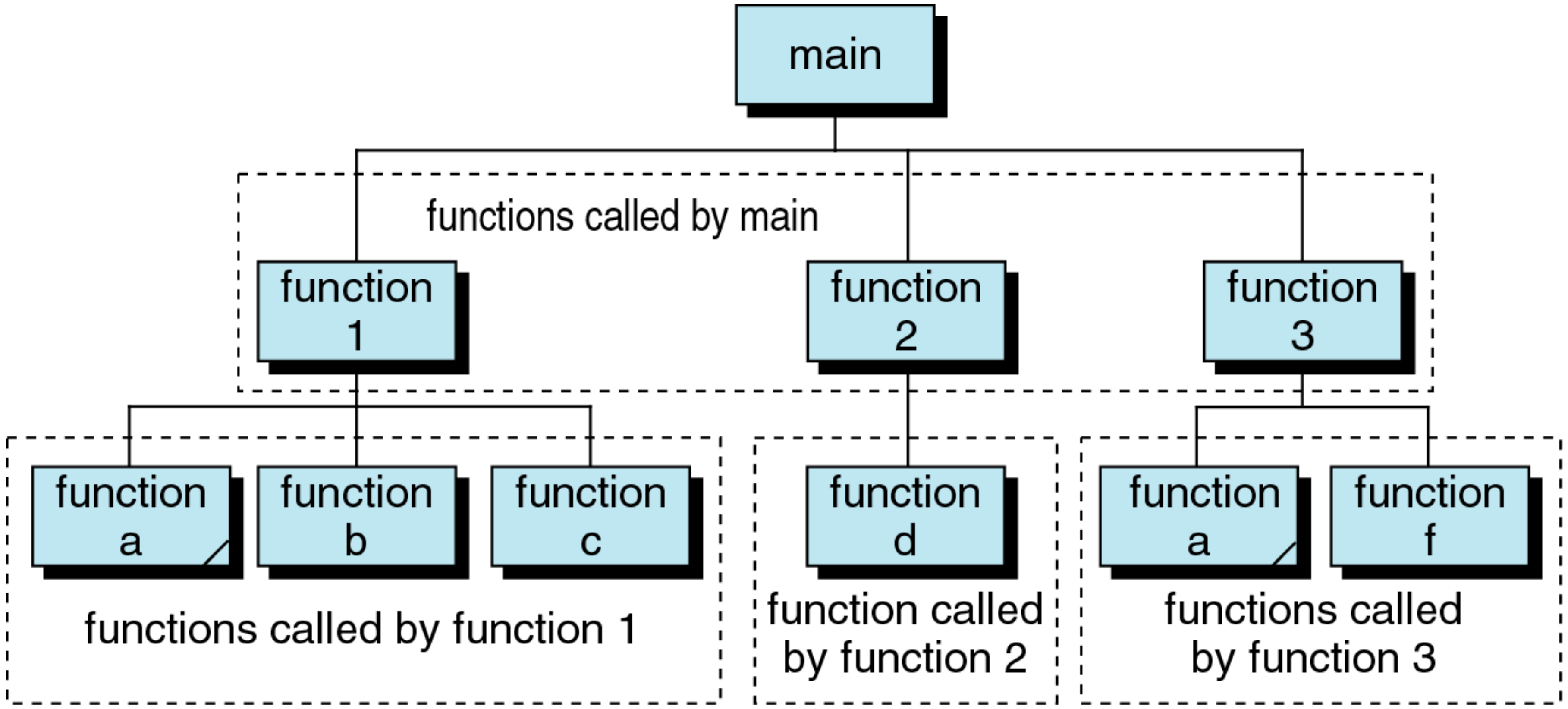
FUNCTIONS IN C

A PROBLEM can be solved easily if it is decomposed into parts. Similarly a C program decomposes a program into its component functions.

Big problems require big programs – too big to be written all at one time or to be written by a single programmer. Thus by decomposing a program into functions, **we divide the work among several programmers.**

We can test the components of programs separately.
We can change one function with out changing or affecting the other functions.


main
functions called by main
function 1
function 2
function 3
function a
function b
function c
functions called by function 1
function d
function called by function 2
function a
function f
functions called by function 3

Functions provide a way to reuse code that is required in more than one place in your program.

A C program is made up of one or more functions, exactly one of which must be named **main**.

Execution begins with **main** and terminates when main does.

The **main function** can call other functions to do specific jobs.

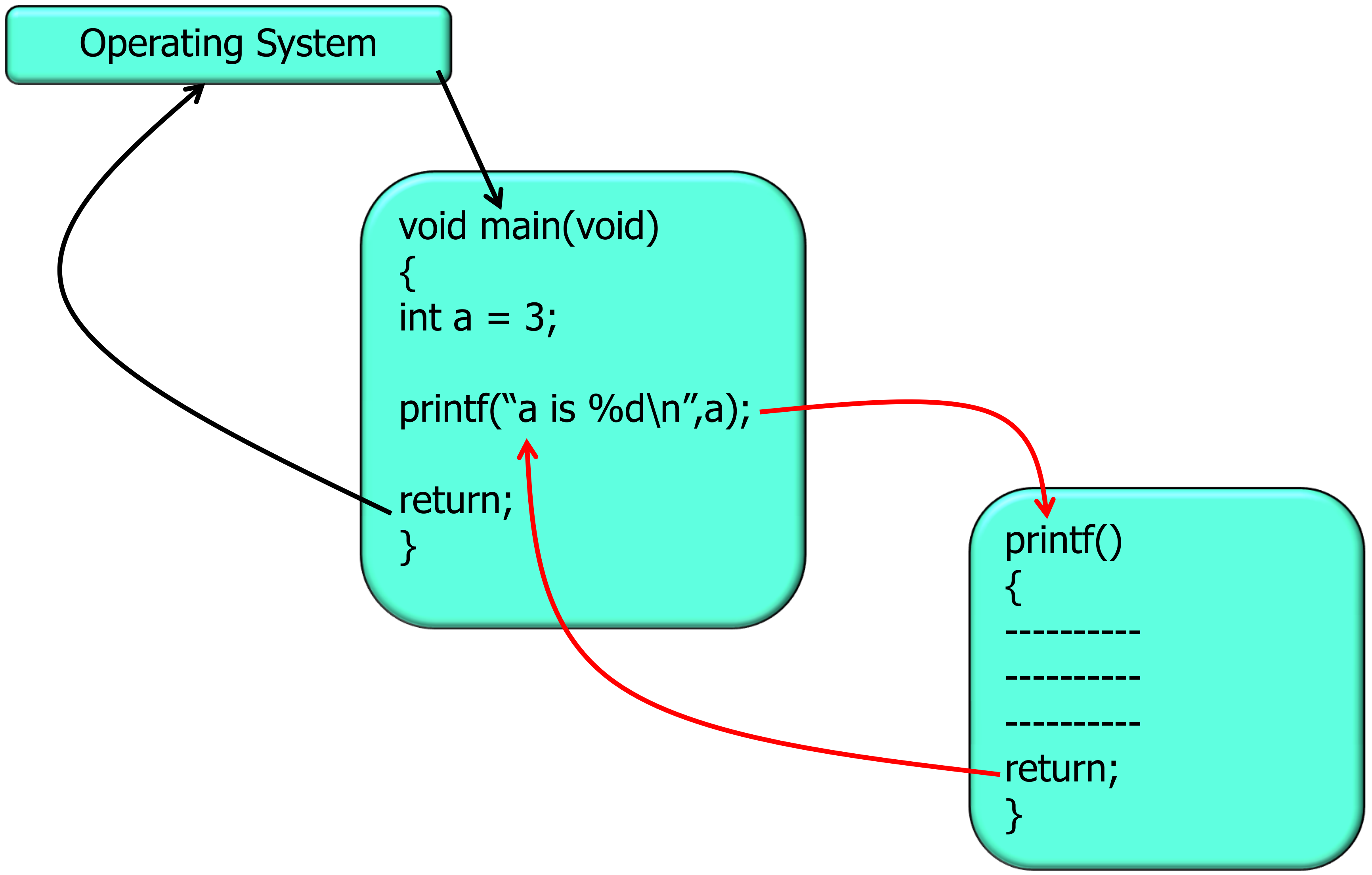
Function is an independent module and each function solves a part of the problem.

Note : A C function can be invoked or called by another function.

```
        OPERATING SYSTEM
                                MAIN FUNCTION.
        MAIN FUNCTION
                                OTHER FUNCTIONS.
        OTHER FUNCTIONS
                                OTHER FUNCTIONS
```

Once after completing the task, the called function returns the control to the caller function.

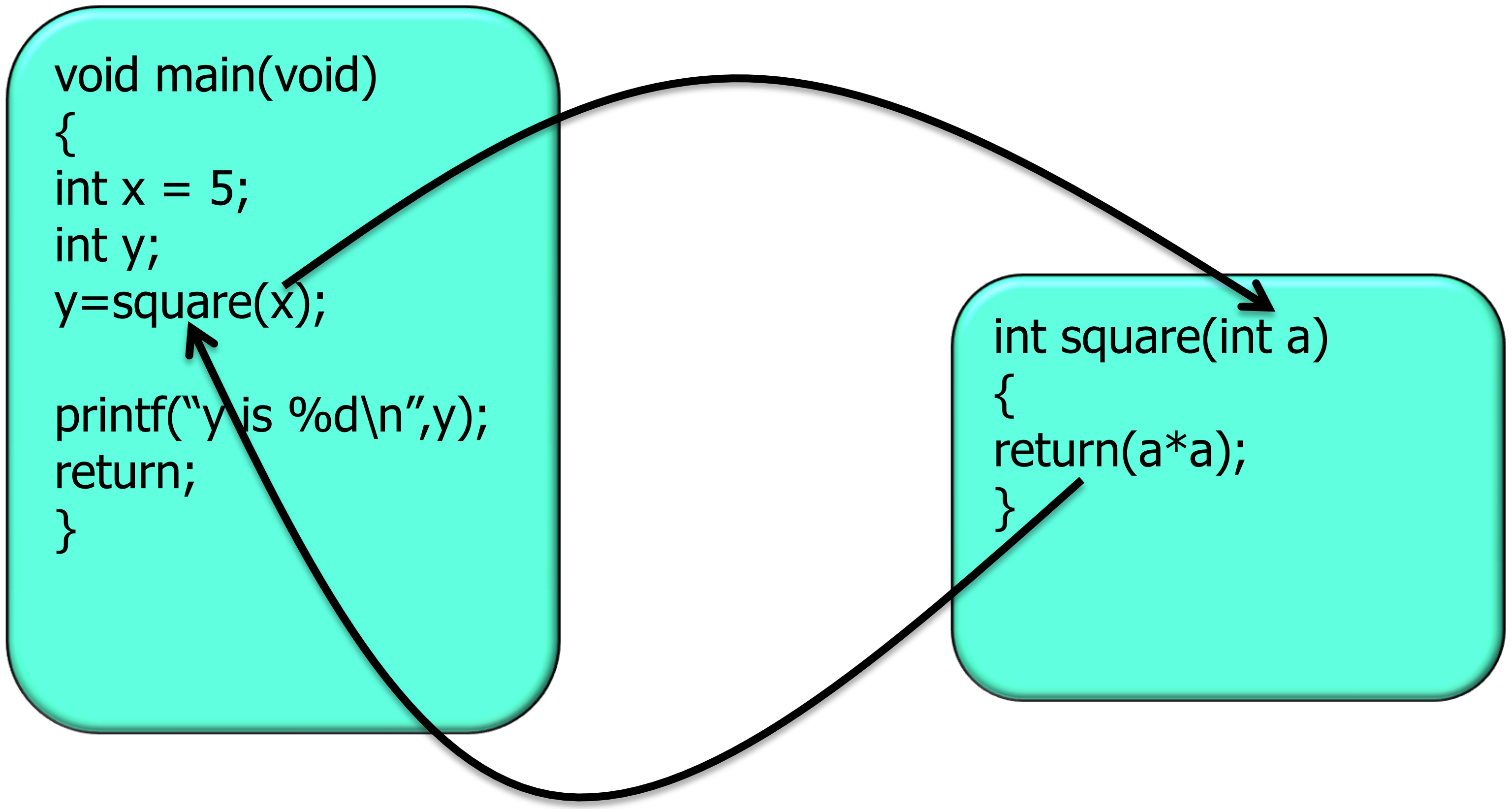
When main completes its operations, control returns to the OS.



```
void main(void)
{
int a = 3;

printf("a is %d\n",a);

return;
}
```

```
printf()
{
----------
----------
----------
return;
}
```

A function can call or invoke another function.
The invoking function passes information to the invoked function.
The invoked function may return information to the invoked.



```
void main(void)
{
int x = 5;
int y;
y=square(x);

printf("y is %d\n",y);
return;
}
```

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

The following combinations of information passing are possible:

- ❖The invoker passes information and the invoked returns information.
- ❖The invoker passes information but the invoked returns none.
- ❖The invoker passes nothing but the invoked returns information.
- ❖The invoker passes nothing and the invoked returns none.

The name of a function is used in three ways.

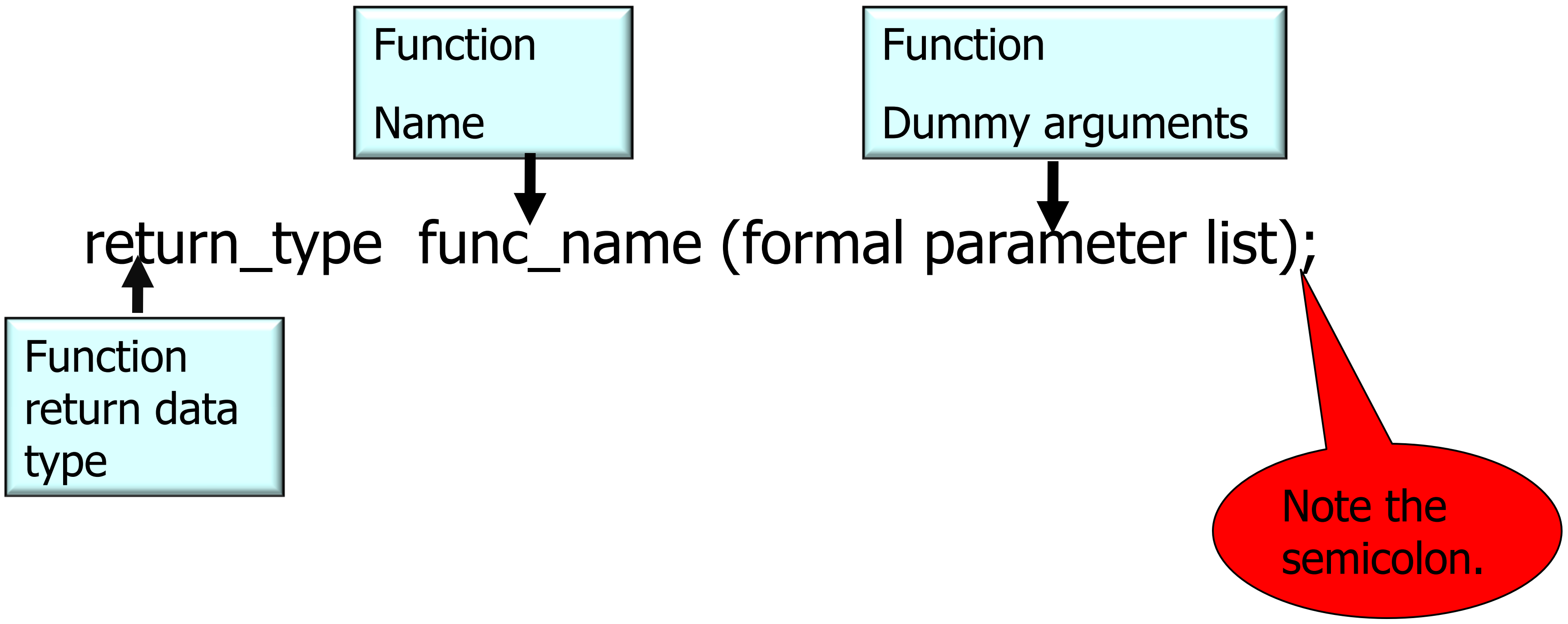
(a) for declaration
(b) in a call
(c) for definition

Function declaration Statement

A function must be first **declared** and **defined**.

Function definition Statements

A Function Declaration



```
return_type  func_name (formal parameter list);
```

A function declaration consists of three parts:

The **type of data** to be returned (namely void, int, float, char, double).

The **name of a function** (which requires to call the function). The function name is a valid C identifier.

Formal parameter list:
The parameters are the place holders for the arguments that the function expects.

If the program has no parameters, code void is used.

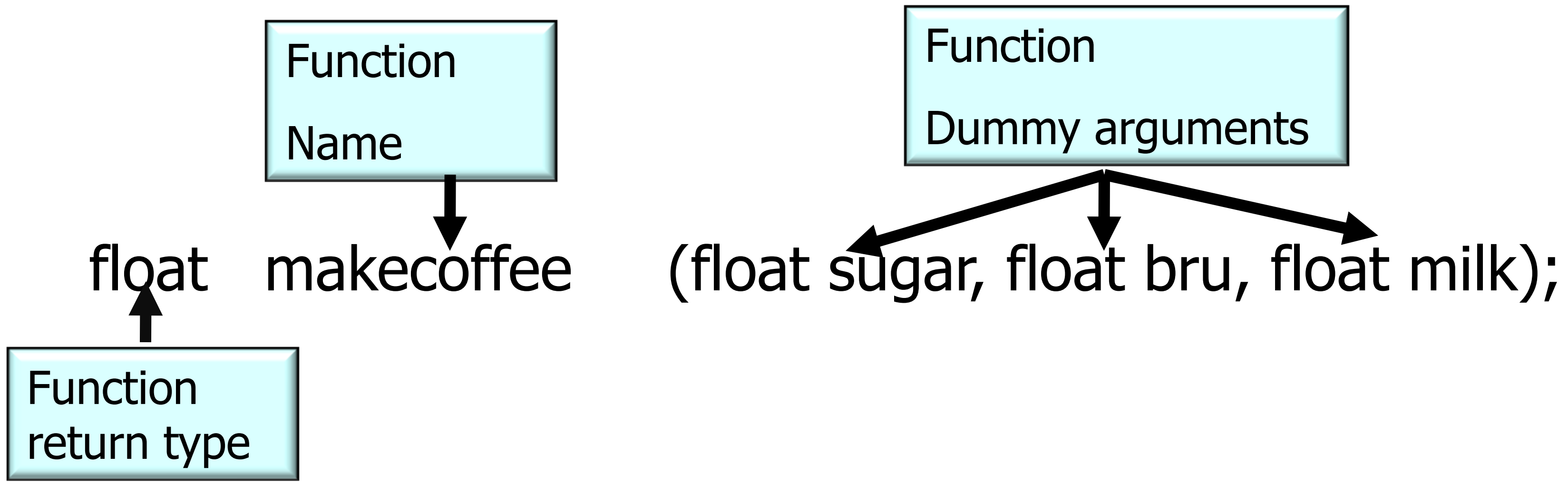
If the program has more than one parameter, the parameters are separated with commas. The prototype declaration statement is terminated by a semi colon.

NOTE:

A function declaration contains no code.

Example:

```
float makecoffee(float sugar, float bru, float milk);
int addnum(int num1,int num2);
void printnum(int number);
void printstar(void);
```

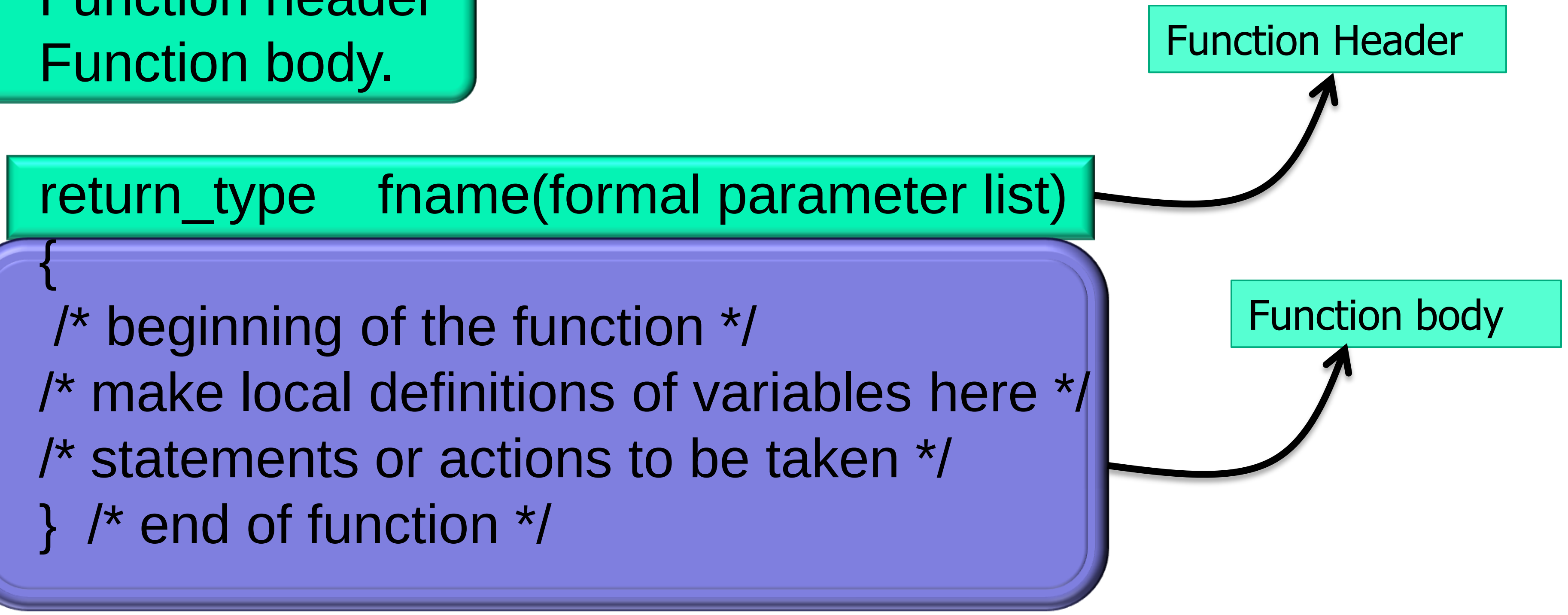


FUNCTION DEFINITION:

It contains the code for a function.

It is made up of two parts:

Function header
Function body.



```
return_type    fname(formal parameter list)
{
 /* beginning of the function */
/* make local definitions of variables here */
/* statements or actions to be taken */
}  /* end of function */
```

NOTE :

There is no semi colon at the end of the first function definition statement.

The format of a function header is:

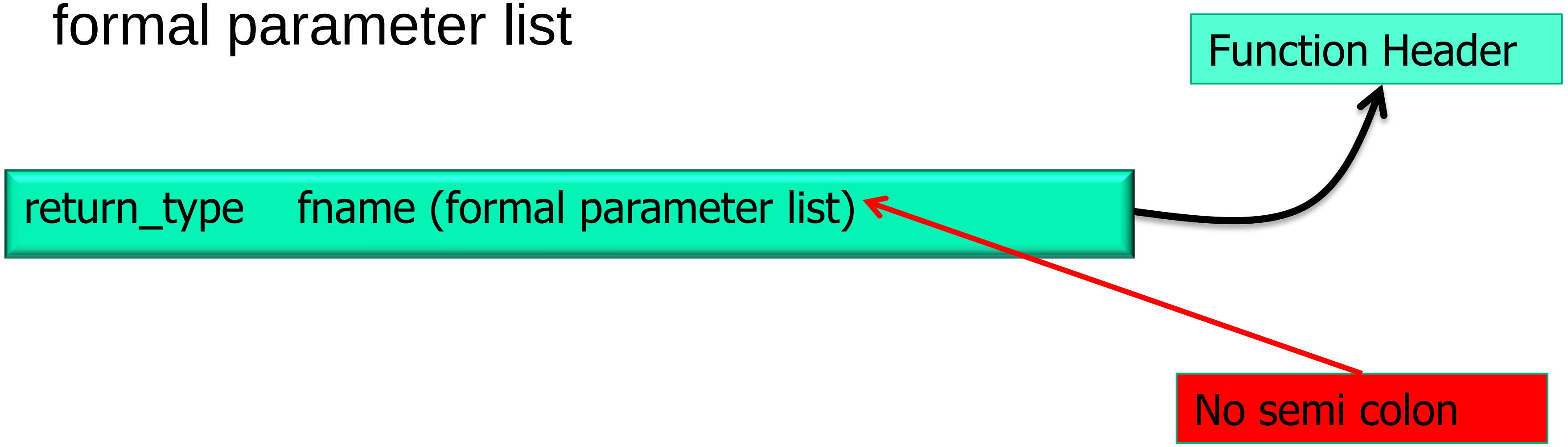
return_type fname(formal parameter list)

The function header has three parts:

return type

function name

formal parameter list



Return Type:
A function return type may be void, int, char, float, pointer, double. long double.

If the return value is null (nothing), it has to be specified as void.

Function name:
It is a valid “C” identifier.

Formal parameter list:
The parameters are place holders for the arguments that the function expects.
The parameters are enclosed in parentheses and separated by commas.
There is no limit to the number of parameters a function can have.
If the function has no parameters (ie. If it not receiving any data from the invoker) then the empty list is declared by the key word void.

Function body:

The body of a function is a compound statement. ie. It is enclosed between open { and closing } braces.
The function body contains local variable declarations and statements and terminated by a return statement.

Return Statement:
When a return statement in a function is executed, the function returns to its invoker.
The return statement is optional in a function that does not return a value but, if used it is written as:

return;

If the return is not specified, C will assume it is of type integer.
If a function that does not have a return statement, the function returns to the invoker after the last statement in the function's body is executed.

A function that returns a value must have at least one return statement which may be written as:

```
return(expression);
                  or
return expression;
```

Example:

```
return(30*6+20);         /*common*/
         or
return 30*6+20;  /* legal */
```

A function can have any number of return statements. Of course only one return statement is executed per invocation, because the return statement returns control and a value to the invoking function.

NOTE: As a rule of thumb keep the number of return statements small; otherwise the function becomes hard to understand, hard to debug and hard to alter.

A Function Call:

A functional call is an post fix expression. (Second highest precedence)

The operand in the functional call is the function name.

The operator is the parenthesis set ( ) which contains the actual parameters.

The actual parameters are the values sent to the calling function.

The actual parameters must match with the function formal parameters in type and order.

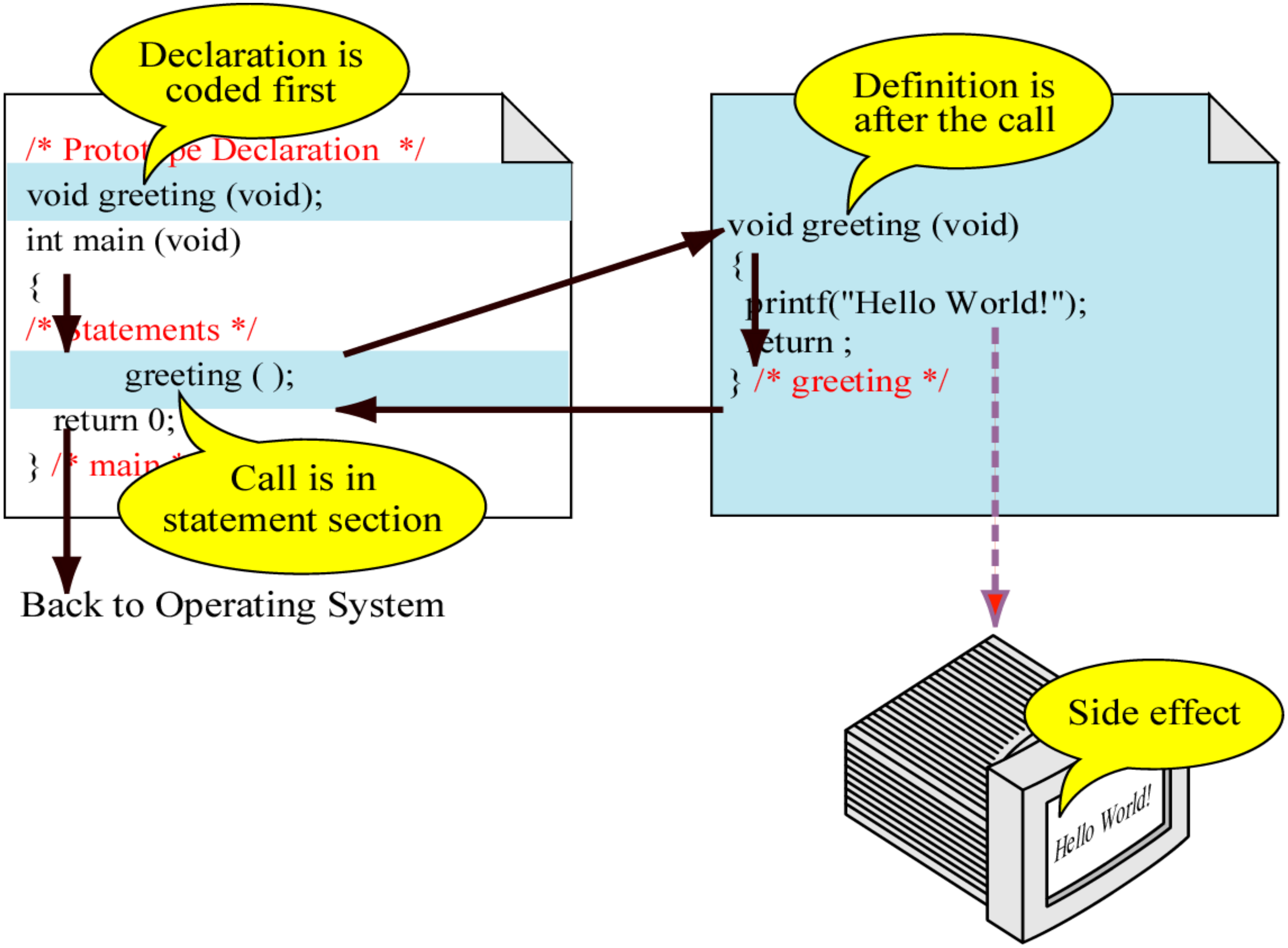
The actual parameters are separated by commas.

Find the errors in the following function definition:

```
void fun(int x,int y);
{
 int z;
return z;
}

void fun1(int x, y)
{
 int z;
return z;
}
```

What is the error in the following prototype:

```
int sum(int x, y);

int sum(int x, int y)

void sum(void,void);
```



```
/* Prototype Declaration  */
void greeting (void);
int main (void)
{
/* Statements */
          greeting ( );
   return 0;
} /* main */
```

```
void greeting (void)
{
  printf("Hello World!");
  return ;
} /* greeting */
```

Declaration is coded first
Call is in statement section
Back to Operating System
Definition is after the call
Side effect
Hello World!

```
/* program using function */
#include<stdio.h>

float far2cen(float far);

int main(void)
{
 float far,cen;
scanf("%f",&far);
cen = far2cen(far);
printf("Farenheit temp = %f\n",far);
printf("Centigrade temp is %f\n",cen);
return 0;
}

float far2cen(float far)
{
return((5.0/9.0)*(far-32.0));
}
```

```
/* program in c to add two numbers using functions */

#include<stdio.h>
float addnumber(float num1, float num2);

void main(void)  {
float n1,n2,sum;
printf(“please enter two numbers”);
scanf(“%f %f”,&n1,&n2);
sum = addnumber(n1,n2);
printf(“%d + %d = %d\n”,n1,n2,sum);
return; }

float addnumber(float num1, float num2)
{
 float result;
result= num1 + num2;
return(result);
}
```

```
/*program in c to find the hypotenuse,
are, perimeter of a right angled triangle given the two sides */

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

float hypotenuse(float side1,float side2);
float areatri(float side1, float side2);
float perimeter(float side1, float side2);

void main(void)  {
float s1,s2,s3,area, hypo, peri;
printf(“please input the two sides ”);
scanf(“%f %f”,&s1,&s2);
area = areatri(s1,s2);
peri = perimeter(s1,s2);
hypo = hypotenuse(s1,s2);
printf(“The sides of the triangle is %f
%f\n”,s1,s2);
printf(“The area = %f\n”,area);
printf(“The perimeter = %f\n”,peri);
printf(“The hypotenuse = %f\n”,hypo);
return;    }
```

```
float hypotenuse(float side1,float side2)
{
return(sqrt(side1*side1 + side2*side2));
}

float areatri(float side1, float side2);
{
return(0.5*side1*side2);
}

float perimeter(float side1, float side2);
{
float side3;
side3 = hypotenuse(side,side2);
return(side+side2+side3);
}
```

Side Effect in a function:

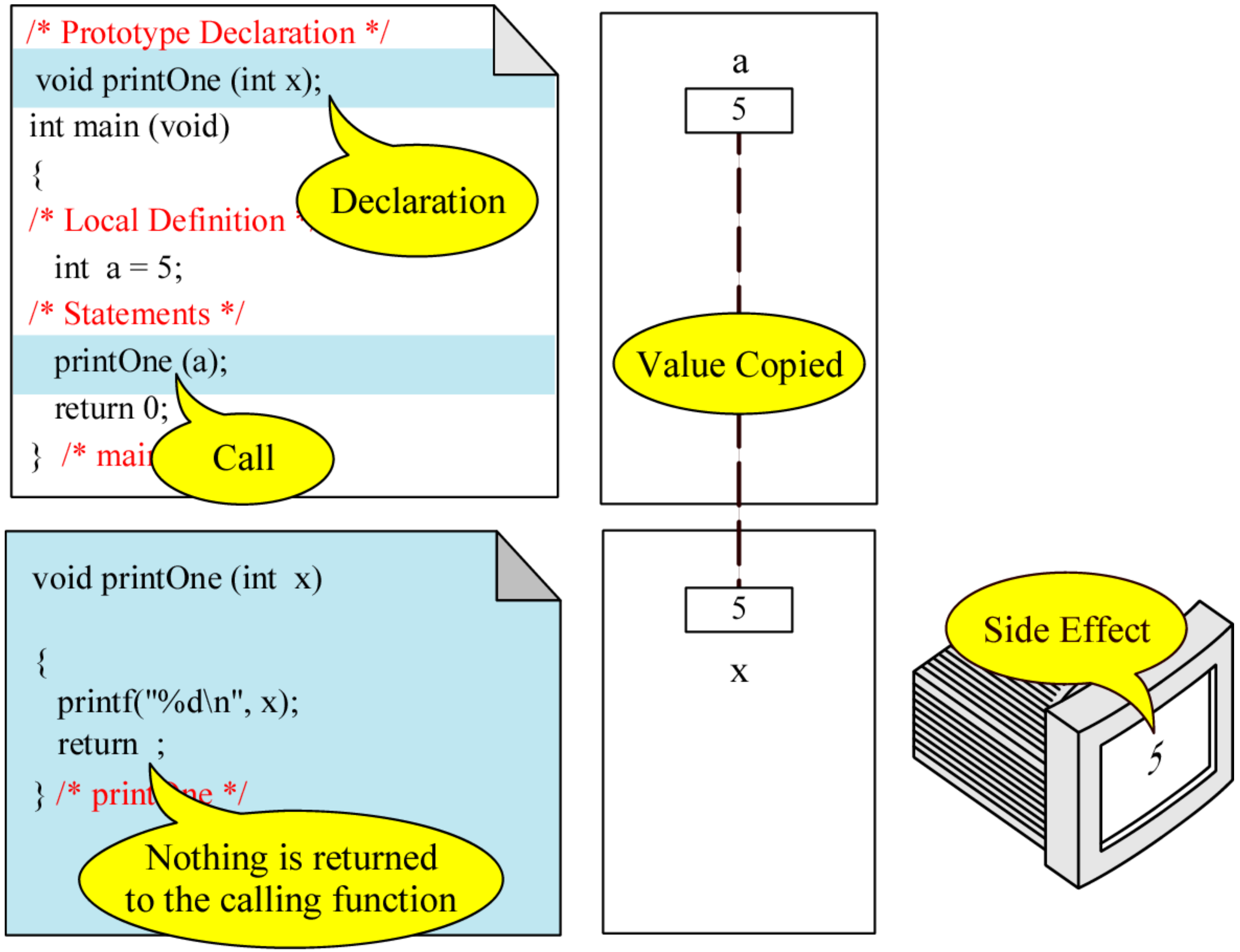
A change of state in the program takes place due to the action of a function, which is termed as a side effect of a function.
The side effect may be:

* accepting data from outside the program.
* sending data out of the program to a file or to the monitor.
* Changing the value of a variable in the calling function.

Call by Value:

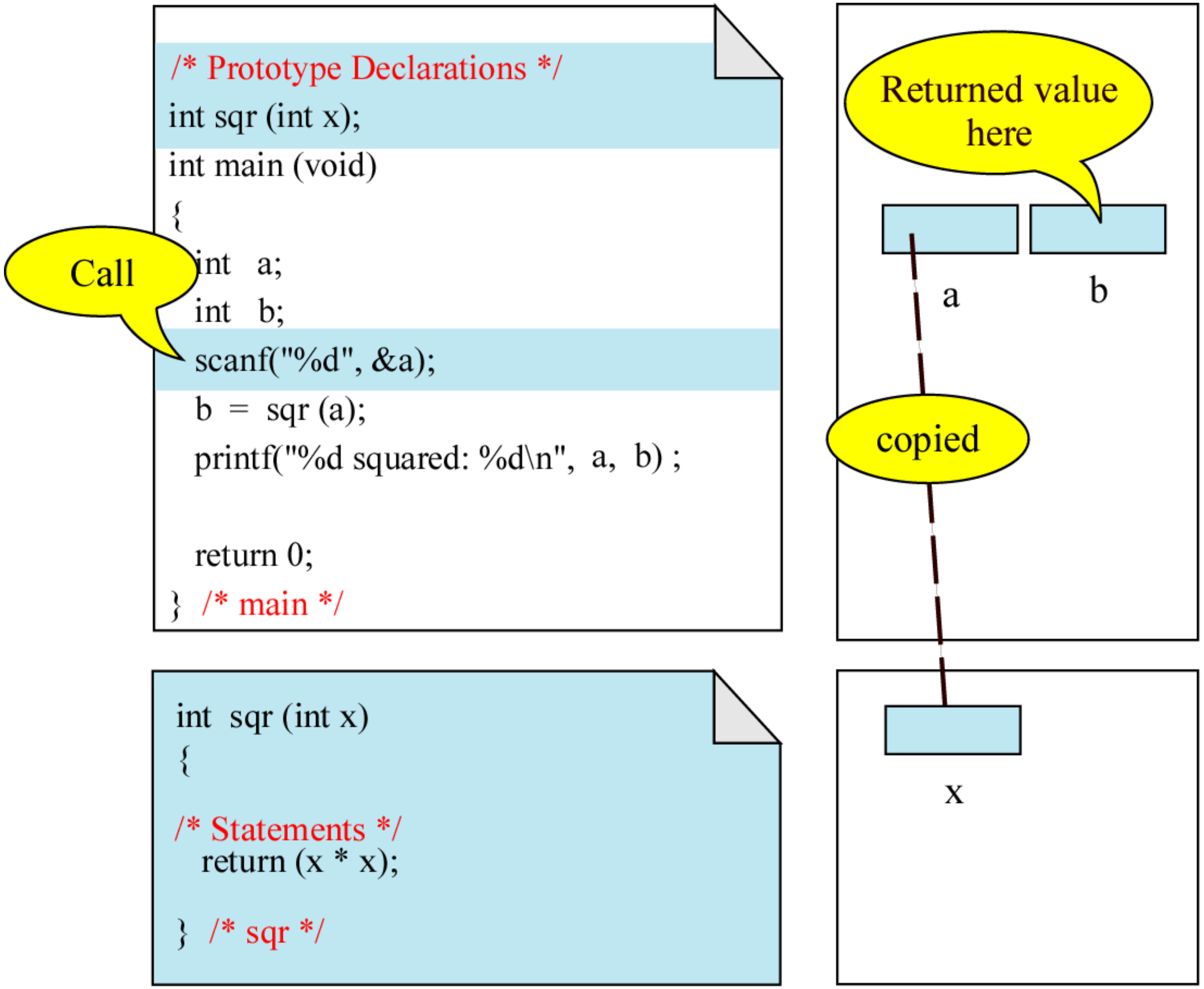
1. Every argument to a function is an expression, which has a value.
2. C passes an argument to an invoked function by making a copy of the expression value, storing it in a temporary cell.
3. Only the copy of the value is passed to the function argument.

NOTE : The original data in the calling function are safe and unchanged.
As only the copy of the values are passed to the function, this method is called call by values.



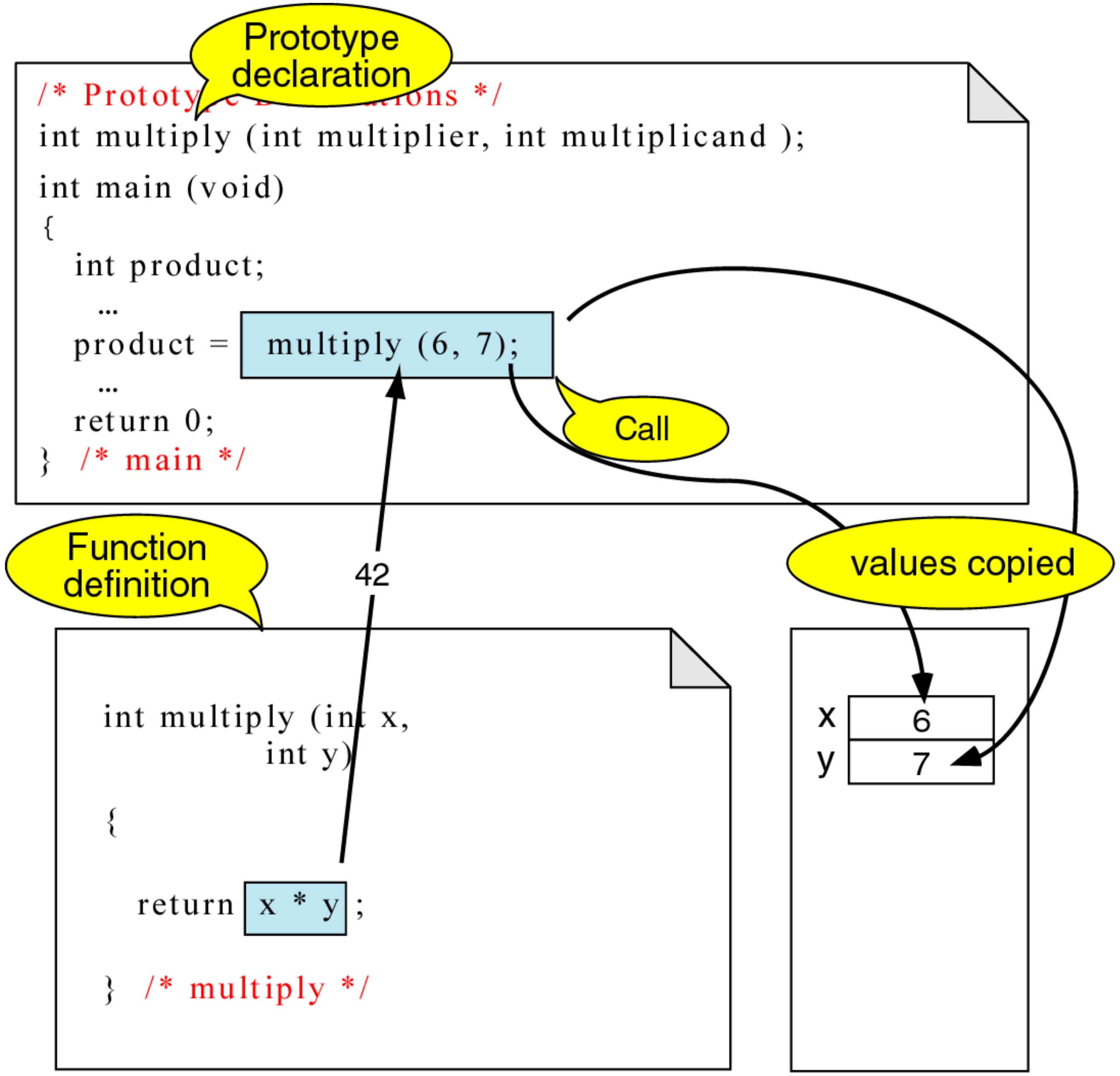
```
/* Prototype Declaration */
 void printOne (int x);
int main (void)
{
/* Local Definition *
   int  a = 5;
/* Statements */
   printOne (a);
   return 0;
}  /* mai
```

```
void printOne (int  x)

{
   printf("%d\n", x);
   return  ;
} /* print    ne */
```



```
/* Prototype Declarations */
int sqr (int x);
int main (void)
{
   int   a;
   int   b;
   scanf("%d", &a);
   b  =  sqr (a);
   printf("%d squared: %d\n",  a,  b) ;

   return 0;
}  /* main */
```

```
int  sqr (int x)
{

/* Statements */
   return (x * x);

}  /* sqr */
```



```
/* Prototype Declarations */
int multiply (int multiplier, int multiplicand );

int main (void)
{
   int product;
     ...
   product = multiply (6, 7);
     ...
   return 0;
}  /* main */
```

```
int multiply (int x,
              int y)

{

   return x * y ;

}  /* multiply */
```

Example: Call by value

/* program to find the square of a number */

```
#include<stdio.h>
int square(int num);
int main(void)
{
int s1,result;
scanf(“%d”,&s1);
result = square(s1);
printf(“The square of %d is %d \n”, s1,result);
return 0;
}
int square(int num)
{
return(num*num);
}
```

```
#include<stdio.h>
void modify(int x);
int main(void)
{
  int x = 5;
printf("value of x before calling
            modify is %d\n",x);
modify(x);
printf("value of x after calling
            modify is %d\n",x);
return 0;
}
```

```
void modify(int x)
{
printf("value of local variable x in
modify is %d\n",x);
x = 10;
printf( value of x after reassigning
local variable x in modify is
%d\n",x);
return;
}
```

Result:

```
value of x before calling modify is 5
value of local variable x in modify is 5
value of x after reassigning local variable  x in modify is 10;
value of x after calling modify is 5
```

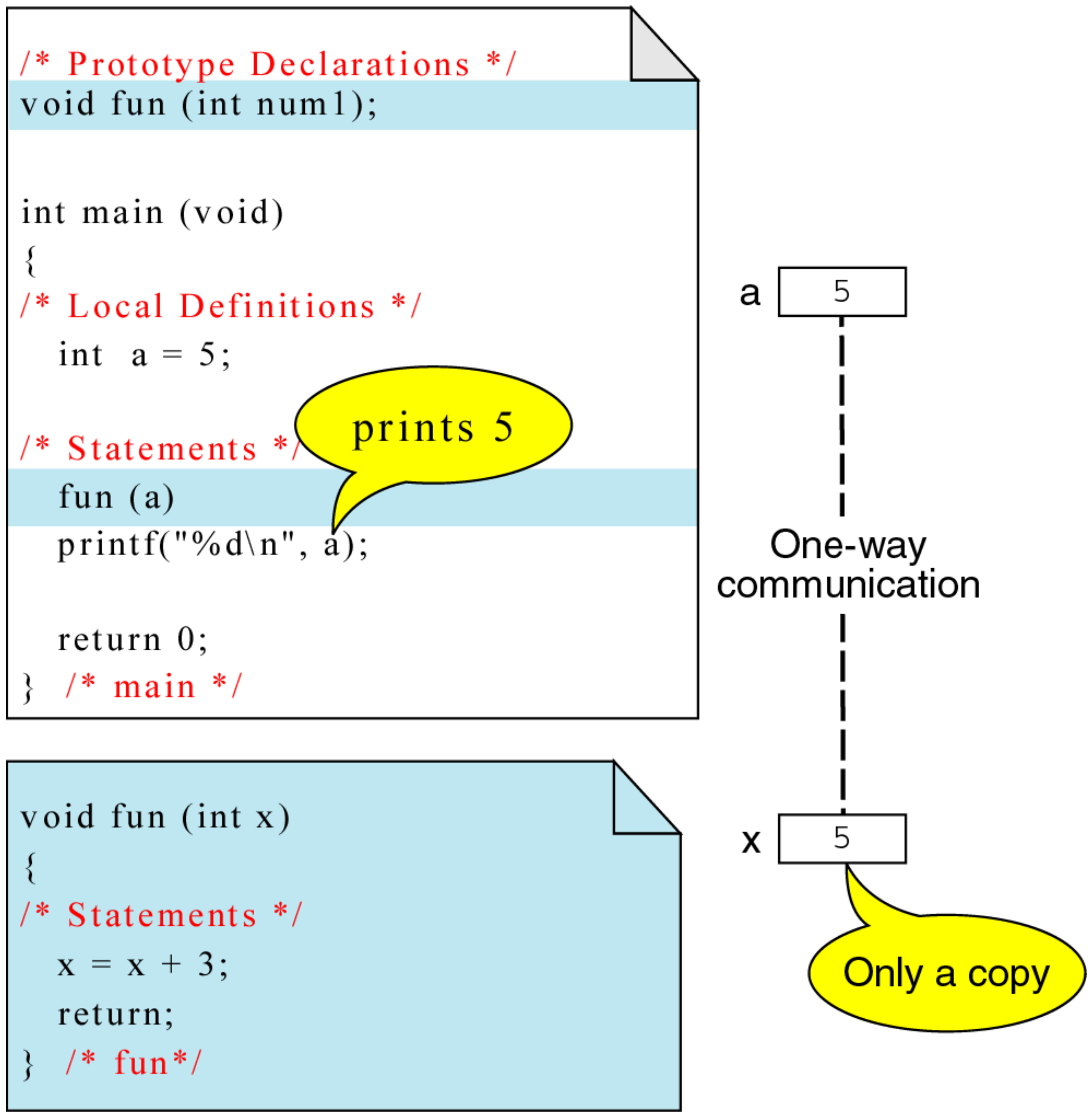
Call by reference:

It links the variable identifiers in the calling function to their corresponding parameters in the called function.

ie., When the called function changes a value in a variable, then it actually changes the variables in the calling function. This is done by passing an address to the called function.

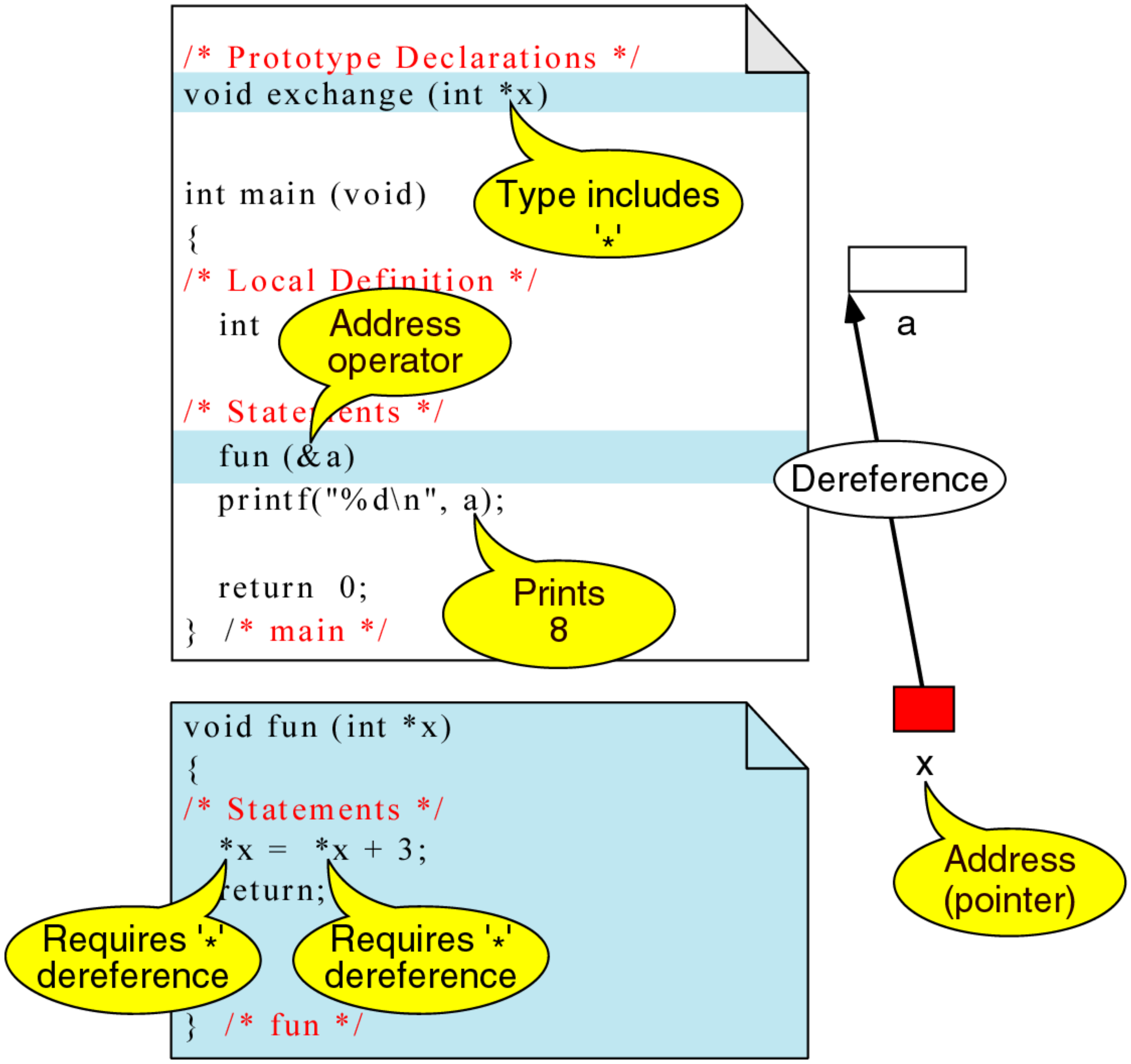
& - the address operator
 * - indirection operator.

If ‘addr’ is a variable that contains an address then *addr means the value pointed by the address stored in the variable ‘addr’. Here ‘addr’ is a pointer variable.



```
/* Prototype Declarations */
void fun (int num1);


int main (void)
{
/* Local Definitions */
   int  a = 5;

/* Statements */
   fun (a)
   printf("%d\n", a);

   return 0;
}  /* main */
```

```
void fun (int x)
{
/* Statements */
   x = x + 3;
   return;
}  /* fun*/
```



```
/* Prototype Declarations */
void exchange (int *x)

int main (void)
{
/* Local Definition */
   int

/* Statements */
   fun (&a)
   printf("%d\n", a);

   return  0;
}  /* main */
```

```
void fun (int *x)
{
/* Statements */
   *x =  *x + 3;
   return;

}  /* fun */
```

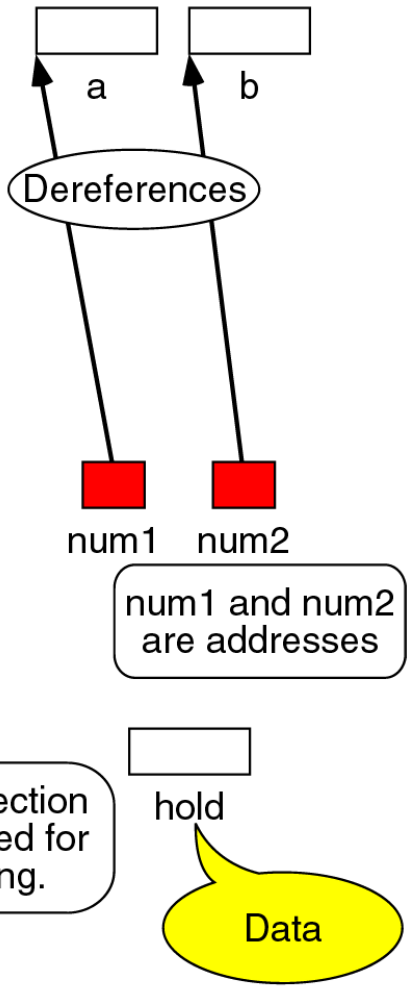
```
/* Prototype Declarations */
void exchange (int *num1,
                  int *num2);

int main (void)
{
/* Local Definitions */
   int  a;
   int  b;
/* Statements */
   ...
   exchange (&a, &b);
   ...
 return 0;
}  /* main */
```

Note that the type includes an asterisk.

Address operators

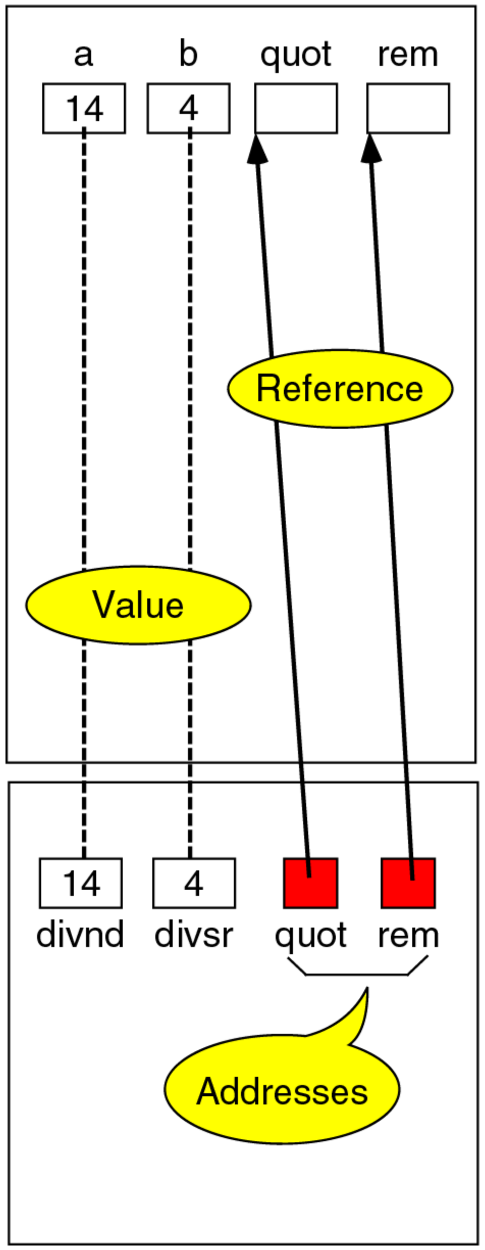
```
void exchange (int *num1,
                   int *num2)
{
/* Local Definitions */
   int  hold;
/* Statements */
   hold   =  *num1;
 *num1   =  *num2;
 *num2   =   hold;
   return;
}  /* exchange */
```

Note the indirection operator is used for dereferencing.



```
/* Prototype Declarations */
void divide (int  divnd, int  divsr,
                int *quot,  int *rem);
int  main (void)
  {
/* Local Definitions */
   int  a;
   int  b;
  int  quot;
   int  rem
/* Statements */
     ...
   divide (a, b, &quot, &rem);
     ...
    return 0;
  } /* main */
```

```
void  divide (int  divnd,
                int  divsr,
                int *quot,
                int *rem)
{
/* Statements */
   *quot =  divnd / divsr;
   *rem  =  divnd % divsr;
    return  ;
}  /* divide */
```



Recursive Functions

- Recursion is a term describing functions which are called by themselves (A function that calls itself)

- Recursive function has two elements:

  Each call either solves one part of the problem or it reduces the size of the problem.

  The statement that solves the problem is known an base case. Every recursive function must have a base case. The rest of the function is known as the general case.

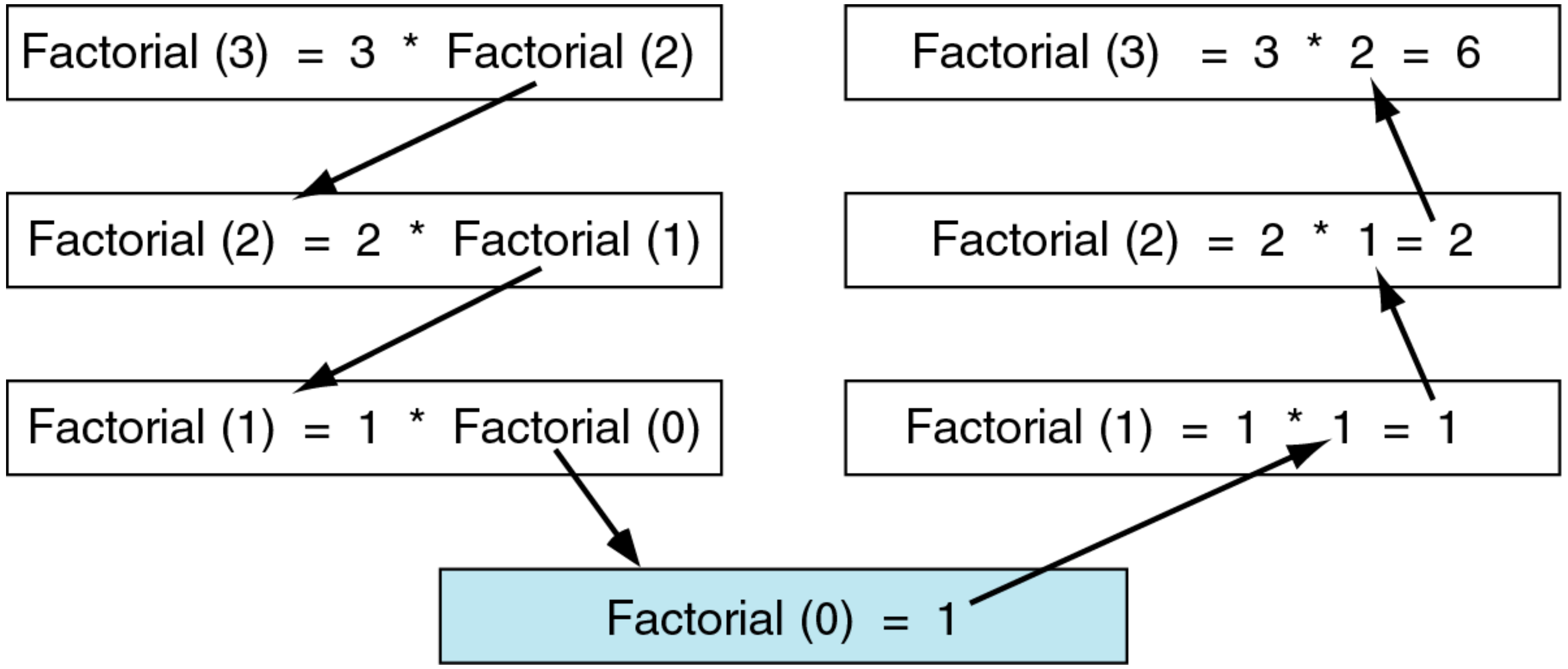
- Recursion is very useful in mathematical calculations and in sorting of lists.

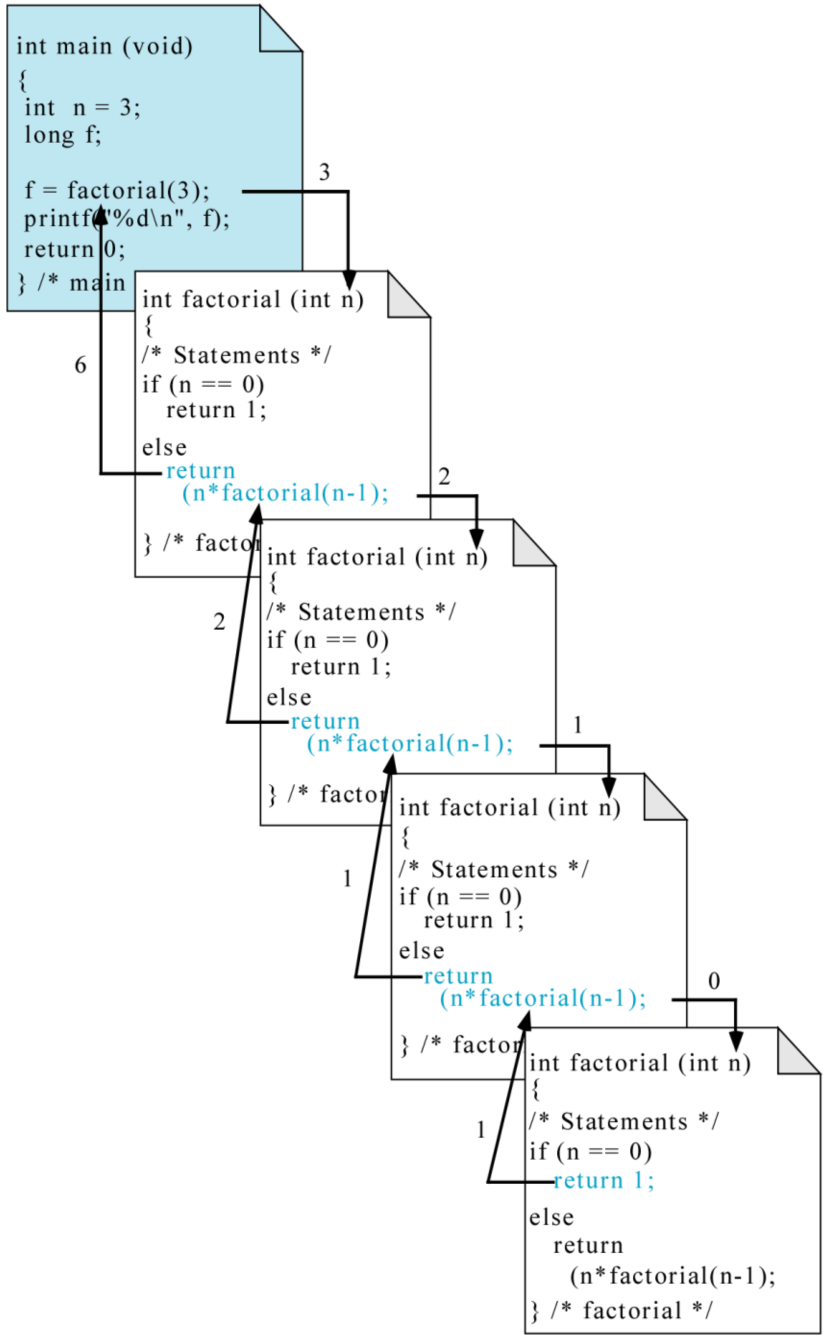
Recursive Functions (cont.)

- Example: factorial

  n! = n * (n – 1) * (n – 2) * … * 1

- Recursive relationship:
  - (n! = n * (n – 1)!)
  - 5! = 5 * 4!
  - 4! = 4 * 3!…
- Base case (1! = 0! = 1)


Factorial (3) = 3 * Factorial (2)
Factorial (2) = 2 * Factorial (1)
Factorial (1) = 1 * Factorial (0)
Factorial (0) = 1
Factorial (1) = 1 * 1 = 1
Factorial (2) = 2 * 1 = 2
Factorial (3) = 3 * 2 = 6



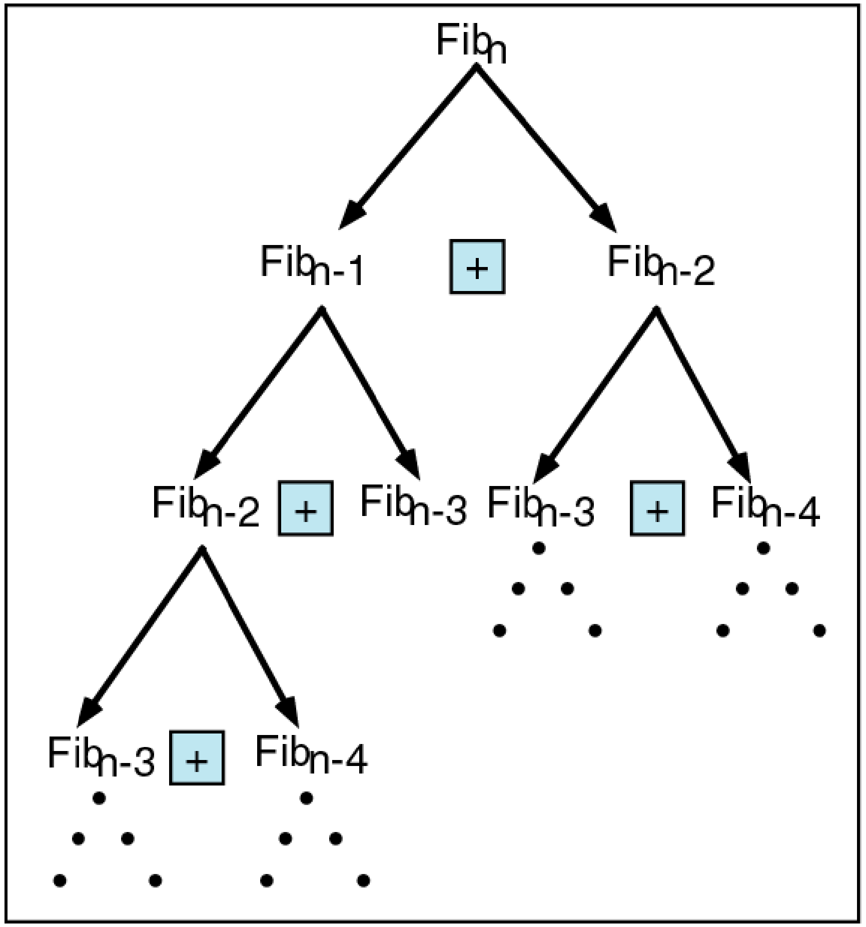
```
int main (void)
{
 int  n = 3;
 long f;

 f = factorial(3);
 printf("%d\n", f);
 return 0;
} /* main
```

```
int factorial (int n)
{
/* Statements */
if (n == 0)
   return 1;

else
   return
     (n*factorial(n-1));

} /* factorial */
```

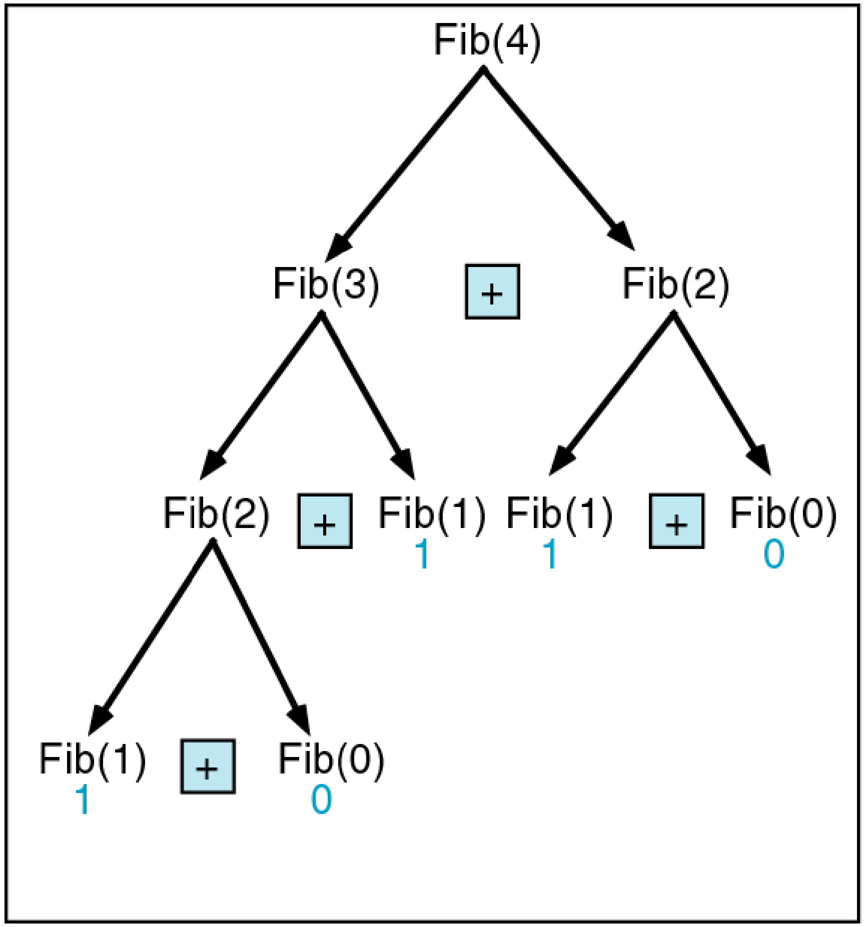
Recursive Functions (Example)

- Fibonacci series: 0, 1, 1, 2, 3, 5, 8...
  - Each number sum of two previous ones
  - Example of a recursive formula:

```
fib(n) = fib(n-1) + fib(n-2)
```


Fib n
Fib n-1
+
Fib n-2
Fib n-2
+
Fib n-3
Fib n-3
+
Fib n-4
Fib n-3
+
Fib n-4

(a) Fib(n)


Fib(4)
Fib(3)
+
Fib(2)
Fib(2)
+
Fib(1)
1
Fib(1)
1
+
Fib(0)
0
Fib(1)
1
+
Fib(0)
0

(b) Fib(4)

Recursive Functions (Example)

Sample code for **`fibonacci`** function

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

```
#include<stdio.h>

void print_value(int n);

void main(void)
{
        print_value(5);
        return;
}

void print_value(int n)
{
        if (n < 1)
                return;
        print_value(n-1);
        printf("%d\n",n);
}
```

```
#include<stdio.h>

int add_recursive(int n);

void main(void)
{

        int value;
        value = add_recursive(10);
        printf("Value is %d\n",value);
        return;
}

int add_recursive(int n)
{
        if(n == 1)
                return(1);
        else
                return(n+add_recursive(n-1));
}
```

Thank you