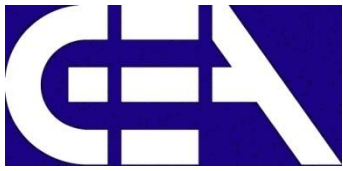




Computer Programming

Sino-European Institute of Aviation Engineering



Module 4 *Function*

Outline

- Introduction**
- Function Definitions**
- Function Calls**
- Recursion**
- Storage Class**
- Local Variable and External Variable**
- Scope Rule**
- Summary**

Introduction

- ❑ A function is **a series of statements** that have been grouped together and given a name.
- ❑ Each function is essentially a small program, with its own declarations and statements.
- ❑ Advantages of functions:
 - A program can be divided into small pieces that are easier to understand and modify.
 - We can avoid duplicating code that's used more than once.
 - A function that was originally part of one program can be reused in other programs.

Introduction

□ Why use functions

- Writing functions avoids rewriting the same code over and over.
 - ◆ printf and scanf are good examples...
- Break your problem down into smaller sub-tasks
 - ◆ easier to solve complex problems
- They make a program much easier to read and maintain.

Introduction

□ Functions

- Modules in C
- Programs combine user-defined functions with library functions
 - ◆ C standard library has a wide variety of functions

□ Function calls

- Invoking functions
 - ◆ Provide function name and arguments (data)
 - ◆ Function performs operations or manipulations
 - ◆ Function returns results
- Function call analogy:
 - ◆ Boss asks worker to complete task
 - ◆ Worker gets information, does task, returns result
 - ◆ Information hiding: boss does not know details

Introduction

```
double square(double a)
{
    return a * a;
}
```

**This is a function defined
outside main**

```
int main(void)
{
    double num = 0.0, sqr = 0.0;
```

```
    printf("enter a number\n");
    scanf("%lf",&num);
```

```
    sqr = square(num);
```



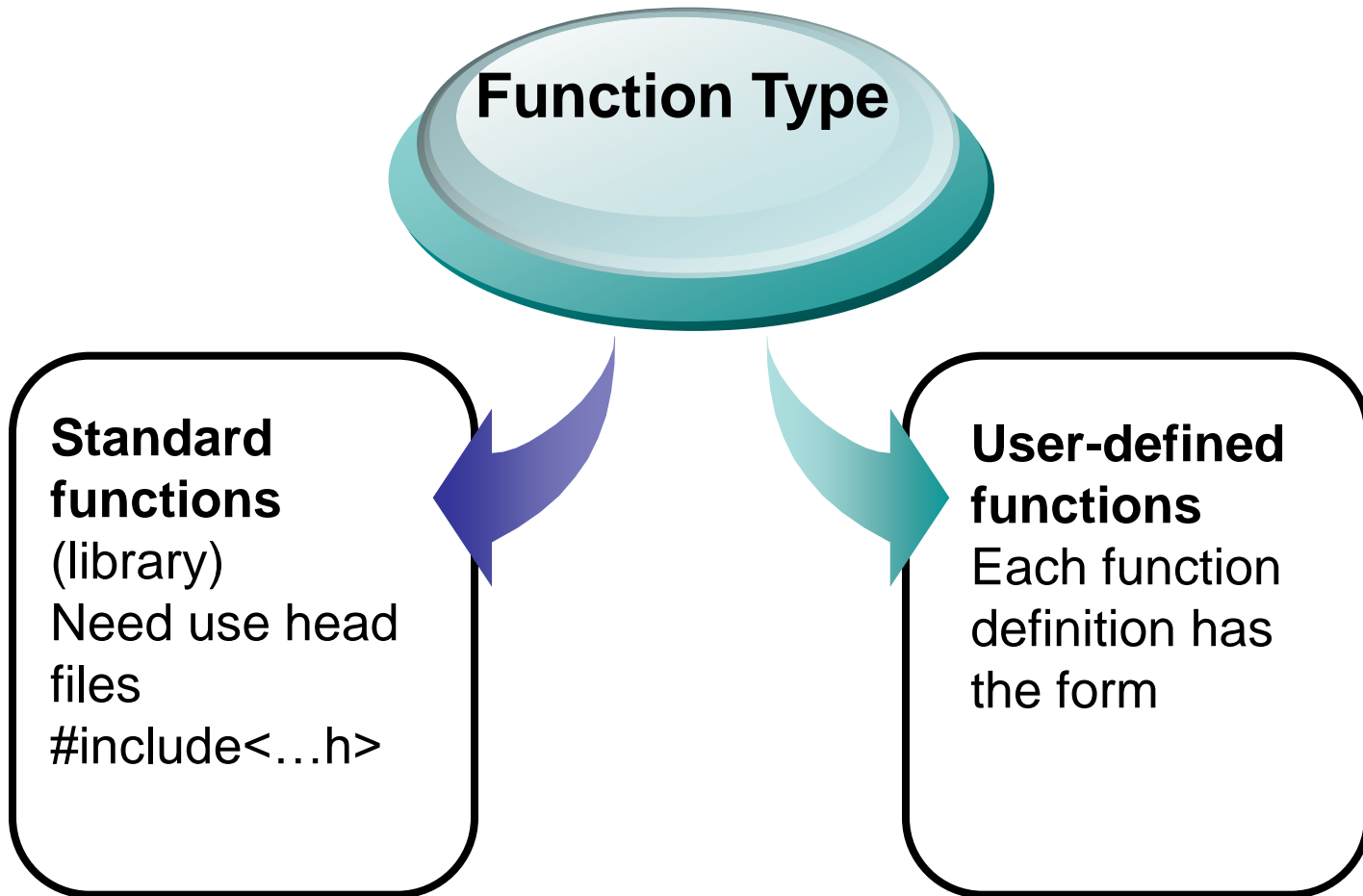
**Here is where we call
the function square**

```
    printf("square of %g is %g\n", num, sqr);
```

```
    return 0;
```

```
}
```

Introduction



Introduction

□ Math functions : `#include <math.h>`

- `sin(x)`, `cos(x)`, `tan(x)`
- `asin(x)`, `acos(x)`, `atan(x)`
 - ◆ `x` is given in radians
- `log(x)`
- `sqrt(x)`
- `pow(x,y)` – raise `x` to the `y`th power.
- `ceil(x)`, `floor(x)` ...and more

Function Definitions

□ General form of a *function definition*:

return-type function-name (parameter list)

```
{  
    declarations  
    statements  
    return return-value;  
}
```

average function:

```
double average(double a, double b)  
{  
    double sum;    /* declaration */  
  
    sum = a + b;    /* statement */  
    return sum / 2; /* statement */  
}
```

- Functions can not be defined inside other functions.
- Nesting function doesn't means nesting definition.

Function Definitions

- ❑ The **return type** of a function is the type of value that the function returns (default int).
- ❑ Rules governing the return type:
 - Functions may not return arrays.
 - Specifying that the return type is **void** indicates that the function doesn't return a value.

Function Definitions

- ❑ After the function name comes a list of parameters (**formal arguments**).
- ❑ Each parameter is preceded by a specification of its type; parameters are separated by commas.
- ❑ If the function has no parameters, the word **void** should appear between the parentheses.

Function Definitions

- ❑ The **body** of a function may include both declarations and statements.
- ❑ Variables declared in the body of a function can't be examined or modified by other functions.
- ❑ Variable declarations and statements can be mixed, as long as each variable is declared prior to the first statement that uses the variable.

Function Definitions

- ❑ The body of a function whose return type is **void** (a “**void** function”) can be empty:

```
void print_pun(void)  
{  
}
```

- ❑ Leaving the body empty may make sense as a temporary step during program development.

Function Definitions

□ Prototype

- Function name
- Parameters – what the function takes in
- Return type – data type function returns (default int)
- Used to validate functions
- Prototype only needed if function definition comes after use in program
- The function with the prototype
 - ◆ `int maximum(int, int, int);`
 - ◆ Takes in 3 ints / Returns an int

□ Promotion rules and conversions

- Converting to lower types can lead to errors

Function Calls

□ Call by value

- Copy of argument passed to function
- Changes in function do not effect original
- Use when function does not need to modify argument
 - ◆ Avoids accidental changes

□ Call by reference

- Passes original argument
- Changes in function effect original
- Only used with trusted functions

Function Calls

- A function call consists of a function name followed by a list of arguments (**actual arguments**), enclosed in parentheses:

average(x, y)

print_count(i)

print_pun()

- If the parentheses are missing, the function won't be called:

*print_pun; /** WRONG **/*

This statement is legal but has no effect.

Function Calls

- A call of a *void* function is always followed by a semicolon to turn it into a statement:

```
print_count(i);  
print_pun();
```

- A call of a *non-void* function produces a value that can be stored in a variable, tested, printed, or used in some other way:

```
avg = average(x, y);  
if (average(x, y) > 0)  
    printf("Average is positive\n");  
printf("The average is %g\n", average(x, y));
```

Function Calls

- ❑ The value returned by a non-void function can always be discarded if it's not needed:

average(x, y); / discards return value */*

- ❑ Ignoring the return value of `average` is an odd thing to do, but for some functions it makes sense.
- ❑ `printf` returns the number of characters that it prints.
- ❑ After the following call, `num_chars` will have the value 9:

num_chars = printf("Hi, Mom!\n");

- ❑ We'll normally discard `printf`'s return value:

printf("Hi, Mom!\n"); / discards return value */*

Program: Computing Average

- ❑ A function named *average* that computes the average of two *double* values:

```
double average(double a, double b)  
{  
    return (a + b) / 2;  
}
```

- ❑ The word *double* at the beginning is the ***return type*** of *average*.
- ❑ The identifiers *a* and *b* (the function's ***parameters***) represent the numbers that will be supplied when *average* is called.

Program: Computing Average

- ❑ Every function has an executable part, called the *body*, which is enclosed in braces.
- ❑ The body of *average* consists of a single *return* statement.
- ❑ Executing this statement causes the function to “return” to the place from which it was called; the value of $(a + b) / 2$ will be the value returned by the function.

Program: Computing Average

- A *function* call consists of a function name followed by a list of ***arguments***.
 - *average(x, y)* is a call of the *average* function.
- Arguments are used to supply information to a function.
 - The call *average(x, y)* causes the values of *x* and *y* to be copied into the parameters *a* and *b*.
- An argument doesn't have to be a variable; any expression of a compatible type will do.
 - *average(5.1, 8.9)* and *average(x/2, y/3)* are legal.

Program: Computing Average

- ❑ We'll put the call of *average* in the place where we need to use the return value.
- ❑ A statement that prints the average of *x* and *y*:
printf("Average: %g\n", average(x, y));
The return value of *average* isn't saved; the program prints it and then discards it.
- ❑ If we had needed the return value later in the program, we could have captured it in a variable:
avg = average(x, y);

Program: Computing Average

- ❑ The program reads three numbers and uses the *average* function to compute their averages, one pair at a time:

Enter three numbers: 3.5 9.6 10.2

Average of 3.5 and 9.6: 6.55

Average of 9.6 and 10.2: 9.9

Average of 3.5 and 10.2: 6.85

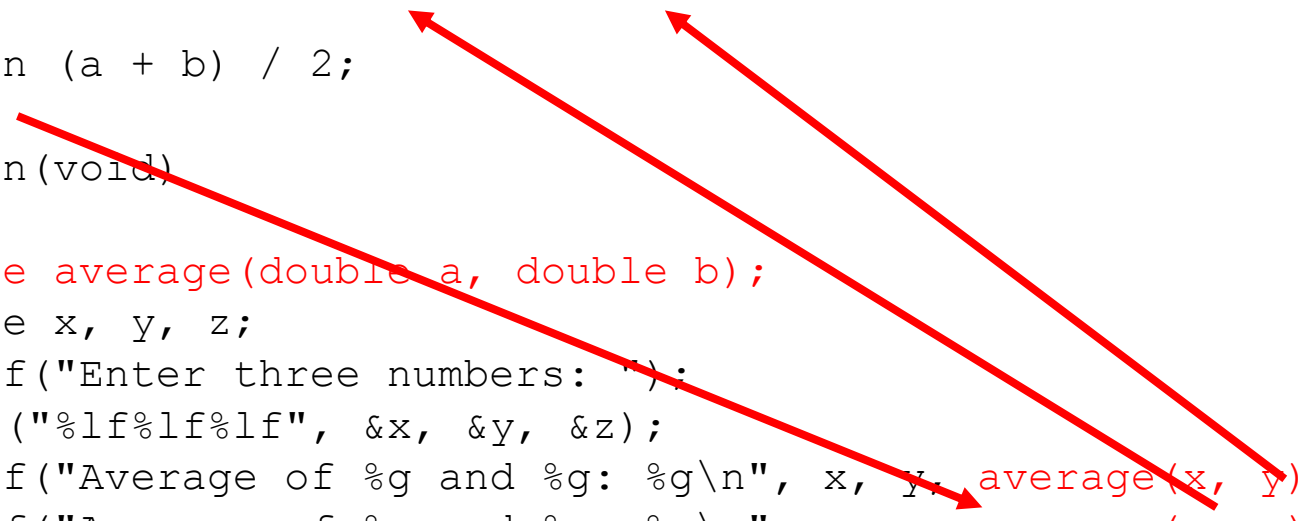
Program: Computing Average

```
/* Computes pairwise averages of three numbers */

#include <stdio.h>

double average(double a, double b)
{
    return (a + b) / 2;
}

int main(void)
{
    double average(double a, double b);
    double x, y, z;
    printf("Enter three numbers: ");
    scanf("%lf%lf%lf", &x, &y, &z);
    printf("Average of %g and %g: %g\n", x, y, average(x, y));
    printf("Average of %g and %g: %g\n", y, z, average(y, z));
    printf("Average of %g and %g: %g\n", x, z, average(x, z));
    return 0;
}
```



The diagram illustrates the flow of function calls. Three red arrows originate from the `average` function calls in the `main` function and point to the `average` function definition. The first arrow starts at `average(x, y)` and points to the `double a` parameter. The second arrow starts at `average(y, z)` and points to the `double b` parameter. The third arrow starts at `average(x, z)` and points to the `double a` parameter. This indicates that the `average` function is called multiple times with different arguments.

Program: Conjecture of Twin Primes

- “Twin Primes” is a pair of prime numbers whose difference is 2.
 - e.g. (3, 5), (11, 13)
 - Write a program that receives as input a number from the user and outputs all the twins up to (and including) that number.
 - For example:
 - ◆ Input: 7
 - ◆ Output: (3, 5), (5, 7)

Program: Conjecture of Twin Primes

□ How To Approach The problem

- Make sure you understand the problem
- Outline the most simple algorithm for solution
- What are the subtasks of the solution?
- each subtask will be a separate function


Program: Conjecture of Twin Primes

```
#include <stdio.h>
int is_prime(int n);

int main(void)
{ int N, i;
  printf("Enter a positive integer: ");
  scanf("%d", &N);
  for (i = 3; i < N - 1; ++i)
  { if (is_prime(i) && is_prime(i + 2)) {
      printf("(%d, %d)\n", i, i + 2);
    }
  }
  return 0;
}

int is_prime(int n)
{ int i = 0;
  for (i = 2; i < n; ++i) /* Check if any of the numbers 2, ... , n-1 divide it. */
  { if (n % i == 0)      /* if this is true, n is not prime */
    { return 0; }
  }
  return 1;              /* if we got here - n is necessarily prime */
}
```

evaluated only if is_prime(i) is true



Recursion

- ❑ A function is ***recursive*** if it calls itself.
- ❑ The following function computes $n!$ recursively, using the formula $n! = n \times (n - 1)!$:

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

Recursion

□ To see how recursion works, let's trace the execution of the statement

i = fact(3);

fact(3) finds that 3 is not less than or equal to 1, so it calls *fact(2)*, which finds that 2 is not less than or equal to 1, so it calls

fact(1), which finds that 1 is less than or equal to 1, so it returns 1, causing

fact(2) to return $2 \times 1 = 2$, causing

fact(3) to return $3 \times 2 = 6$.

Recursion

- The following recursive function computes x^n , using the formula $x^n = x \times x^{n-1}$.

```
int power(int x, int n)  
{  
    if (n == 0)  
        return 1;  
    else  
        return x * power(x, n - 1);  
}
```

Recursion

- ❑ We can condense the *power* function by putting a conditional expression in the *return* statement:

```
int power(int x, int n)
{
    return n == 0 ? 1 : x * power(x, n - 1);
}
```

- ❑ Both *fact* and *power* are careful to test a “termination condition” as soon as they’re called.
- ❑ All recursive functions need some kind of termination condition in order to prevent infinite recursion.

Recursion

□ Example: factorials

- $5! = 5 * 4 * 3 * 2 * 1$

- Notice that

- ◆ $5! = 5 * 4!$

- ◆ $4! = 4 * 3! \dots$

- Can compute factorials recursively

- Solve base case ($1! = 0! = 1$) then plug in

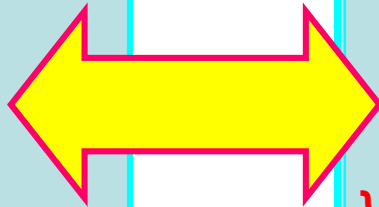
- ◆ $2! = 2 * 1! = 2 * 1 = 2;$

- ◆ $3! = 3 * 2! = 3 * 2 = 6;$

Recursion

Recursion formula
when $n = 1$, $n! = 1$
when $n > 1$, $n! = n * (n-1)!$

```
facto (int n )  
{ int s;  
  if ( n == 1 )  
    s = 1;  
  else  
    s = n * facto(n-1);  
  return (s);  
}
```



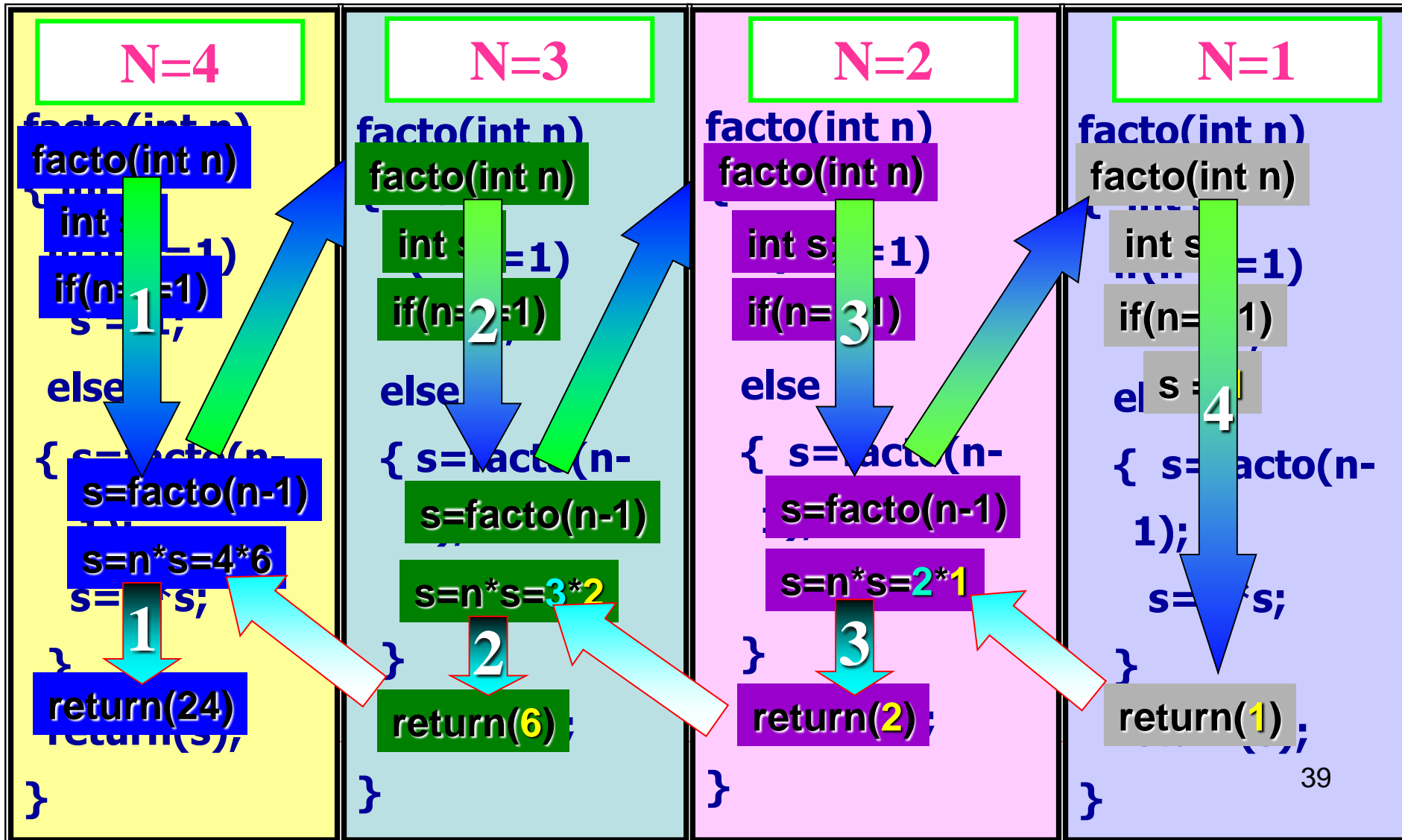
```
facto ( int n )  
{ int s;  
  if ( n == 1 )  
    s = 1;  
  else  
  { s = facto (n-1);  
    s = n*s;  
  }  
  return (s);  
}
```

Recursion

```
#include <stdio.h>
main ( )
{ int n, p;
  printf ("N=?");   scanf ("%d", &n);
  p = facto (n);    printf ("%d!=%d\n", n, p);
}
```

```
facto ( int n )
{ int r;
  if ( n == 1 )
    r = 1;
  else
    r = n * facto(n-1);  /* recursion call */
  return (r);
}
```

Recursion



Recursion

□ Fibonacci series: 0, 1, 1, 2, 3, 5, 8...

- Each number is the sum of the previous two

- Can be solved recursively:

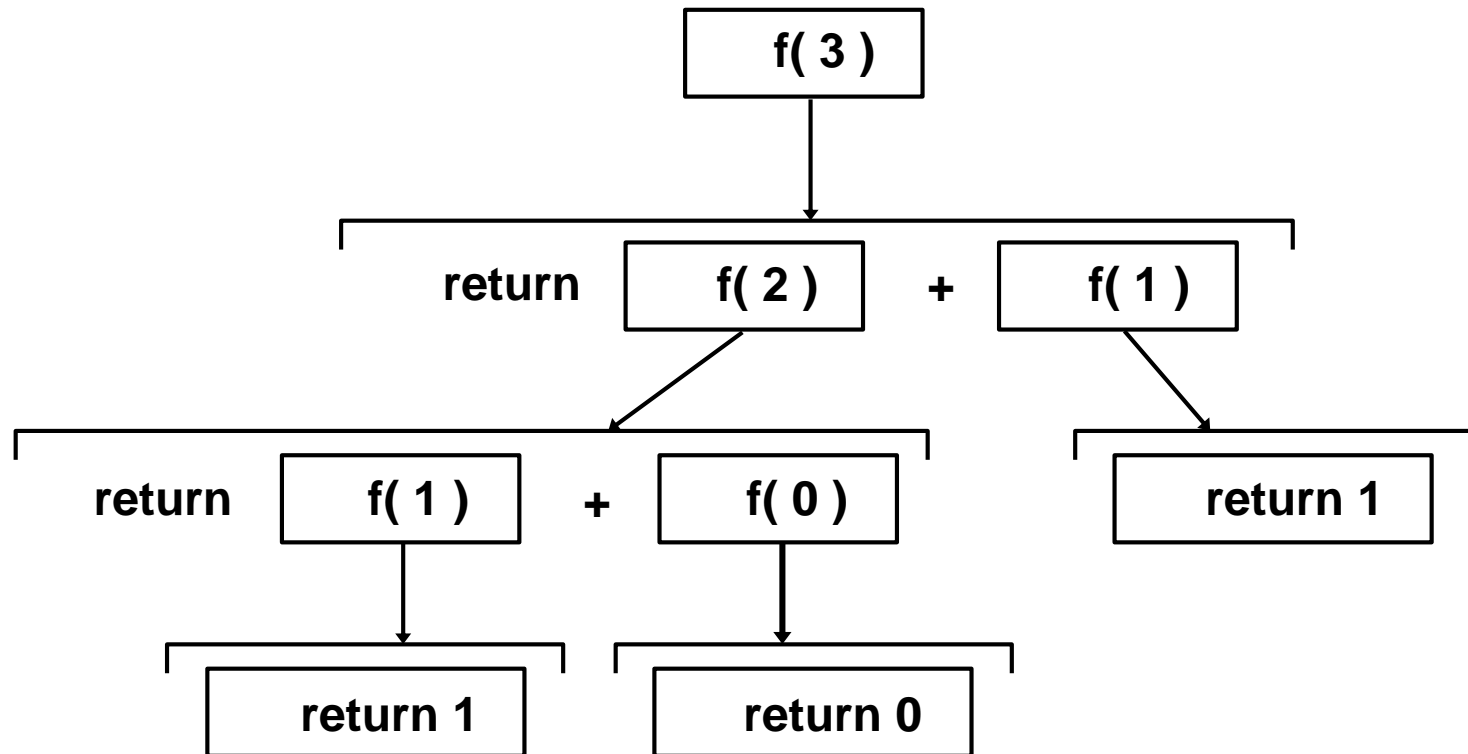
- ◆ $\text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2)$

- Code for the fibaonacci function

```
long fibonacci( long n )
{
    if (n == 0 || n == 1) // base case
        return n;
    else
        return fibonacci( n - 1) +
            fibonacci( n - 2 );
}
```

Recursion

□ Set of recursive calls to function fibonacci



```

1
2  /* Recursive fibonacci function */
3  #include <stdio.h>
4
5  long fibonacci( long );
6
7  int main()
8  {
9      long result, number;
10
11     printf( "Enter an integer: " );
12     scanf( "%ld", &number );
13     result = fibonacci( number );
14     printf( "Fibonacci( %ld ) = %ld\n", number, result );
15     return 0;
16 }
17
18 /* Recursive definition of function fibonacci */
19 long fibonacci( long n )
20 {
21     if ( n == 0 || n == 1 )
22         return n;
23     else
24         return fibonacci( n - 1 ) + fibonacci( n - 2 );
25 }

```

**Function
prototype**

**Initialize
variables**

**Input an
integer**

Call function

Output results

**Define
fibonacci**

recursively

Program Output

```
Enter an integer: 0
Fibonacci(0) = 0

Enter an integer: 1
Fibonacci(1) = 1

Enter an integer: 2
Fibonacci(2) = 1

Enter an integer: 3
Fibonacci(3) = 2

Enter an integer: 4
Fibonacci(4) = 3

Enter an integer: 5
Fibonacci(5) = 5

Enter an integer: 6
Fibonacci(6) = 8

Enter an integer: 10
Fibonacci(10) = 55

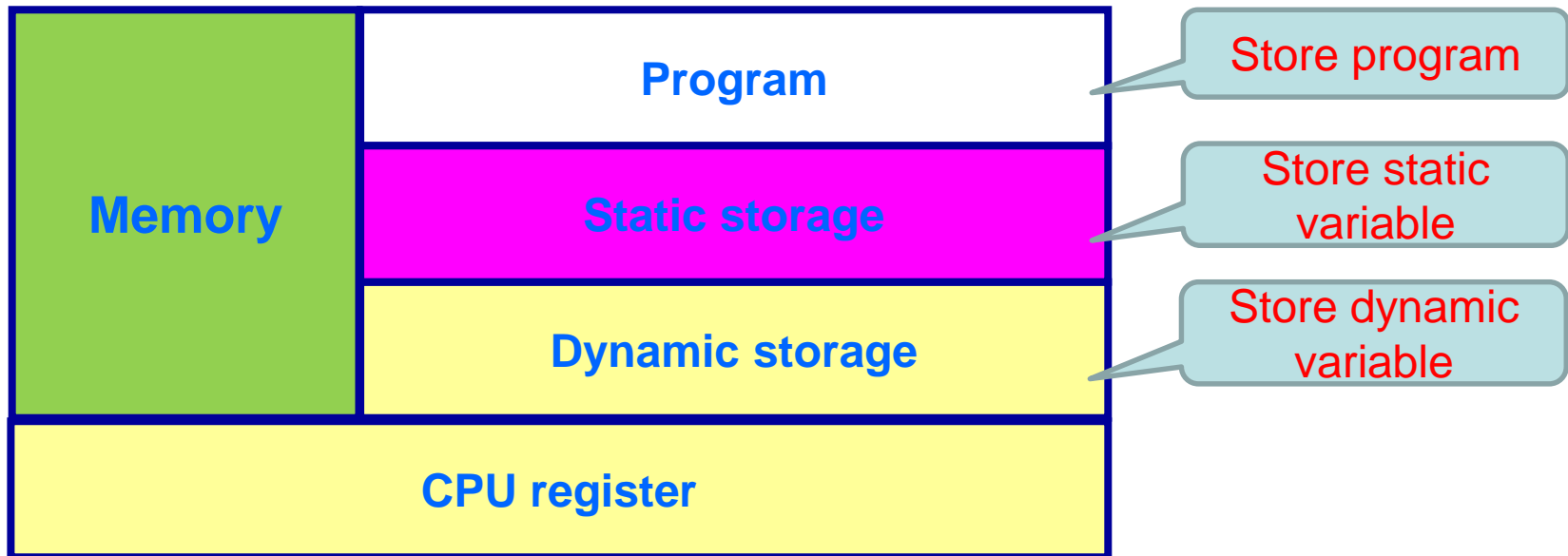
Enter an integer: 20
Fibonacci(20) = 6765

Enter an integer: 30
Fibonacci(30) = 832040

Enter an integer: 35
Fibonacci(35) = 9227465
```


Storage Class

Memory for User



Storage Class

□ Storage class specifiers

- Storage duration – how long an object exists in memory
- Scope – where object can be referenced in program
- Linkage – specifies the files in which an identifier is known

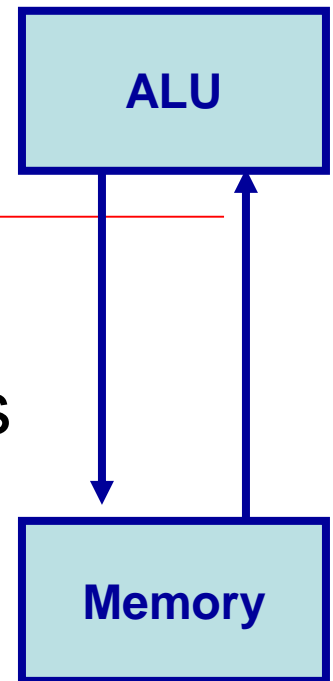
□ There are four storage classes in C:

- Automatic storage class
- Static storage class
- Register storage class
- External storage class

Storage Class

□ Automatic storage

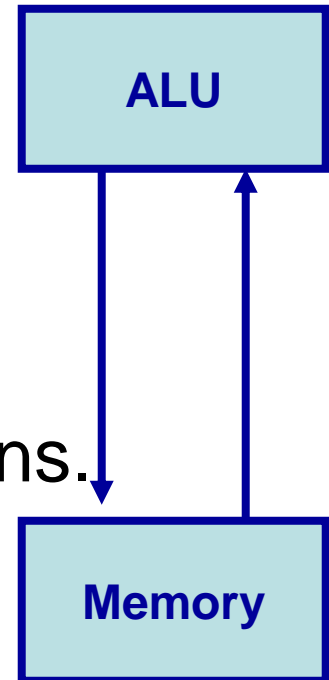
- Object created and destroyed within its block
- auto: default for **local variables**
auto double x, y;
- register: tries to put variable into high-speed registers
 - ◆ Can only be used for automatic variables
register int counter = 1;



Storage Class

□ Static storage

- Variables exist for entire program execution
- Default value of zero
- **static**: local variables defined in functions.
 - ◆ Keep value after function ends
 - ◆ Only known in their own function
- **extern**: default for global variables and functions
 - ◆ Known in any function



Local Variables and External Variable

□ Local Variables

- Declared within a function or a block
- Can Only be used in the function or the block which define them.
- They are created a new each time the function is called and destroyed on return from the function or outside the block.
- The formal arguments are created like local variables.
- Local variables in different functions can share the same name.

Local Variables and External Variable

□ Local Variables

```
main()
{  int a,b;
   a=3;
   b=4;
   printf("main:a=%d,b=%d\n",a,b);
   sub();
   printf("main:a=%d,b=%d\n",a,b);
}

sub()
{  int a,b;
   a=6;
   b=7;
   printf("sub:a=%d,b=%d\n",a,b);
}
```

```
#define N 5
main()
{  int i;
   int a[N]={ 1,2,3,4,5 };
   for(i=0;i<N/2;i++)
   {   int temp;
       temp=a[i];
       a[i]=a[N-i-1];
       a[N-i-1]=temp;
   }
   for(i=0;i<N;i++)
       printf("%d ",a[i]);
}
```

Local Variables and External Variable

□ External Variables

- Defined outside any function
- should be declared if use it before its define
- Their value is retained and is available to any other function which accesses them
- Can be initialized only once
- Can share the same name with local variable, but the latter is useful within the naming function.

Local Variables and External Variable

□ External Variables

```
float  max,min;
float average(float array[], int n)
{  int i;  float sum=array[0];
   max=min=array[0];
   for(i=1;i<n;i++)
   {  if(array[i]>max) max=array[i];
      else if(array[i]<min) min=array[i];
      sum+=array[i];
   }
   return(sum/n);
}
main()
{  int i;  float ave,score[10];
   /*Input */
   ave=average(score,10);
   printf("max=%6.2f\nmin=%6.2f\naverage=%6.2f\n",max,min,ave);
}
```


Scope Rule

□ File scope

- Identifier defined outside function, known in all functions
- Used for global variables, function definitions, function prototypes

□ Function scope

- Can only be referenced inside a function body
- A variable declared within a function is unrelated to variables declared elsewhere, even if they have the same name.

Scope Rule

□ Block scope

- Identifier declared inside a block
 - ◆ Block scope begins at declaration, ends at right brace
- Used for variables, function parameters (local variables of function)
- Outer blocks "hidden" from inner blocks if there is a variable with the same name in the inner block

□ Function prototype scope

- Used for identifiers in parameter list

Scope Rule

	Local Variable			External Variable	
Storage class	auto	register	Local static	External static	extern
Storage mode	Dynamic		Static		
Memory	Dynamic storage	Register	Static storage		
Duration	Within the function		Within the program		
Scope	Function or block which define them			Local file	Other files
Initialization	When calling the function		When compiling the program, only once		
Not Initialized	Not certain		0 or ' '		

- Local variable: default auto
- Register : only a few variables in each function (int , char)
- Local static variable: remain in existence , only be used in the function
- Extern: can extern the scope of external variables

Scope Rule

❑ Wrong way to do it

```
int add_one(int b)
{
    a = b + 1;
}
```

Compilation error!
a is in a different scope

```
int main(void)
{
    int a = 34, b = 1;

    add_one(b); /*call the function add_one*/

    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```

Scope Rule

□ Right way to do it

```
int add_one(int b)
```

```
{
```

```
    b = b + 1;  
    return b;
```

```
}
```

```
int main(void)
```

```
{
```

```
    int a = 34, b = 1;
```

```
    a=add_one(b); /*call the function add_one*/
```

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

```
    return 0;
```

```
}
```

```

1
2  /*    A scoping example */
3  #include <stdio.h>
4
5  void a( void );    /* function prototype */
6  void b( void );    /* function prototype */
7  void c( void );    /* function prototype */
8
9  int x = 1;          /* global variable */
10
11 int main()
12 {
13     int x = 5;      /* local variable to main */
14
15     printf("local x in outer scope of main is %d\n", x );
16
17     {               /* start new scope */
18         int x = 7;
19
20         printf( "local x in inner scope of main is %d\n", x );
21     }               /* end new scope */
22
23     printf( "local x in outer scope of main is %d\n", x );
24
25     a();            /* a has automatic local x */
26     b();            /* b has static local x */
27     c();            /* c uses global x */
28     a();            /* a reinitializes automatic local x */
29     b();            /* static local x retains its previous value */
30     c();            /* global x also retains its value */

```

**Function
prototypes**

**Initialize
global
variable**

**Initialize local
variable**

**Initialize local
variable in
block**

Call functions

```

31
32     printf( "local x in main is %d\n", x );
33     return 0;
34 }
35
36 void a( void )
37 {
38     int x = 25;  /* initialized each time a is called */
39
40     printf( "\nlocal x in a is %d after entering a\n", x );
41     ++x;
42     printf( "local x in a is %d before exiting a\n", x );
43 }
44
45 void b( void )
46 {
47     static int x = 50;  /* static initialization only */
48                        /* first time b is called */
49     printf( "\nlocal static x is %d on entering b\n", x );
50     ++x;
51     printf( "local static x is %d on exiting b\n", x );
52 }
53
54 void c( void )
55 {
56     printf( "\nglobal x is %d on entering c\n", x );
57     x *= 10;
58     printf( "global x is %d on exiting c\n", x );
59 }

```

Function definitions

Program Output

```
local x in outer scope of main is 5  
local x in inner scope of main is 7  
local x in outer scope of main is 5
```

```
local x in a is 25 after entering a  
local x in a is 26 before exiting a
```

```
local static x is 50 on entering b  
local static x is 51 on exiting b
```

```
global x is 1 on entering c  
global x is 10 on exiting c
```

```
local x in a is 25 after entering a  
local x in a is 26 before exiting a
```

```
local static x is 51 on entering b  
local static x is 52 on exiting b
```

```
global x is 10 on entering c  
global x is 100 on exiting c  
local x in main is 5
```



```

#include <stdio.h>
int i=1;
main()
{  static int a;
   register int b=-10;
   int c=0;
   printf("-----MAIN-----\n");
   printf("i:%d a:%d \
        b:%d c:%d\n",i,a,b,c);
   c=c+8;
   other();
   printf("-----MAIN-----\n");
   printf("i:%d a:%d \
        b:%d c:%d\n",i,a,b,c);
   i=i+10;
   other();
}

```

```

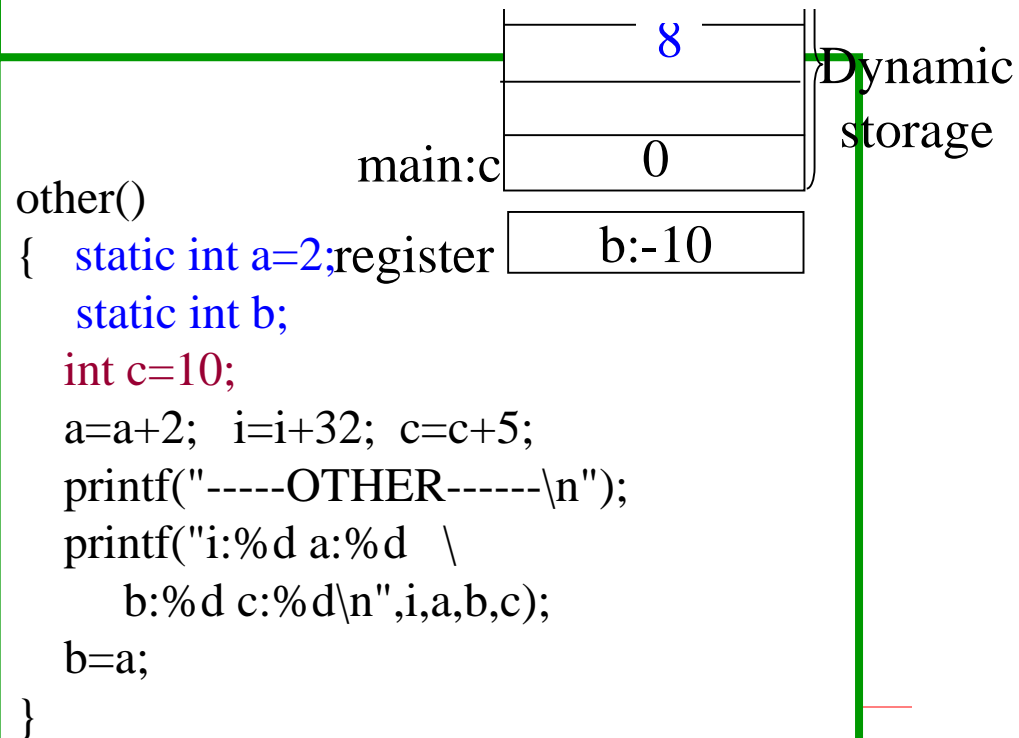
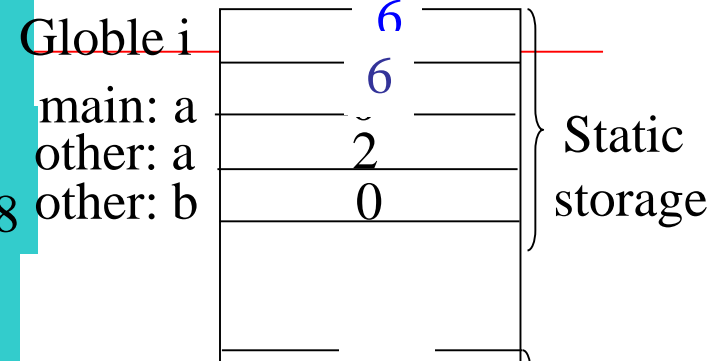
-----Main-----
i:1  a:0  b:-10 c:0

-----Other-----
i:33 a:4  b:0  c:15

-----Main-----
i:33 a:0  b:-10 c:8

-----Other-----
i:75 a:6  b:4  c:15

```



Summary

- ❑ Definition, declaration and calling of functions
 - ❑ Distinguish different kinds of arguments of function, and master how the values is transferred by arguments
 - ❑ Scope rule: difference between local variable and external variable
 - ❑ Understand how recursion works
-

Thank you!