Ch.9 Functions and Variables

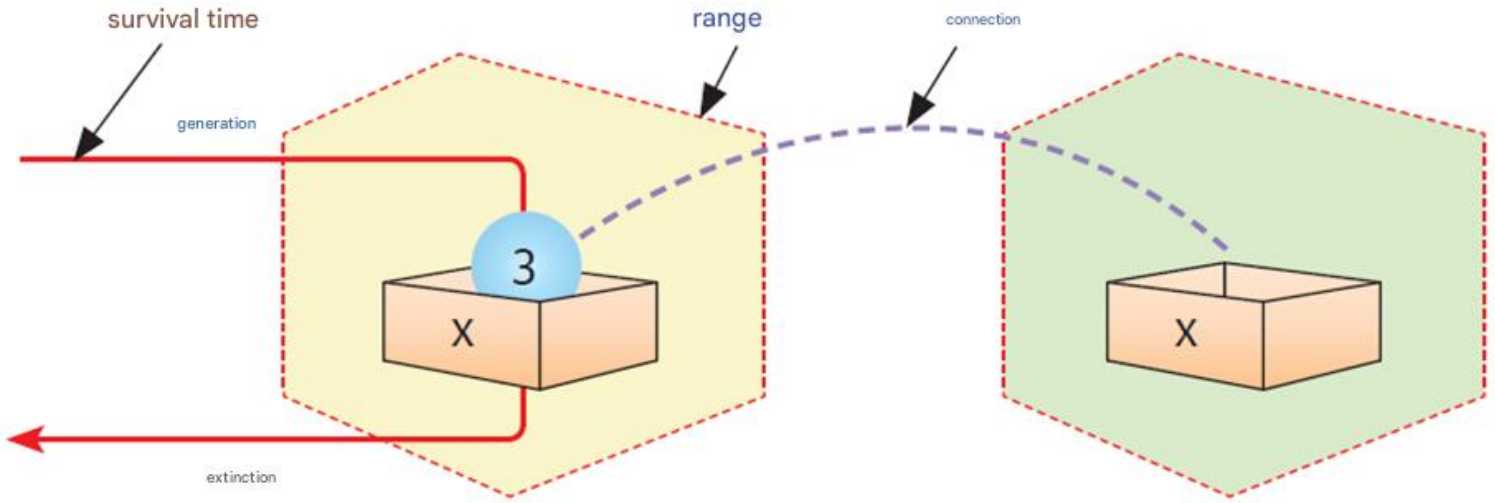
What you will learn in this chapter

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

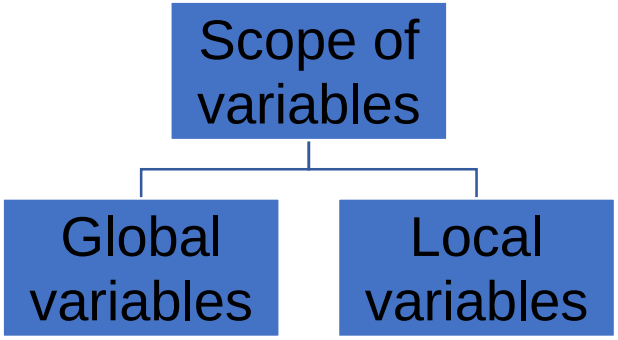
In this chapter, we will focus on the relationship between functions and variables.
We will also look at recursive calls, where a function calls itself .

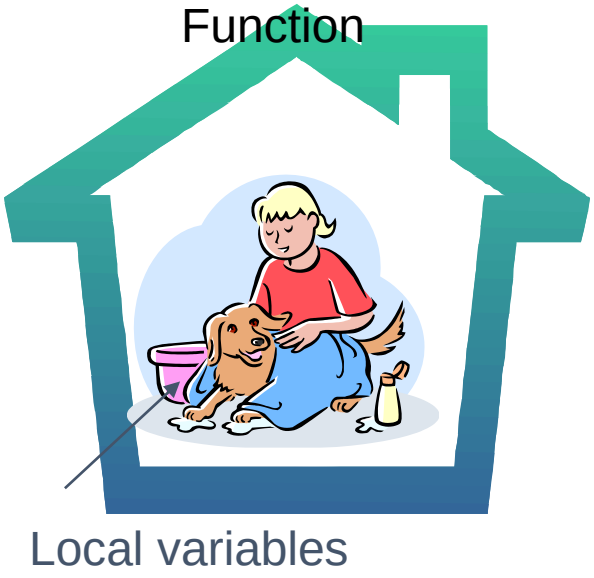
Variable properties

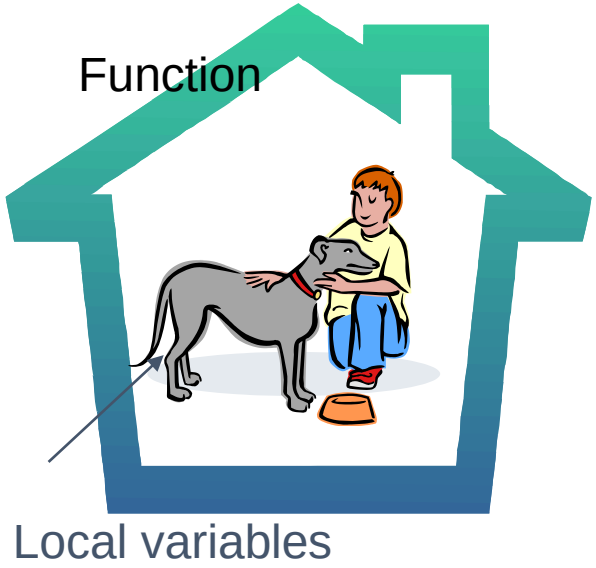
- **Variable properties : name , type , size , value + range , life time , linking**
  - **Scope : The scope** in which a variable is available , its visibility
  - **Lifetime :** The time it exists in memory
  - **Linkage : Status of** connection with variables in other areas



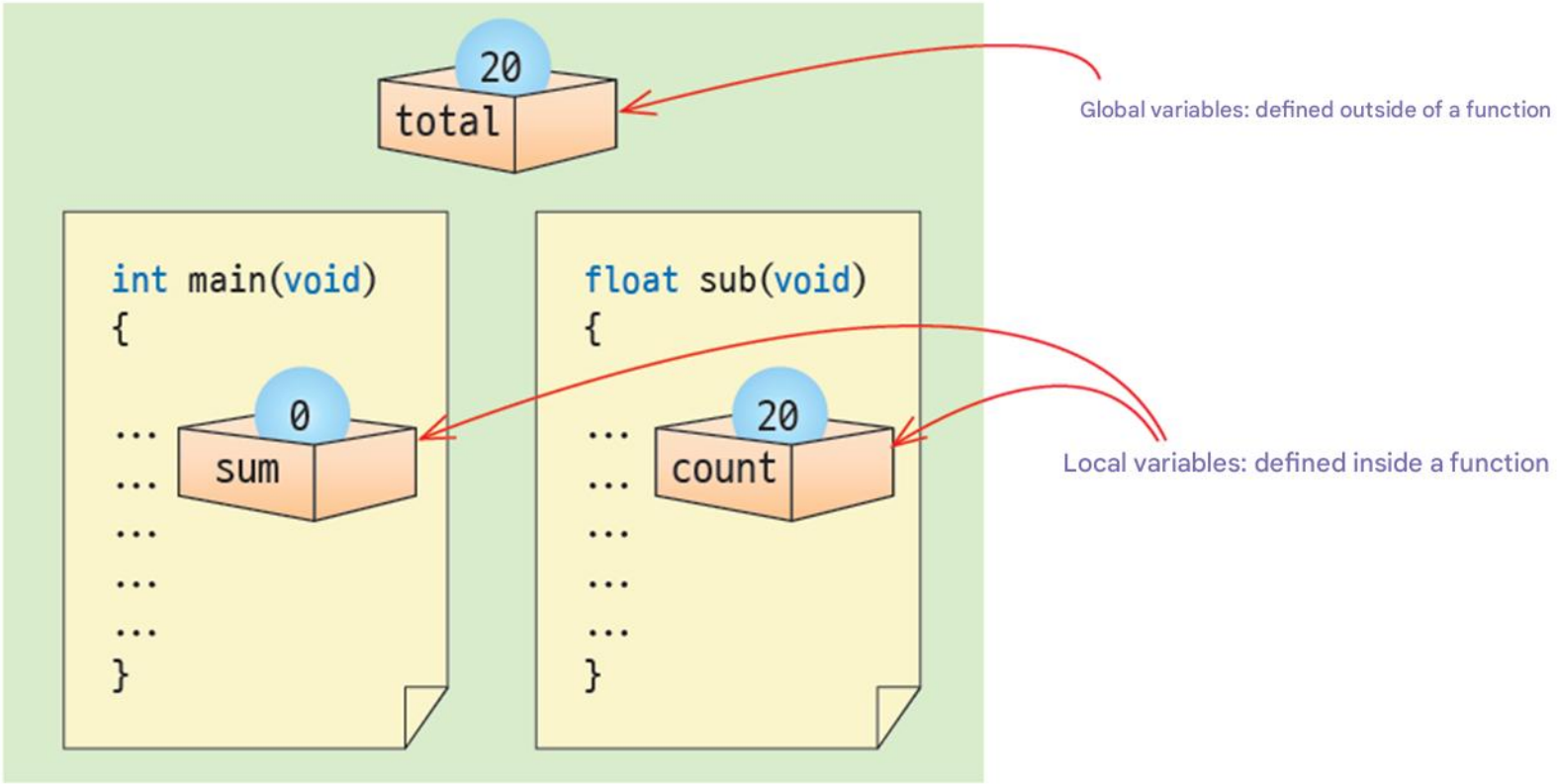
Scope of variables





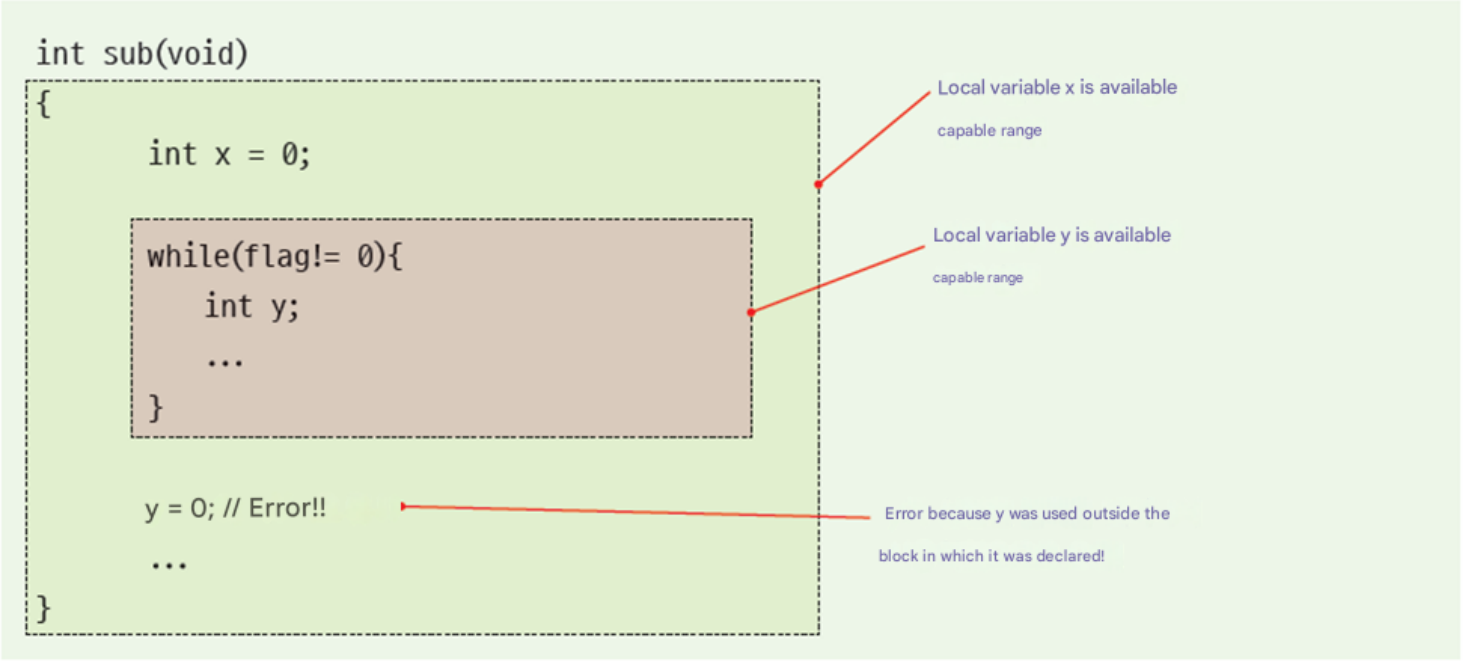


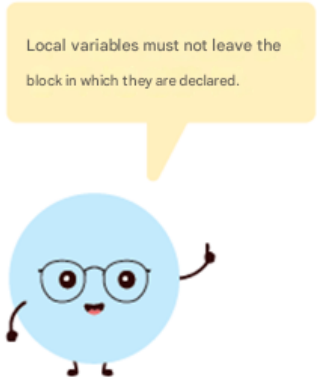
Global Variables and local variables



Local variables

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





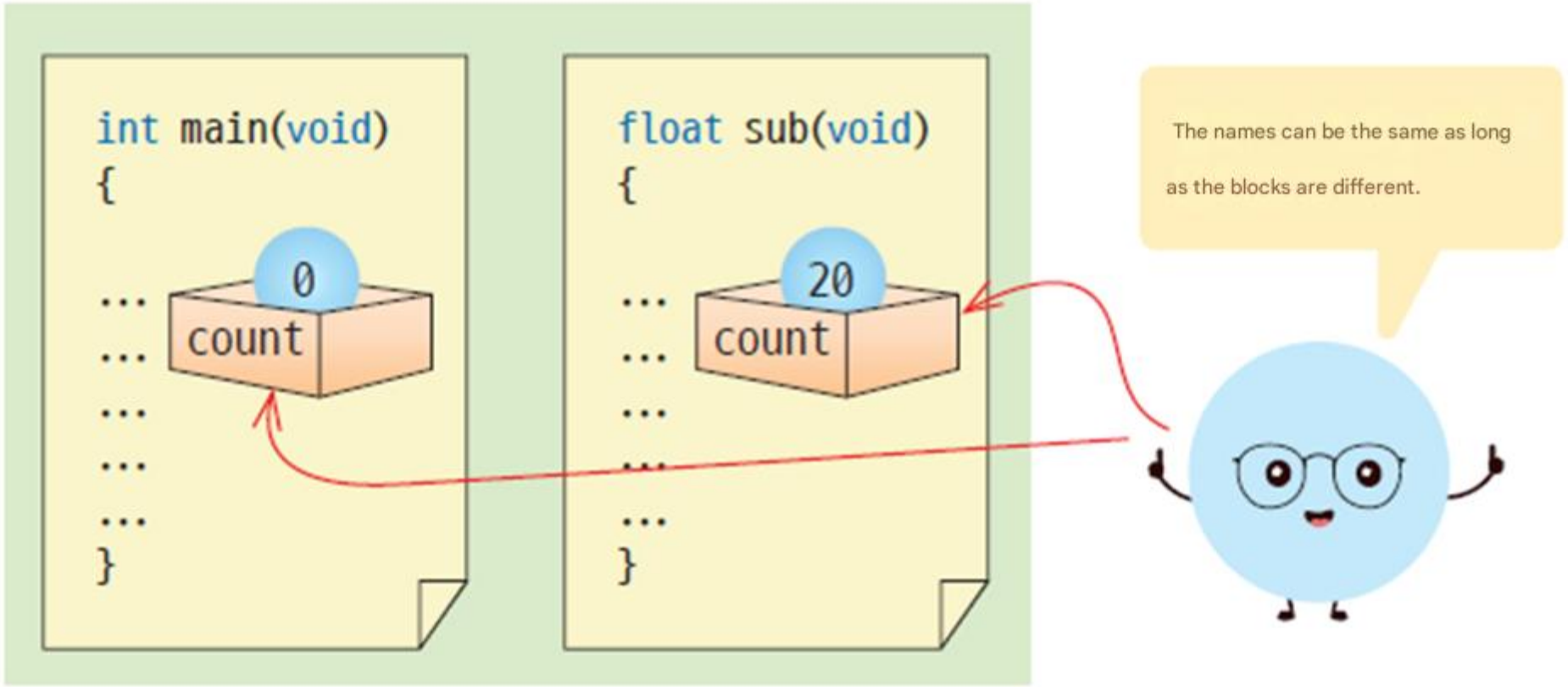
Local variable declaration location

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

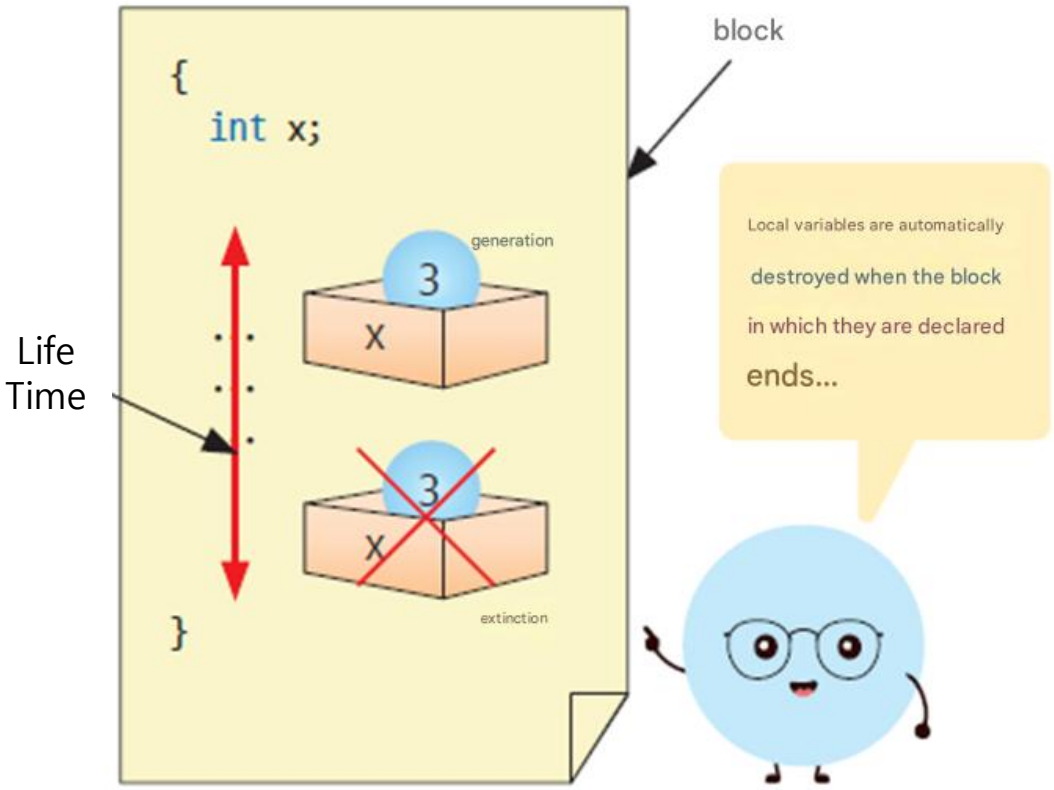
```
while(1) {
        ...
        ...
        int sum = 0;
        ...
}
```

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

Local variables with the same name



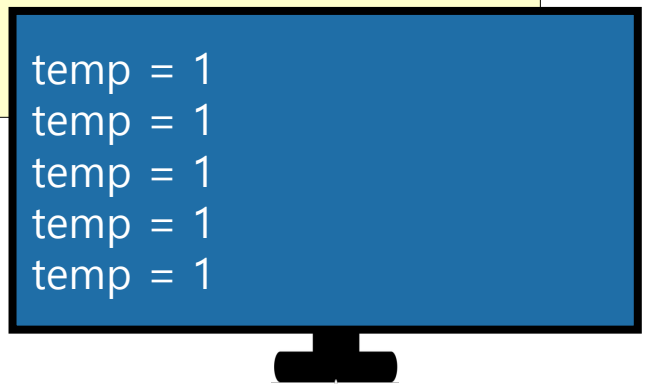
Life time of local variables



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;
}
```

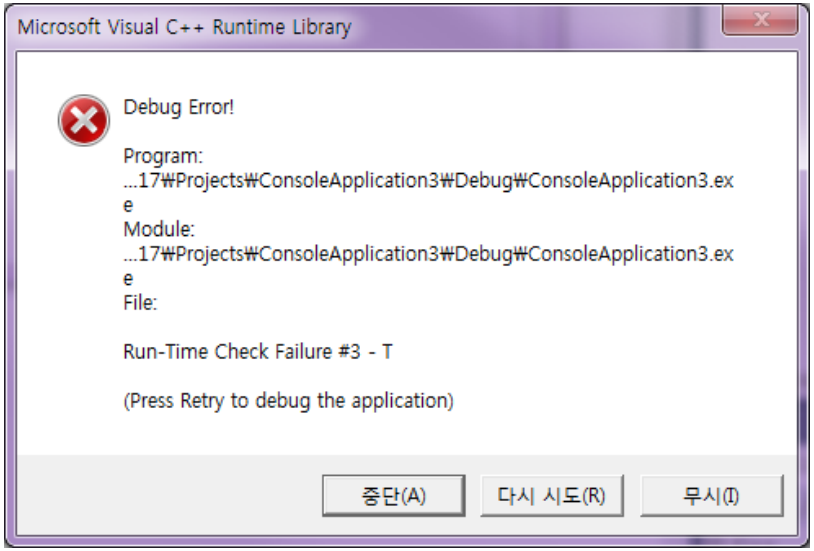
At the start of each block
It is created and initialized



Initial value of local variable

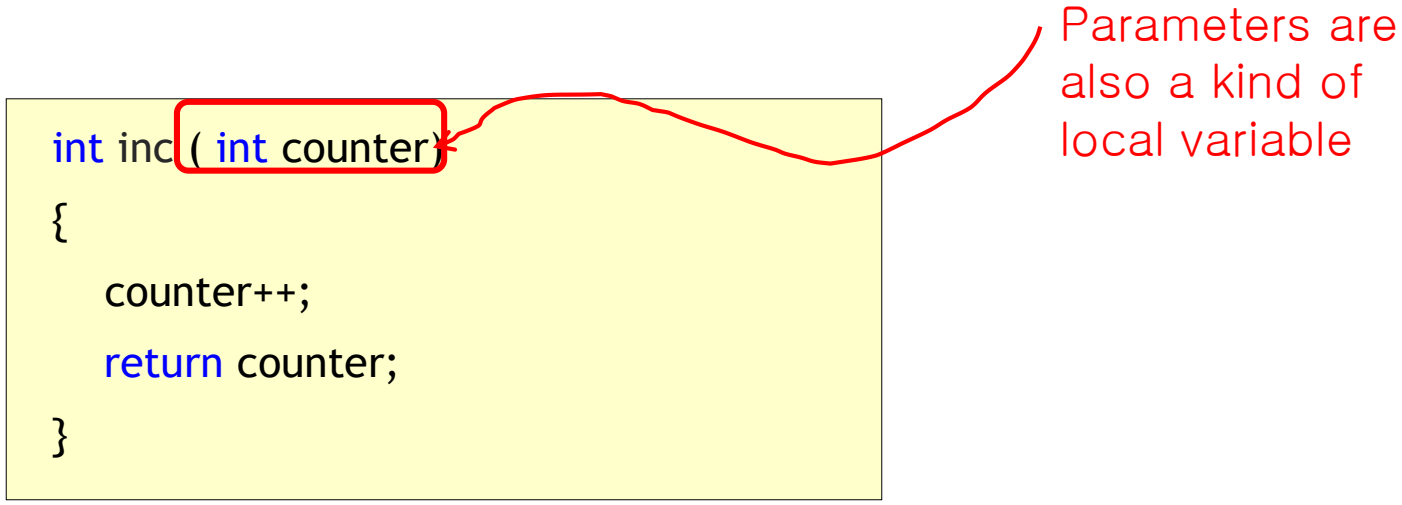
```
#include < stdio.h >
int main( void )
{
        int temp;
         printf ( "temp = %d\n" , temp);
        return 0;
}
```

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 init ialized with the argument values when the function is called .



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++;
}
```

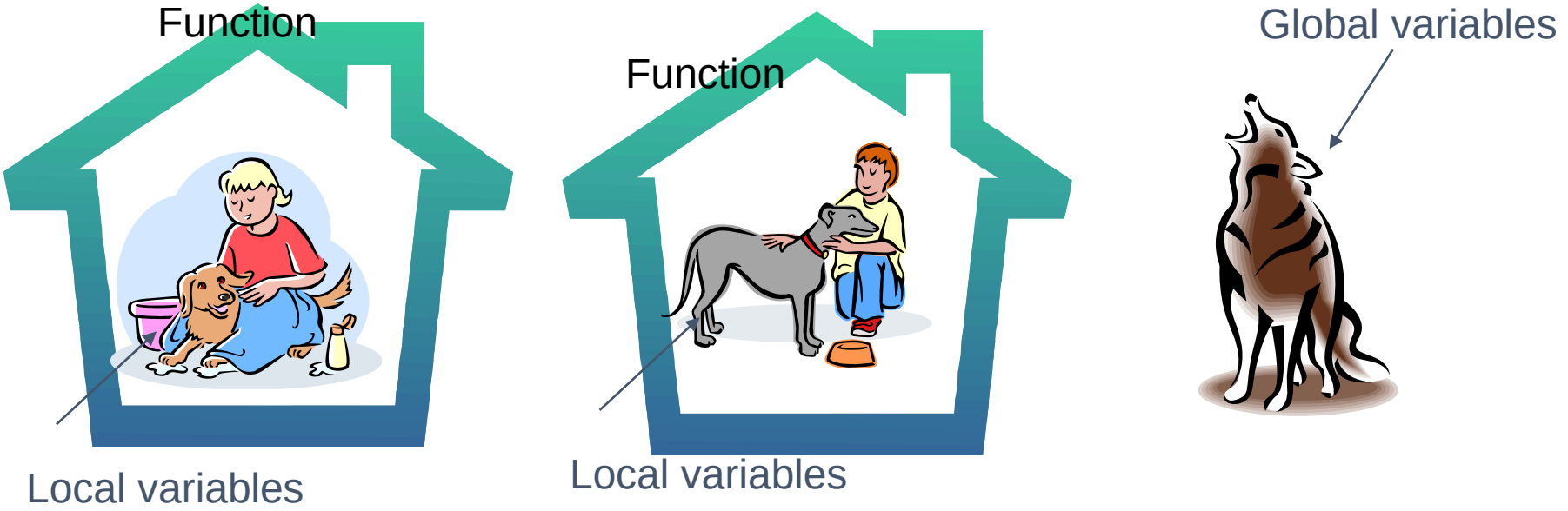
Call by value
(call by value)

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 .

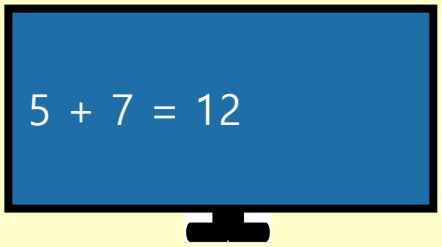


Initial values and life time of global variables

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;
}
```

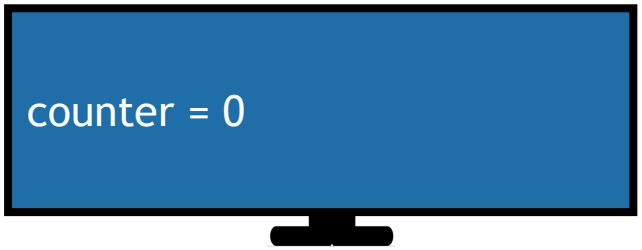
Global variables
The initial value is 0



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 ( "*" );
}
```

Use of global variables

- Common data used in almost all functions is made into global variables.

- Data that is only used by some functions should be passed as function arguments rather than as global variables.

Global and local variables with the same name

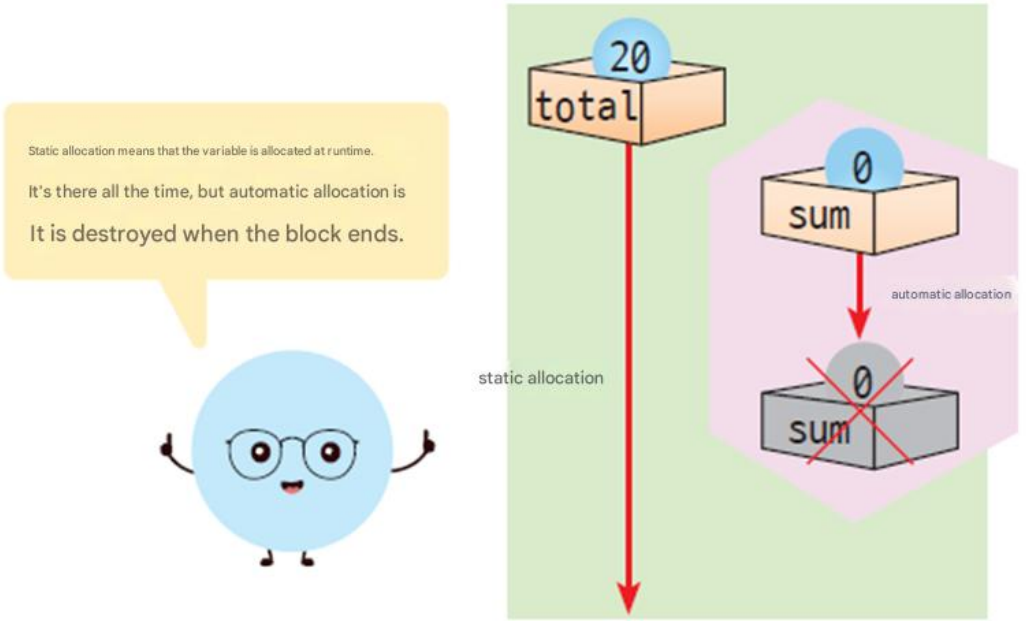
```
#include < stdio.h >

int sum = 1; // global variable

int main( void )
{

        int sum = 0; // local variable

        printf ( "sum = %d\n" , sum);

        return 0;
}
```

Global and local variables are declared with the same name .

Survival period

- Static allocation :
  - Keep it alive while the program runs

- Automatic allocation :
  - Created when entering a block
  - Destroys when exiting the block



Survival period

- Factors that determine survival time
  - Where the variable is declared
  - Storage type specifier

- Storage type specifier
  - auto
  - register
  - static
  - extern

Storage type specifier auto

- Specifies a storage type that is automatically created at the location where the variable is declared, and is automatically destroyed when the block is exited.
- Local variables become automatic variables even if auto is omitted .

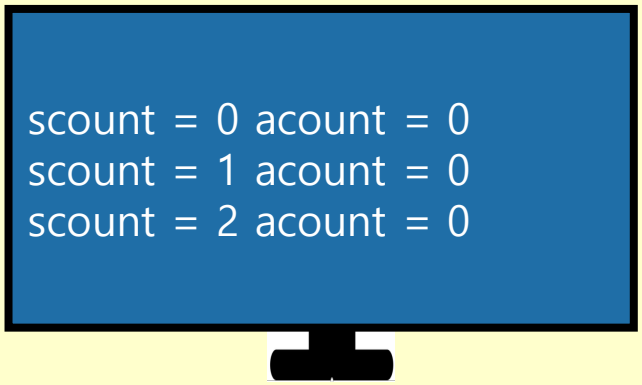
```
int main( void )
{
  auto int sum = 0;
  int i = 0;
…
…
}
```

All of them are automatic variables, created when the function starts and destroyed when it ends .

Storage type specifier “static”

```
#include < stdio.h >

void sub() {

        static int scount = 0;
        int acount = 0;
        printf ( " scount = %d\t" , scount );

        printf ( " acount = %d\n" , acount );
        scount ++;
        acount ++;

}

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

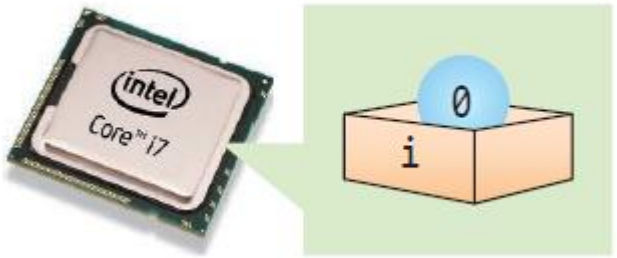
If you add
Local variables become static variables



Storage type specifier “register”

- Store variables in registers .

```
register int i;
for (i = 0;i < 100; i++)
sum += i;
```



Variables are stored in registers inside the CPU

volatile

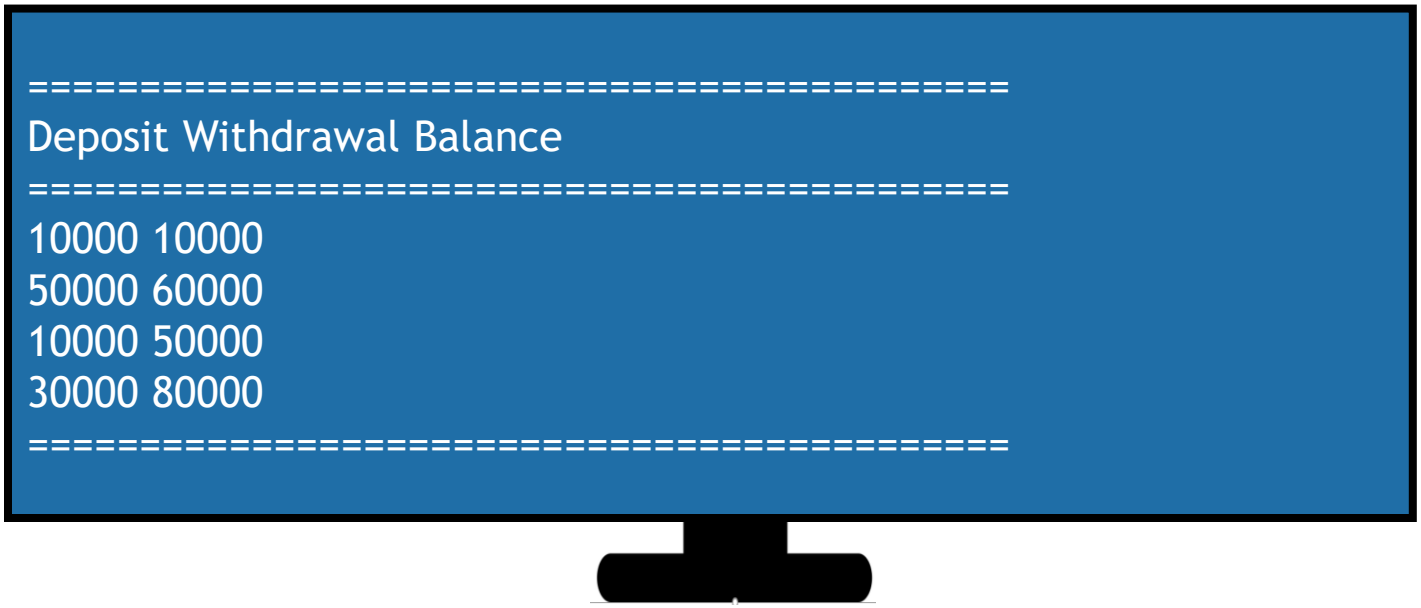
- The volatile specifier is used when the hardware changes the value of a variable from time to time.

```
volatile int io_port ; // Variable connected to hardware

void wait( void ) {
        io_port = 0;
        while ( io_port != 255)
                ;
}
```

If you specify it as volatile , the computer
The filer will stop optimizing .

Lab: Bank Implementing an account

- Let's assume a person who saves money whenever he gets it. Let's write a function save(int amount) for this person . This fun ction takes only one argument, amount, which indicates the amount to save , and is called like save(100) . save() uses a static variable to remember the total amount saved so far, and every time it is called, it prints the total amount saved to the screen like this :



source

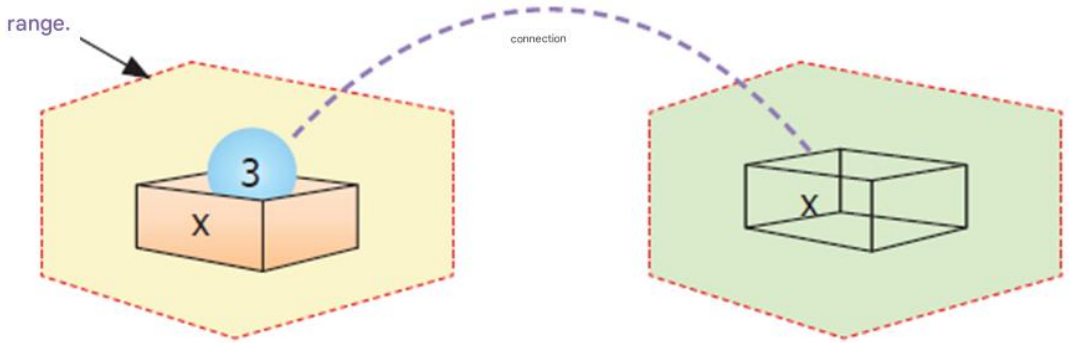
```
#include < stdio.h >

// If the amount is positive, it is considered a deposit, and if it is negative, it is
considered a withdrawal .
void save( int amount )
{
        static long balance = 0;

        if ( amount >= 0)
                printf( "%d \t\t" , amount );
        else
                printf( "\t %d \t" , - amount );

        balance += amount ;
        printf ( "%d \n" , balance);
}
```

source

```
int main( void ) {
        printf ( "============================\n" );
        printf ( " Deposit \t Withdrawal \t Balance \n" );
        printf ( "============================\n" );
        save(10000);
        save(50000);
        save(-10000);
        save(30000);
        printf ( "============================\n" );
        return 0;
}
```

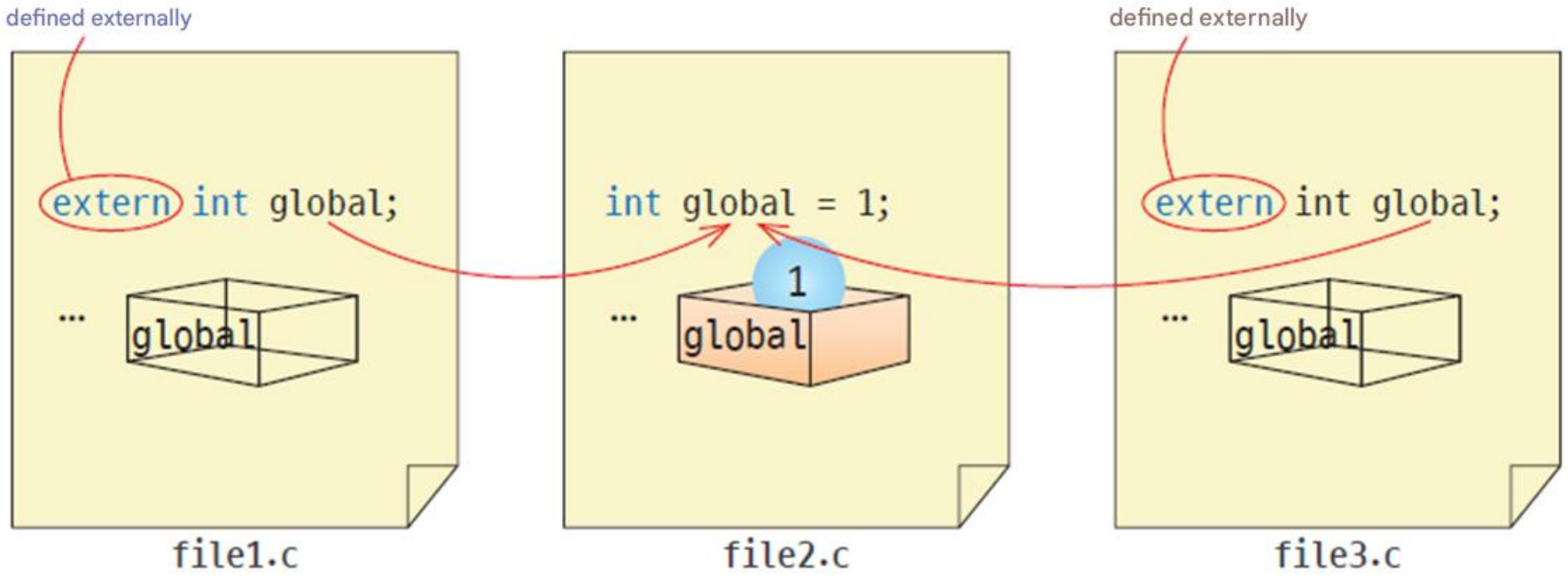
connection

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

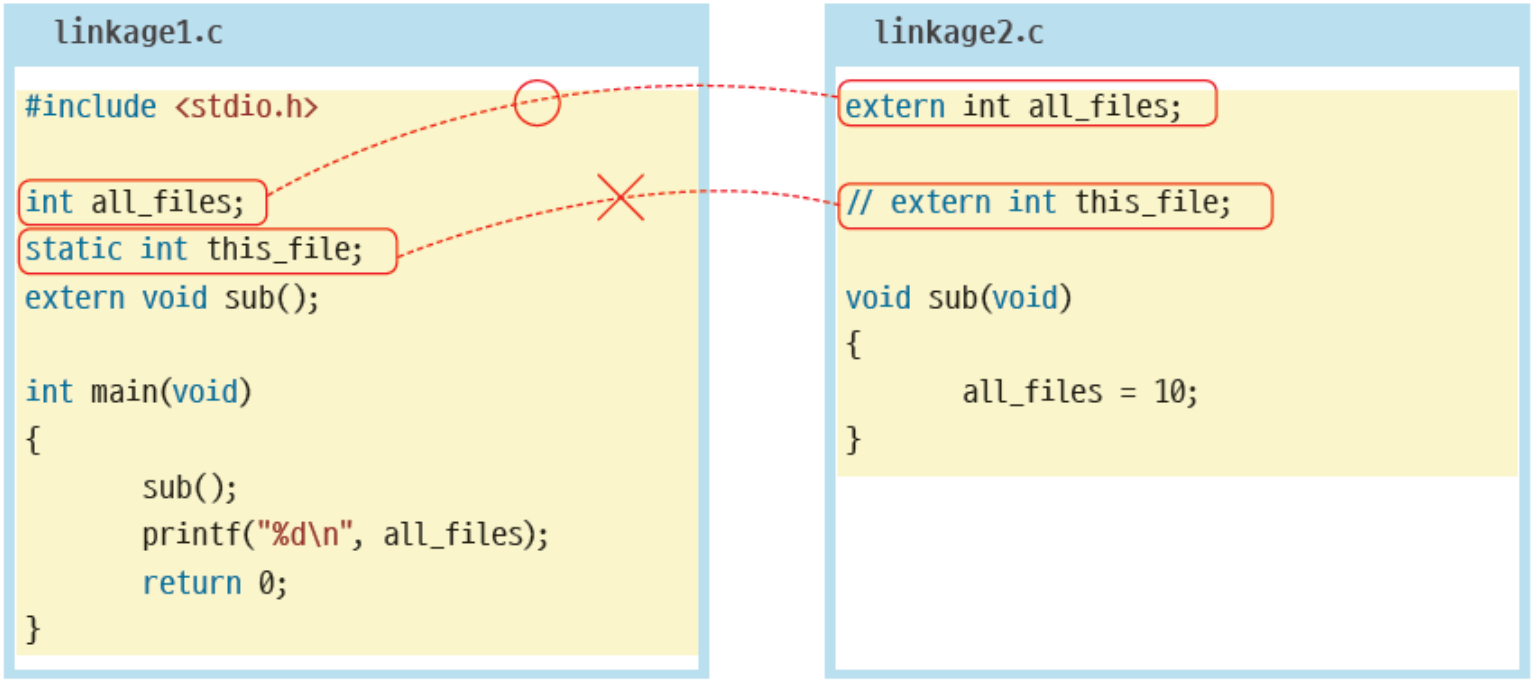


External connection

- 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");
}
```

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;
}
```

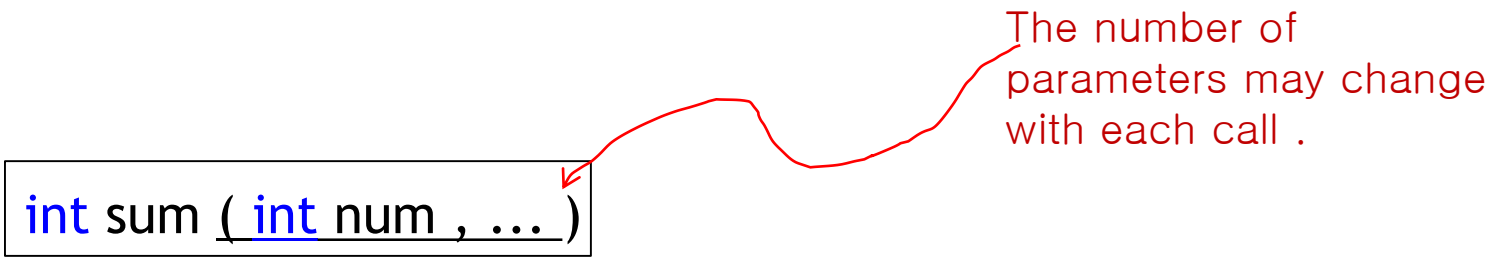
What storage type do you use ?

- In general, it is recommended to use *the auto-save type.*
- If the value of a variable needs to remain the same even after t he function call ends, use *local static*
- If it is a variable that needs to be shared among many functions , *it is an external reference variable.*

storage type	keyword	position to be defined	range	survival time
automatic	auto	Inside the function	region	temporary
register	register	Inside the function	region	temporary
static area	staic	Inside the function	region	everlasting
Global	doesn't exist	outside the function	all source files	everlasting
static global	static	outside the function	one source file	everlasting
external reference	extern	outside the function	all source files	everlasting

Variable parameters

- A feature where the number of parameters can vary variably.



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 -- )
                answer += va_arg ( argptr , int );
        va_end ( argptr );
        return ( answer );
}
```

Number of parameters

What is recursion ?

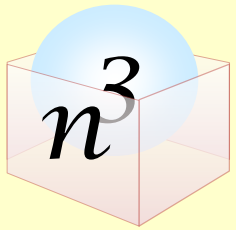
- A function can also call itself . This is called recursion .

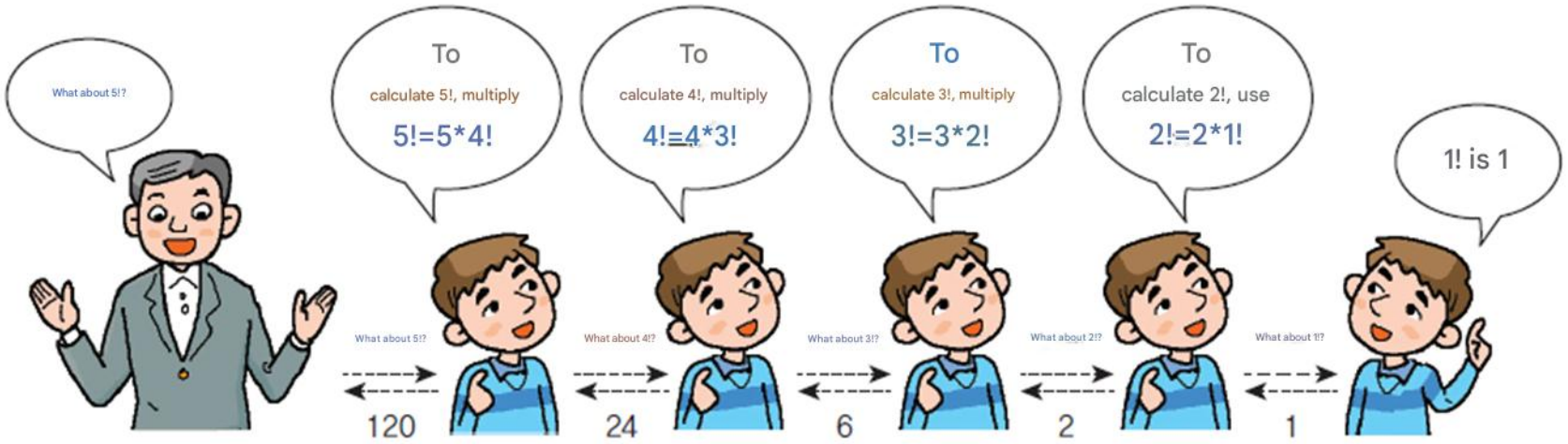
$$n! = \begin{cases} 1 & n=0 \\ n^*(n-1)! & n \geq 1 \end{cases}$$

Calculating factorial

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

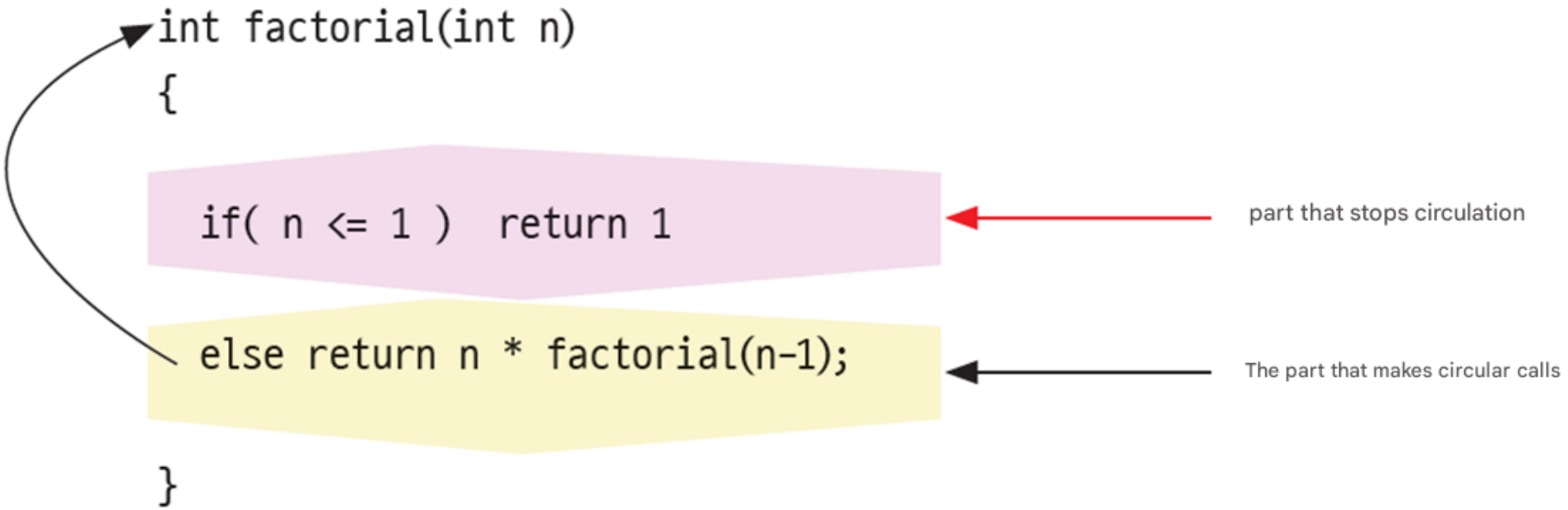
```
int factorial( int n)
{
    if ( n <= 1 ) return (1);
    else return (n * factorial(n-1) );
}
```





Structure of a cyclic function

- The recursive algorithm consists of a part that recursively calls itself and a part that stops the recursive call, as shown in Figure 9-9 .



Calculating factorial

- Factorial calling order

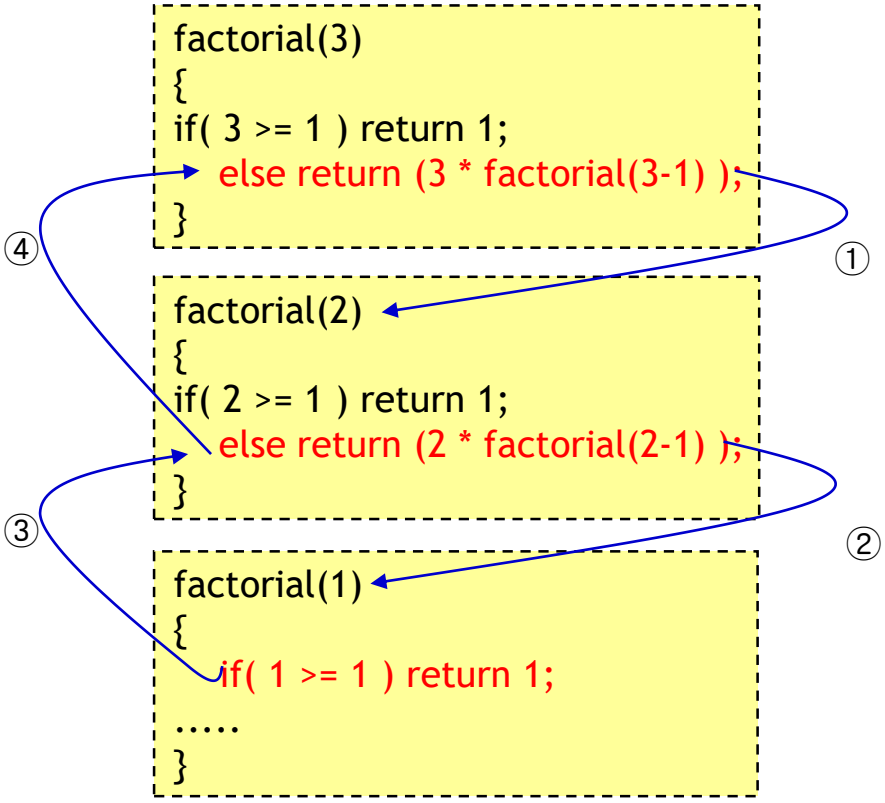
factorial(3) = 3 * factorial(2)

= 3 * 2 * factorial(1)

= 3 * 2 * 1

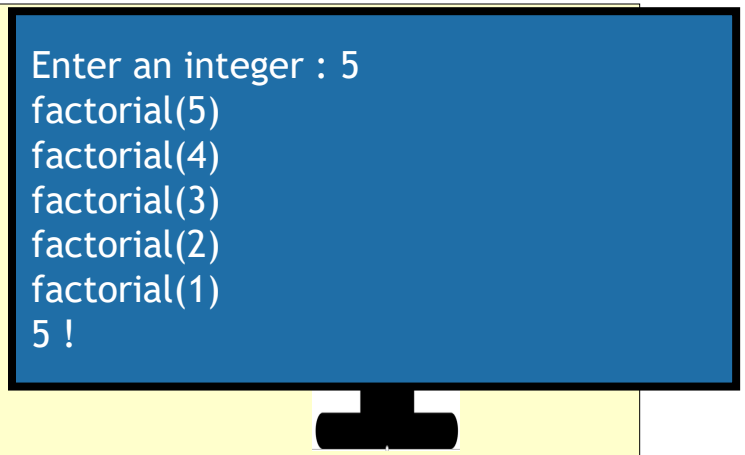
= 3 * 2

= 6



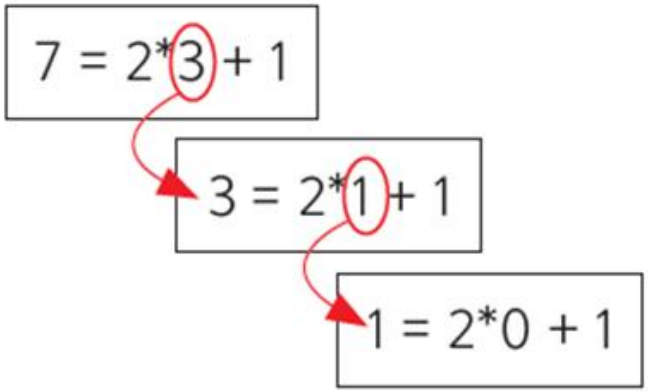
Factorial calculation

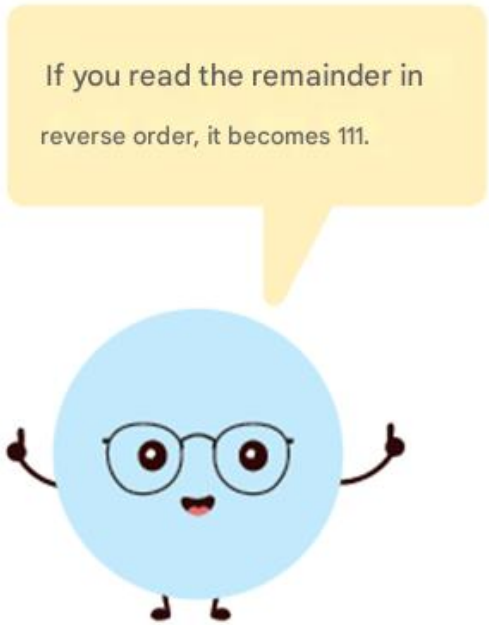
```
// Calculate the
#include < stdio.h >

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”, &n);
        printf (“%d! is %d . \n”, n, factorial(n));
        return 0;
}
```



Output in binary format

- C does not have a function to output an integer as a binary number . Let's implement this function using a circular call .





Output in binary format

```
// Output in binary format
#include < stdio.h >

void print_binary ( int x );

int main( void )
{
        print_binary (9);
        printf ("\n");
        return 0;
}

void print_binary ( int x )
{
        if ( x > 0)
        {
                print_binary ( x / 2); // recursive call
                printf ( "%d" , x % 2); // Print the remainder
        }
}
```

Q & A