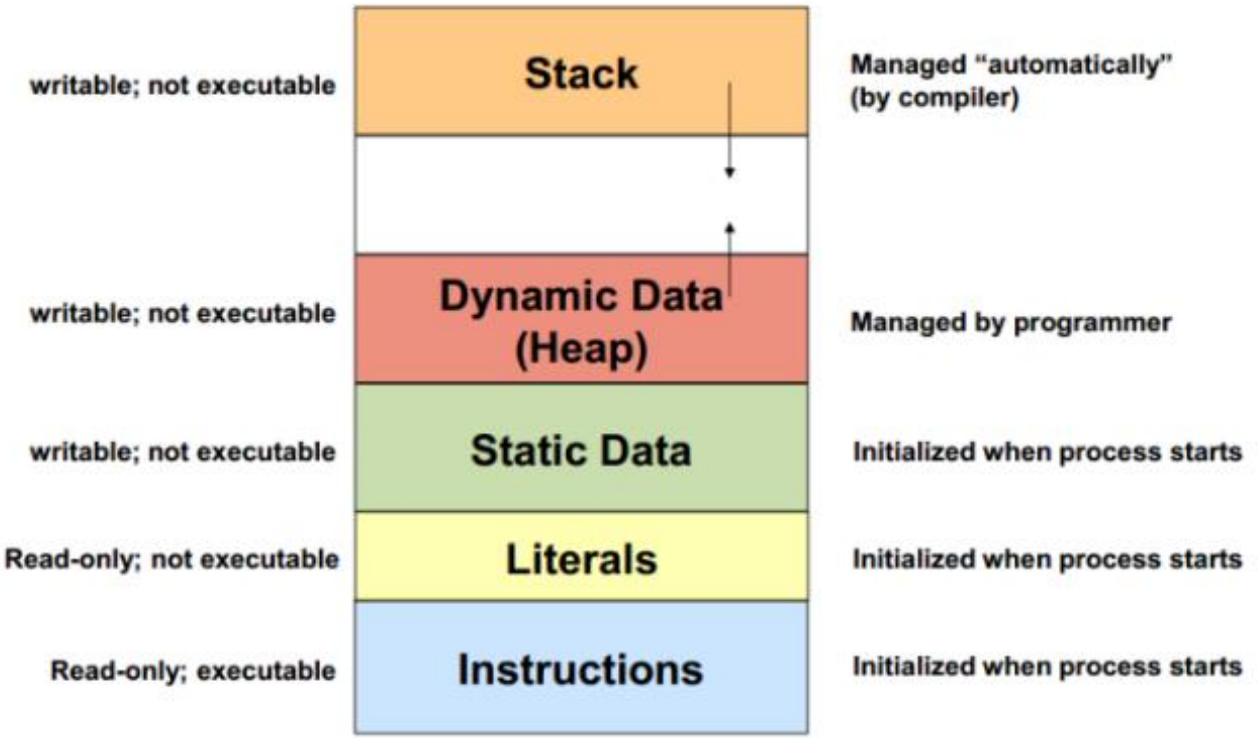
- Static allocation :
  - Keep it alive while the program runs
- Automatic allocation :
  - Created when entering a block
  - Destroys when exiting the block

Static allocation means that the variable exists throughout the execution time,
while automatic allocation means that the variable is destroyed when the block ends.



Lifetime of a variable

- C provides four main storage class specifiers.
  They determine where variables/functions are stored, how long they exist, and their scope/linkage.
  - auto
  - register
  - static
  - extern



static & extern

Keyword	Meaning	Effect
`static`	Local to the file (for global variables) or function (for local variables)	Limits visibility **inside the same file**; lifetime is the **entire program run**
`extern`	Declares a variable or function that is **defined elsewhere**	Tells the compiler "this variable is somewhere else" (another file)

- `static` = **"only usable inside this file"** (or **"remember between calls"** for local variables)
- `extern` = **"this is declared elsewhere, trust me"**

static & extern

1. `static`

Global scope (`file`):
If you declare a global variable or function `static`, it is **private to that `.c` file** — it cannot be seen or used by other files.

```
// file1.c
static int counter = 0; // Only visible in file1.c
```

Local scope (`function`):
If you declare a **local** variable `static`, it **keeps its value** between function calls.

```
void foo() {
    static int x = 0;
    x++;
    printf("%d\n", x);
}
```

Every time `foo()` runs, `x` remembers its previous value instead of resetting.

static & extern

2. `extern`

- Used to **declare** a global variable or function that is **defined in another file**.

```
// file1.c
int global_value = 42; // Define it
```

```
// file2.c
extern int global_value; // Just tell compiler it exists
void use_value() {
    printf("%d\n", global_value);
}
```

Without `extern`, the compiler would not know what `global_value` is in `file2.c`.

Visibility

- static
  - Inside a function : The variable **remembers its value** between function calls.

```
void counter() {
    static int count = 0;
    count++;
    printf("%d\n", count);
}
```

  - Outside a function: Limits **visibility to the same file** (not accessible from other files)

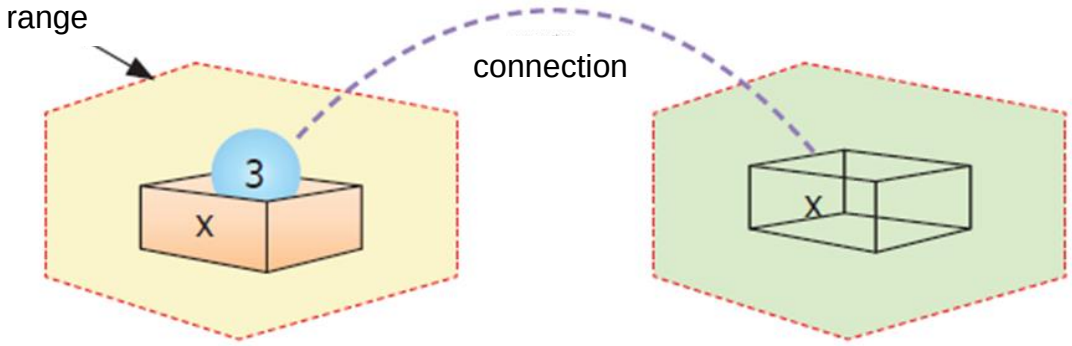
```
static int globalVar = 100;
```

- extern
  - Used to declare a **variable or function** that is **defined in another file**
  - Used for **cross-file access**

```
// file1.c
int x = 10;

// file2.c
extern int x;  // Use the variable from file1.c
```

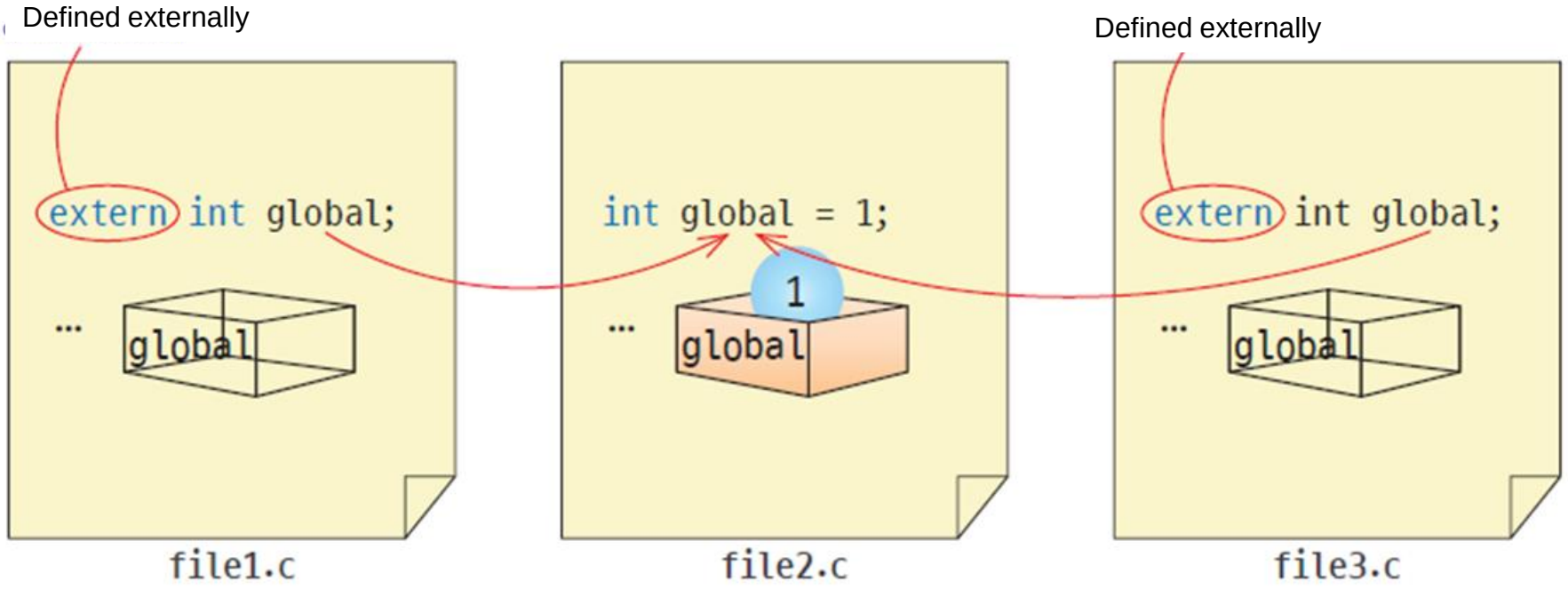
connection

- *Linkage :* Linking variables belonging to different scopes
  - External connection
  - Internal connection
  - No connection
- Only global variables can have associations .

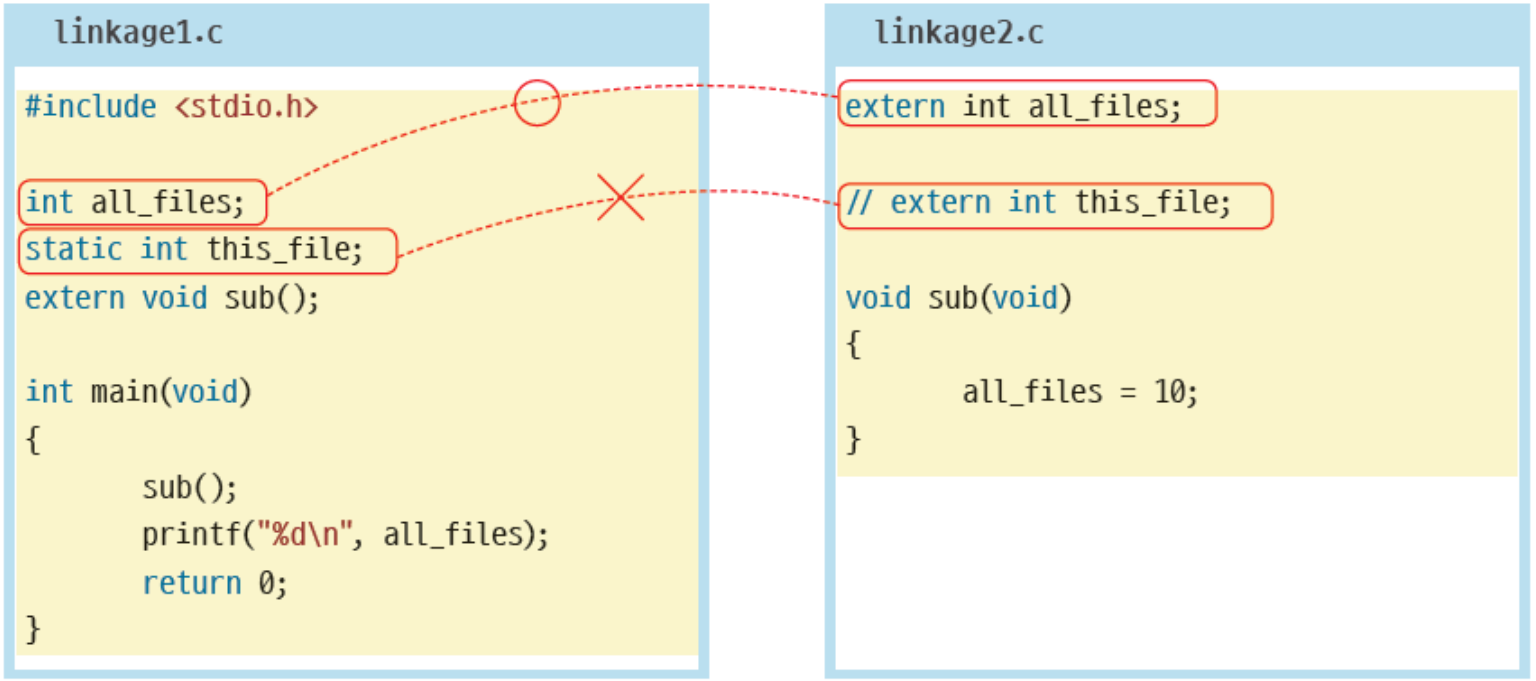


External connection

- Global variables using extern



Connection example



static in front of function

main.c

```c
#include <stdio.h>


//extern void f1();
extern void f2();

int main(void)
{
        f2();
        return 0;
}
```

sub.c

```c
#include <stdio.h>

static void f1()
{
        printf("f1()가 호출되었습니다.\n");
}

void f2()
{
        printf("f2()가 호출되었습니다.\n");
}
```

Private function

Referencing global variables using extern in a block

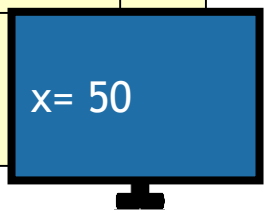
- extern is also used to access global variables from a block .

```
#include < stdio.h >
int x = 50;

int main( void )
{
        int x = 100;
        {
                extern int x;
                printf( "x= %d\n" , x);
        }
        return 0;
}
```

Global variable: scope (entire this file)

Local variable: scope (in main function)
Local variable → hides the global x (Shadowing)

Refer to global variable : scope (entire this file)
you can re-access the global variable using the extern keyword



When you declare a **local variable** with the same name as a **global variable**, the local one takes **precedence** inside its scope, and the global variable becomes **inaccessible (hidden)** within that block.

Variable parameters

```
#include < stdio.h >
#include < stdarg.h >
int sum( int , ... );
int main( void )
{
        int answer = sum( 4, 4, 3, 2, 1 );
        printf ( " The sum is %d .\n" , answer );
        return ( 0 );
}
int sum( int num , ... )
{       int answer = 0;
        va_list argptr ;
        va_start ( argptr , num );
        for ( ; num > 0; num -- ) {
            int temp = va_arg ( argptr , int );
            printf("va_arg num=%d (%d)\n", num, temp);
            answer += temp;
        }
        va_end ( argptr );
        return ( answer );
}
```

The sum is 10 .

Number of parameters

Variable parameters

- A feature where the number of parameters can vary.

```
int sum ( int num , ... )
```

The number of parameters may change with each call.

```
void print_numbers(int count, ...) {
    va_list args;          // 1) declare the list
    va_start(args, count); // 2) start reading arguments

    for (int i = 0; i < count; i++) {
        int num = va_arg(args, int);  // 3) read next argument
        printf("%d ", num);
    }

    va_end(args);          // 4) clean up
    printf("\n");
}

int main() {
    print_numbers(3, 10, 20, 30);
    print_numbers(5, 1, 2, 3, 4, 5);
    return 0;
}
```

Variable parameters

- Basic Syntax Pattern of va_list in C

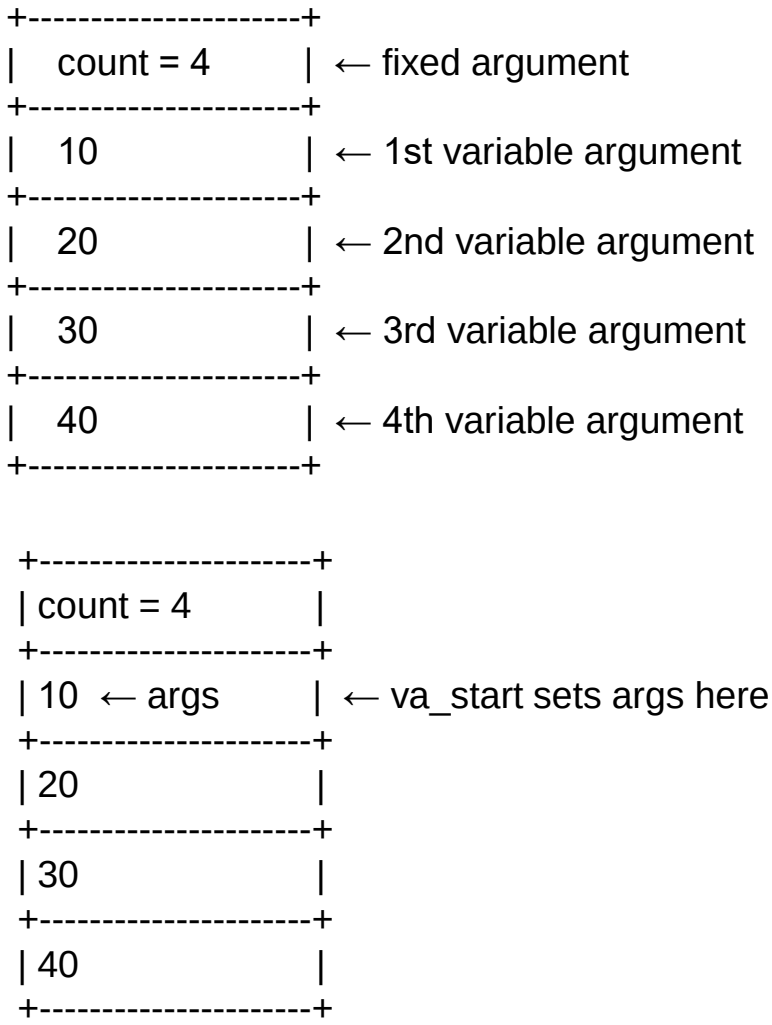
The typical usage of variable argument functions in C follows this standard pattern:

1. Declare a va_list variable
2. Initialize it with va_start()
3. Access each argument with va_arg()
4. Finish with va_end()

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

void print_numbers(int count, ...) {
    va_list args;
    va_start(args, count);

    for (int i = 0; i < count; i++) {
        int value = va_arg(args, int);
        printf("%d ", value);
    }

    va_end(args);
}
```

Variable parameters

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

void print_numbers(int count, ...) {
    va_list args;
    va_start(args, count);

    for (int i = 0; i < count; i++) {
        int value = va_arg(args, int);
        printf("%d ", value);
    }

    va_end(args);
}
print_number(4, 10, 20, 30, 40);
```



Main function with variable arguments


```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Number of arguments: %d\n", argc);

    for (int i = 0; i < argc; i++) {
        printf("Argument %d: %s\n", i, argv[i]);
    }

    return 0;
}
```

```
gcc args.c -o args
./args hello world 123
```

What is recursion ?

Important

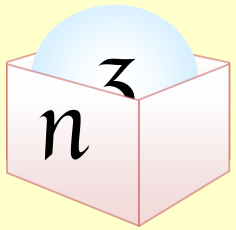
- A function can also call itself. This is called recursion.

$$n! = \begin{cases} 1 & n = 0 \\ n^*(n-1)! & n \geq 1 \end{cases}$$

Calculating factorial

- Factorial Programming : Calculate the factorial of (n-1)!
  by calling the function you are currently writing again (recursive call)

```
int factorial( int n)
{
    if ( n <= 1 ) return (1);
    else return (n * factorial(n-1) );
}
```



Structure of printing 1~n function

- The recursive algorithm consists of a part that recursively calls itself and a part that stops the recursive call.

1 2 3 4 5

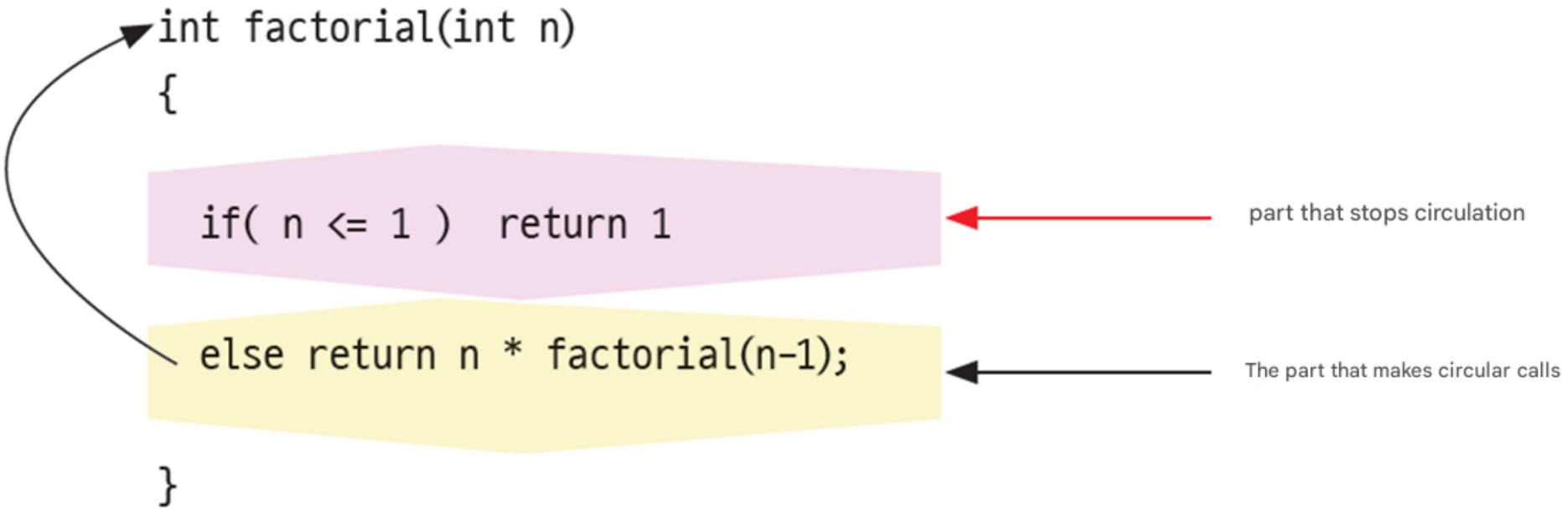
```c
#include <stdio.h>

void print_numbers(int n) {
    if (n == 0)
        return;     // base case (recursive end)

    print_numbers(n - 1);  // recursive call
    printf("%d ", n);
}

int main() {
    print_numbers(5);
    return 0;
}
```

print_numbers(5)
→ print_numbers(4)
→ print_numbers(3)
→ print_numbers(2)
→ print_numbers(1)
→ print_numbers(0)
→ return
← prints **1**
← prints **2**
← prints **3**
← prints **4**
← prints **5**

Structure of a factorial function

- The recursive algorithm consists of a part that recursively calls itself and a part that stops the recursive call.

  n! = n × (n-1) × (n-2) × ... × 1



Calculating factorial

- Factorial calling order

factorial(3)

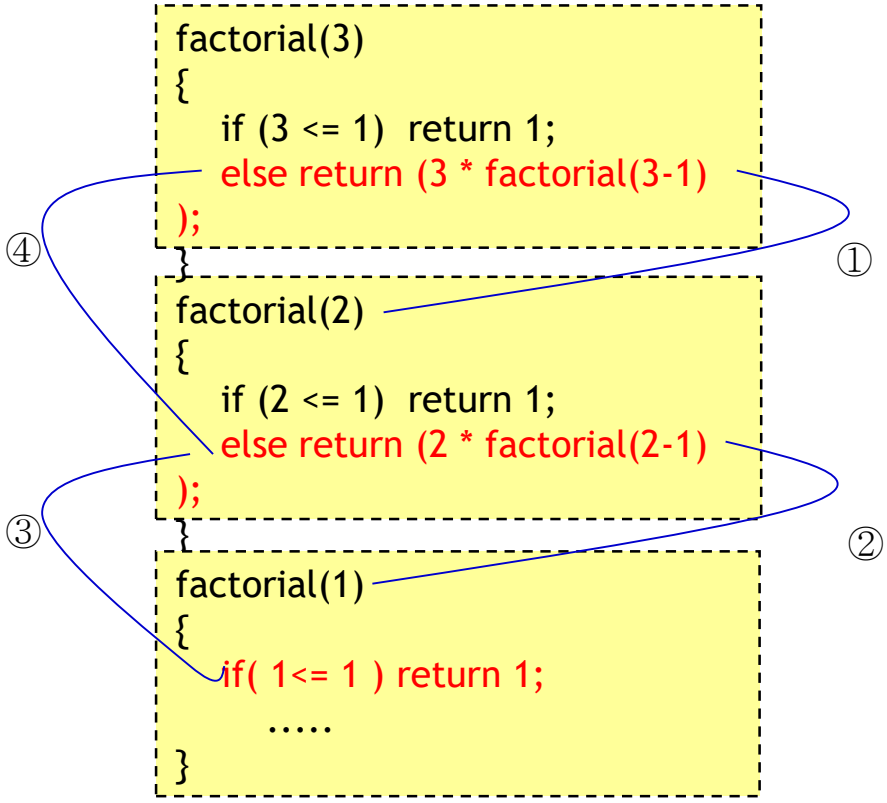
= 3 * factorial(2)

= 3 * 2 * factorial(1)

= 3 * 2 * 1

= 3 * 2

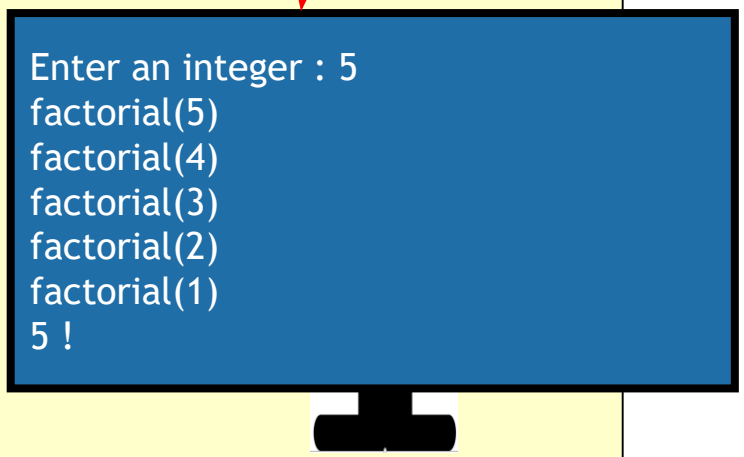
= 6



Factorial calculation

Practice
Ch9-2.c

```
// Calculate the
#include < stdio.h >

long factorial( int n )
{
        printf( "factorial(%d)\n" , n );

        if ( n <= 1)
              return 1;
        else
             return n * factorial( n - 1);
}

int main( void )
{
        int x = 0;
        long f;

        printf ( " Enter an integer :" );
        scanf (“%d”, &x);
        printf (“%d! is %ld . \n”, x, factorial(x));
        return 0;
}
```



Q & A