CSE1142 – Functions in C

Sanem Arslan Yılmaz

Some of the slides are from:
CMPE150 – Boğaziçi University
Deitel & Associates

Agenda

- Functions
  - Definition
  - Syntax
  - Input/output
- Function Prototypes
- Function Call Stack
- Scope and lifetime of variables
- Local vs. global variables
- Value parameters (call by value)
- Variable parameters (call by reference)
- Recursive Functions
- Macro Substitution
- Standart Functions in C

Functions - Definition

- A function groups a set of related statements under a single title.
- You can "call" the function using its name.
- You may also provide parameters to the function during the call.
- A function performs the actions defined by its statements and returns a result.

Functions - Syntax

- Syntax:

```
return_type function_name (parameter_list)
{  local_variable_definitions
   statement(s)
  return return_value
}
```

- Functions operate on their parameters and produce results.
  - The result is returned via the return value.
- The parameters may or may not be altered by the function.
- If *return_type* is missing, it means int.

Example

- A function that takes three parameters and returns their sum.

```
int add3(int a, int b, int c){
        int retval;
        retval = a + b + c;
        return retval;
}
```

- Or more succinctly

```
int add3(int a, int b, int c){
        return a + b + c;
}
```

Example

Consider the polynomial $P(x)=8x^5+5x^4+6x^3+3x^2+4x+2$

Read a value for `x` and display the result.

Example: Solution with functions

```
#include <stdio.h>
/* Calculate a^b */
float power(float a, int b){
   float result=1;
   for (; b>0; b--)
      result *= a;
   return result;
}
int main(){
   float P_x, x;
   scanf("%f", &x);
   P_x = 8 * power(x,5) +
         5 * power(x,4) +
         6 * power(x,3) +
         3 * power(x,2) +
         4 * x + 2;
   printf("Result=%f\n", P_x);
   return 0;
}
```

Functions – Caller and Callee

- A function that is being called is named callee.
- The function from which the call is made is named caller.
- In the previous example, `main()` is the caller, `power()` is the callee.

Example: Solution with functions II

```
#include <stdio.h>
float power(float a, int b);
int main(){
   float P_x, x;
   scanf("%f", &x);
   P_x = 8 * power(x,5) +
         5 * power(x,4) +
         6 * power(x,3) +
         3 * power(x,2) +
         4 * x + 2;
   printf("Result=%f\n", P_x);
   return 0;
}
/* Calculate a^b */
float power(float a, int b){
   float result=1;
   for (; b>0; b--)
      result *= a;
   return result;
}
```

Function prototype

Function Prototypes

- An important feature of C is the function prototype.
- The compiler uses function prototypes to validate function calls.
- The following prototype states that power takes two arguments of type **float** and **int** and returns a result of type **float**.
  - `float power(float a, int b);`
- Notice that the function prototype is the same as the first line of power’s function definition.
- Include parameter names in function prototypes for documentation purposes.
  - The compiler ignores these names, so the prototype `float power(float, int);` is also valid.
  - Forgetting the semicolon at the end of a function prototype is a syntax error.

Function Prototypes (cont.)

- A function call that does not match the function prototype is a compilation error.
- If there is no function prototype for a function before its use in main function, it leads to warnings or errors, depending on the compiler.
  - Since C is a single-pass language.
  - The compiler does not look ahead to see the definition of a function or variable.
  - The declaration of a function must come before the use of the function, otherwise the compiler does not know what its type signature is.

Functions – Input & Outputs

- We will cover functions in the following order:
  - Void functions
  - Functions without parameters
  - Functions with parameters
  - Functions that return a value
  - Functions that alter their parameters

Example: power() function with parameters

```
float power(float a, int b)
{
   float result=1;

   for (; b>0; b--)
      result *= a;
   return result;
}
```

Function Call Stack

- To understand how C performs function calls, we first need to consider a data structure (i.e., collection of related data items) known as a stack.
- Think of a stack as analogous to a pile of dishes.
- When a dish is placed on the pile, it’s normally placed at the top (referred to as pushing the dish onto the stack).
- Similarly, when a dish is removed from the pile, it’s normally removed from the top (referred to as popping the dish off the stack).
- Stacks are known as last-in, first-out (LIFO) data structures—the *last* item pushed (inserted) on the stack is the *first* item popped (removed) from the stack.

Function Call Stack (cont.)

- An important mechanism for computer science students to understand is the function call stack (sometimes referred to as the program execution stack).
- It supports the function call/return mechanism.
- Each time a function calls another function, an entry is *pushed* onto the stack.
- If a function makes a call to another function, a stack frame for the new function call is simply pushed onto the call stack.
- If the called function returns, instead of calling another function before returning, the stack frame for the function call is popped.

Function Call Stack (cont.)

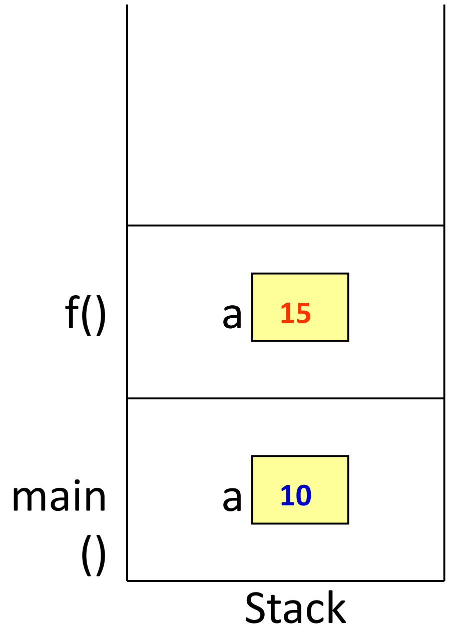
- The called function's stack frame is a perfect place to reserve the memory for *local variables*.
- That stack frame exists only as long as the called function is active.
- When that function returns, its stack frame is *popped* from the stack, and those local variables are no longer known to the program.
- Of course, the amount of memory in a computer is finite, so only a certain amount of memory can be used to store stack frames on the function call stack.
- If more function calls occur than can have their stack frames stored on the function call stack, a *fatal* error known as a stack overflow occurs.

Changes in parameters are not reflected

```
#include <stdio.h>

void f(int a)
{
   a+=5;
   printf("in function f(): a=%d\n", a);
}

int main()
{  int a=10;
   printf("in main(), before calling f(): a=%d\n",a);
   f(a);
   printf("in main(), after calling f(): a=%d\n",a);
}
```

OUTPUT

```
in main(), before calling f(): a=10
in function f(): a=15
in main(), after calling f(): a=10
```



Scope of variables

- A local variable can be used only in the function where it is defined
  - i.e., the scope of a local variable is the current function.

- A global variable can be used in all functions below its definition
  - i.e., the scope of a global variable is the range from the variable definition to the end of the file.

Lifetime of variables

- The lifetime of a variable depends on its scope.

- A local variable is alive as long as the function where it is defined is active.
  - When the function terminates, all of the local variables (and their values) are lost.
  - When the function is called again, all variables start from scratch; they don't continue with their values from the previous call (except for local static variables which are not discussed).

- The lifetime of a global variable is equivalent to the lifetime of the program.

Scope and lifetime of variables

```
#include <stdio.h>
int count;
int func(){
   int i, j;

   scanf("%d %d", &i, &j);
   count += i+j;
   return 0;
}

int main(){
   int i;

   func();
   for (i=0; i<count; i++)
      printf("%d",i);
   return 0;
}
```

Output

```
3
5
01234567
```

Global vs. local variables

Global variables	Local variables
Visible in all functions	Visible only within the function they are defined
Zero by default	Uninitialized
A change made by a function is visible everywhere in the program	Any changes made on the value are lost when the function terminates (since the variable is also removed)
Scope extends till the end of the file	Scope covers only the function
Lifetime spans the lifetime of the program	Lifetime ends with the function

Changing local variables

- Any change made on local variables is lost after the function terminates.

```
void f()
{   int a=10;
    a++;
    printf("in f(): a=%d\n",a);
}
int main()
{   int a=5;
    f();
    printf("After first call to f(): a=%d\n",a);
    f();
    printf("After second call to f(): a=%d\n",a);
}
```

Output

```
in f(): a=11
After first call to f(): a=5
in f(): a=11
After second call to f(): a=5
```

Changing global variables

- Any change made on global variables remains after the function terminates.

```
int b;
void f()
{
   b++;
   printf("in f(): b=%d\n",b);
}
int main()
{
   f();
   printf("After first call to f(): b=%d\n",b);
   f();
   printf("After second call to f(): b=%d\n",b);
}
```

Output

```
in f(): b=1
After first call to f(): b=1
in f(): b=2
After second call to f(): b=2
```

Changing value parameters

- Any change made on value parameters is lost after the function terminates.

```
void f(int c)
{
    c++;
    printf("in f(): c=%d\n",c);
}
int main()
{   int c=5;
    f(c);
    printf("After f(): c=%d\n",c);
}
```

Output

```
in f(): c=6
After f(): c=5
```

Local definition veils global definition

- A local variable with the same name as the global variable veils the global definition.

```
int d=10;
void f()
{
   d++;
   printf("in f(): d=%d\n",d);
}
int main()
{  int d=30;
   f();
   printf("After first call to f(): d=%d\n",d);
   f();
   printf("After second call to f(): d=%d\n",d);
}
```

Output

```
in f(): d=11
After first call to f(): d=30
in f(): d=12
After second call to f(): d=30
```

Parameter definition veils global definition

- Similar to local variables, parameter definition with the same name as the global variable also veils the global definition.

```
int e=10;
void f(int e)
{
   e++;
   printf("in f(): d=%d\n",e);
}
int main()
{  int g=30;
   f(g);
   printf("After first call to f(): g=%d\n",g);
}
```

Output

```
in f(): d=31
After first call to f(): g=30
```

Example

- Write a function that calculates the factorial of its parameter.

  n! = n(n-1)(n-2)...1

```
long factorial(int n)
{  int i; long res = 1;
   if (n == 0)
      return 1;
   for (i = 1; i <= n; i++)
      res *= i;
   return res;
}
```

Static variables

- Used when we want to retain the value of a local variable between calls.

```
#include <stdio.h>
void genie(void)
{
        static int wish = 0; /* value is kept in memory */
        wish++;
        if ( wish <= 3)
                printf("What is your %d. wish?\n", wish);
        else
                printf("No more wishes.\n");
}
int main()
{
        int i;
        for (i=1; i<=5; i++) /* call genie() five times */
                genie();
}
```

Output

```
What is your 1. wish?
What is your 2. wish?
What is your 3. wish?
No more wishes.
No more wishes.
```

Static variables

- Have permanent storage in memory.
- Lifetime is the lifetime of the program.
- However, they are local variables. The variable wish is not recognized in main()
- The declaration and initialization line

  static int wish=0;

  is used only in the first call of the function. Ignored in later calls.

Variable Parameters

Value parameters – Call by value

- We used "value parameters" up to now.

- We did not pass the variable in the argument; we passed the ***value*** of the expression in the argument.

  - That is why the changes made on the parameter were not reflected to the variable in the function call.

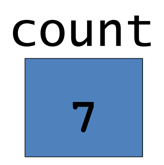
Variable parameters

- Sometimes, the caller may want to see the changes made on the parameters by the callee.
  - E.g.:
    ```
    int main()
    {   int num;
        scanf("%d", &num);
    }
    ```
  - You expect **`scanf()`** to "put" the input in the variable **`num`**, but this is not possible with value parameters.

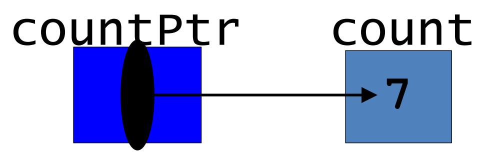
    (It is possible to return a single value using the return type; so it is not practical enough. Also, you should use the same variable both as an argument and for the return value.)

Pointer variable definitions and initialization

- Pointer variables
  - Contain memory addresses as their values
  - Normal variables contain a specific value (direct reference)



  - Pointers contain address of a variable that has a specific value (indirect reference)
  - Indirection – referencing a pointer value



Pointer variable definitions and initialization – cont.

- Pointer definitions
  - `*` used with pointer variables

    ```
    int *myPtr;
    ```

  - Defines a pointer to an `int` (pointer of type `int *`)
  - Multiple pointers require using a `*` before each variable definition

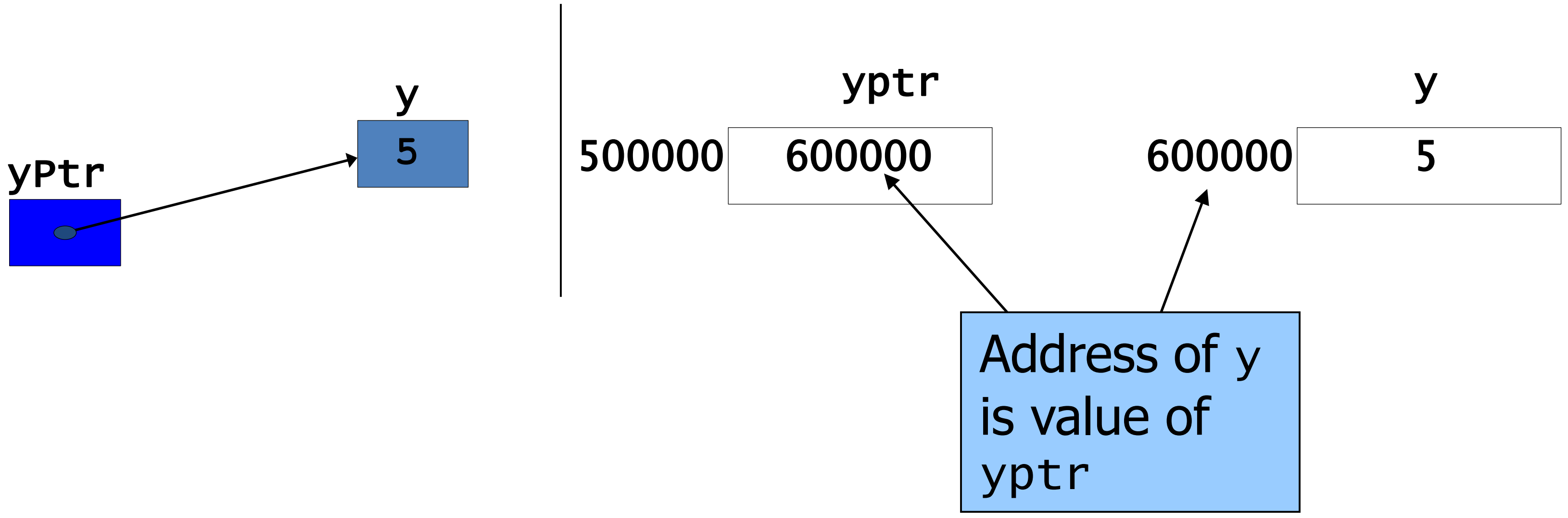
    ```
    int *myPtr1, *myPtr2;
    ```

  - Can define pointers to any data type
  - Initialize pointers to `0`, `NULL`, or an address
    - `0` or `NULL` – points to nothing (`NULL` preferred)

Pointer Operators

- & (address operator)
  - Returns address of operand

```
int y = 5;
int *yPtr;
yPtr = &y;     /* yPtr gets address of y */
yPtr “points to” y
```



Pointer Operators – cont.

- * (indirection/dereferencing operator)
  - Returns a synonym/alias of what its operand points to
  - `*yptr` returns `y` (because `yptr` points to `y`)
  - * can be used for assignment
    - Returns alias to an object

      ```
      *yptr = 7;  /* changes y to 7 */
      ```

- * and & are inverses
  - They cancel each other out

Variable parameters

- Converting a value parameter to a variable parameter is easy.

```
void func(int *num)
{   *num = 5;
}
int main()
{   int count=10;
    func(&count);
}
```

Define the parameter as pointer

Use '*' before the parameter name so that you access the value at the mentioned address

Send the address of the argument

Call by reference

- The idea is very simple actually:
  - When a function terminates everything inside the stack entry for that function is erased.

  - Therefore, if you want the changes to remain **after** the function call, you should make the change outside the function's stack entry.

  - Here is what you do:
    - At the caller instead of sending the value, send the reference (address), i.e. a pointer.
    - At the callee, receive the address (i.e., define the parameter as a pointer).
    - Inside the callee, use a de-referencer ('*') with the parameter name since the parameter contains the address, not the value.

Call by reference

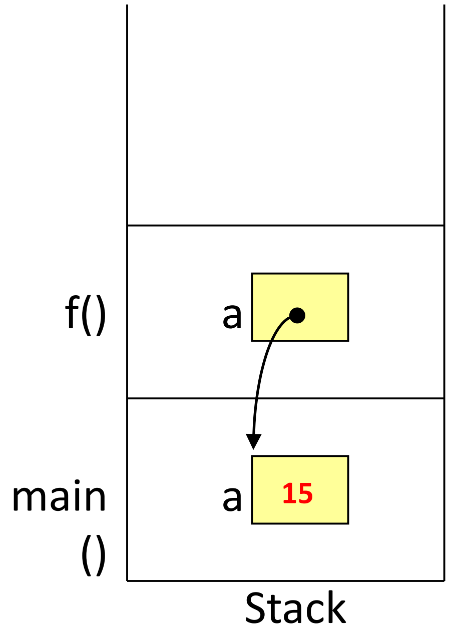
- This is why:
  - "using value parameters" is also called "call by value‟
  - "using variable parameters" is also called "call by reference"

Variable parameters

```
#include <stdio.h>

void f(int *a)
{
   *a+=5;
   printf("in function f(): a=%d\n", *a);
}

int main()
{  int a=10;
   printf("in main(), before calling f(): a=%d\n",a);
   f(&a);
   printf("in main(), after calling f(): a=%d\n",a);
}
```

OUTPUT

```
in main(), before calling f(): a=10
in function f(): a=15
in main(), after calling f(): a=15
```



Example

- Write a function that exchanges its parameters.
- Solution 1:

```
void swap(int a, int b)
{
   a=b;
   b=a;
}
```

WRONG!

Example

- Solution 2:

```
void swap(int *a, int *b)
{
   *a=*b;
   *b=*a;
}
STILL WRONG!
```

Example

- Solution 3:

```
void swap(int *a, int *b)
{
   int temp;
   temp = *a;
   *a=*b;
   *b=temp;
}
```

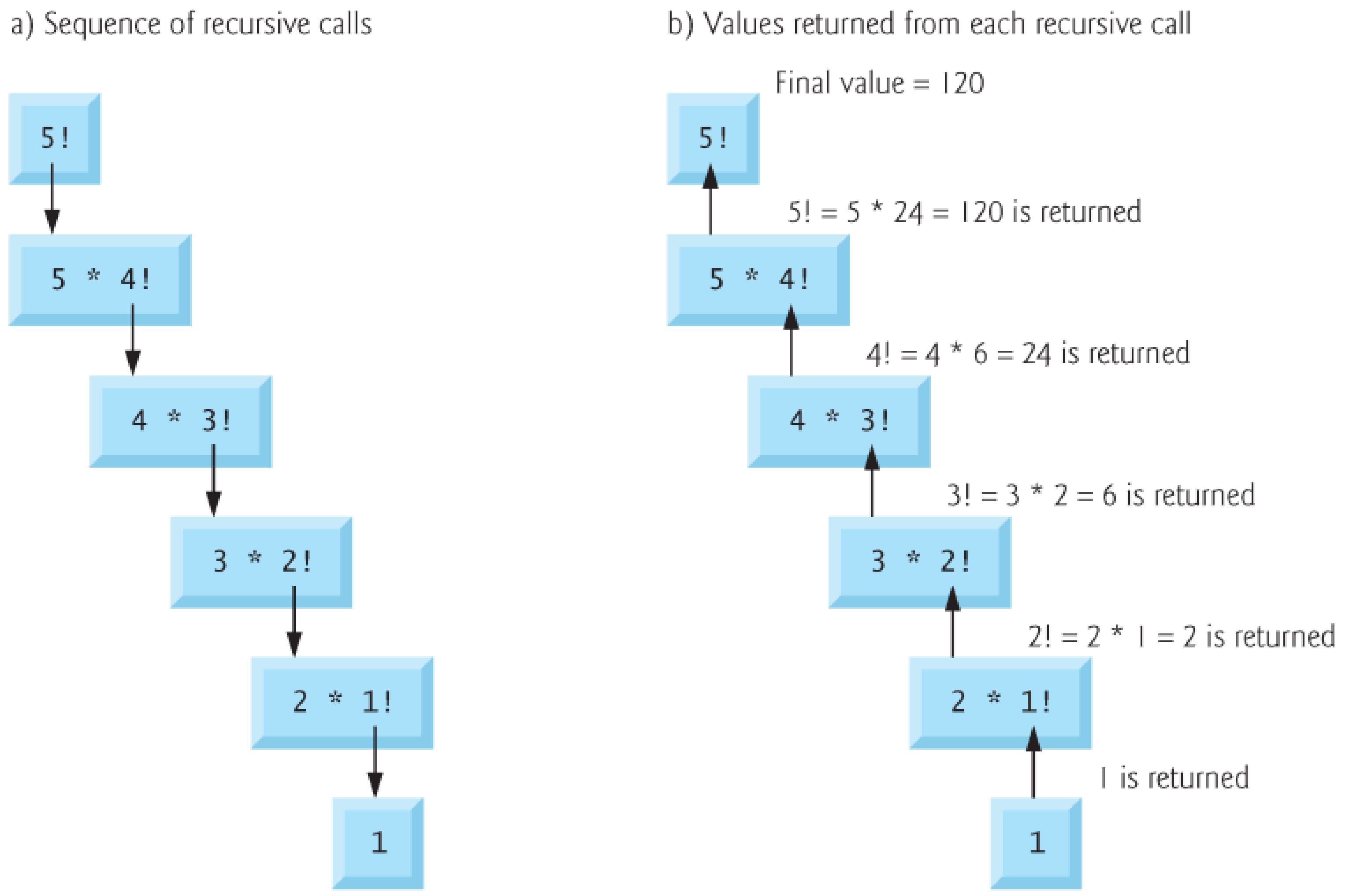
Recursion

- Recursive functions
  - Functions that call themselves
  - Can only solve a base case
  - Divide a problem up into
    - What it can do
    - What it cannot do
      - What it cannot do resembles original problem
      - The function launches a new copy of itself (recursion step) to solve what it cannot do
  - Eventually base case gets solved
    - Gets plugged in, works its way up and solves whole problem

Recursion – Example

- Example: factorials
  - `5! = 5 * 4 * 3 * 2 * 1`
  - Notice that
    - `5! = 5 * 4!`
    - `4! = 4 * 3! ...`
  - 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 – Sequence of calls and returns



a) Sequence of recursive calls
5!
5 * 4!
4 * 3!
3 * 2!
2 * 1!
1
b) Values returned from each recursive call
Final value = 120
5!
5! = 5 * 24 = 120 is returned
5 * 4!
4! = 4 * 6 = 24 is returned
4 * 3!
3! = 3 * 2 = 6 is returned
3 * 2!
2! = 2 * 1 = 2 is returned
2 * 1!
1 is returned
1

Recursion – Factorial code

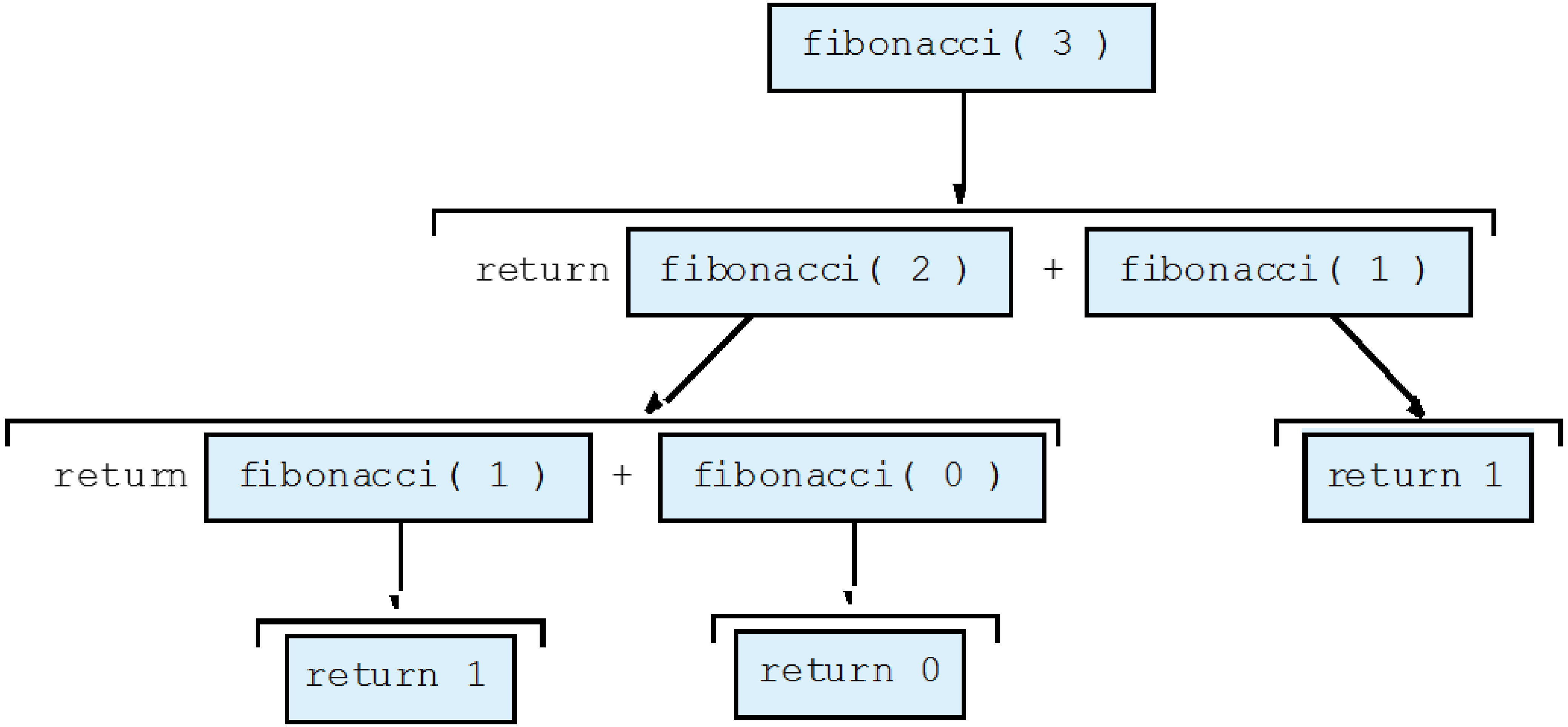
```
#include <stdio.h>
long factorial(int n)
{
  if (n>0)
    return n*factorial(n-1);
  else if (n==0)
    return 1;
  return 0;
}

int main()
{
  int n;
  printf("Enter an integer: ");
  scanf("%d", &n);
  printf("%d! = %ld\n", n, factorial(n));
  return 0;
}
```

Example Using Recursion: The Fibonacci Series

- Fibonacci series: 0, 1, 1, 2, 3, 5, 8...
  - Each number is the sum of the previous two
  - Can be solved recursively:
    - `fib( n ) = fib( n - 1 ) + fib( n – 2 )`
  - Code for the `fibonacci` function

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

Set of recursive calls to function fibonacci



Recursion vs. Iteration

- Repetition
  - Iteration: explicit loop
  - Recursion: repeated function calls

- Termination
  - Iteration: loop condition fails
  - Recursion: base case recognized

- Both can have infinite loops

- Balance
  - Choice between performance (iteration) and good software engineering (recursion)

Preprocessor macros

Macro Substitution

- Remember

  ```
  #define PI 3.14
  ```

- It is also possible to write macros with parameters.

  ```
  #define square(x) x*x
  ```

  - The name of the macro is `square`.
  - Its parameter is `x`.
  - It is replaced with `x*x` (with paramater `x` substituted.)

Defining Macros with Parameters

- macro
  - facility for naming a commonly used statement or operation

```
#define macro_name(parameter list) macro body
```

- macro expansion
  - process of replacing a macro call by its meaning

Macro Substitution

- `#define square(x) x*x`

- Note that when the macro is called as

  `square(a+1)`

  the substituted form will be

  `a+1*a+1`

  which is not correct.

- The macro should have been defined as

  `#define square(x) (x)*(x)`

  so that its substituted form would be

  `(a+1)*(a+1)`

Macro Substitution

- A macro is NOT a function.
  - A macro is implemented as a substitution.
    - ▪ The code segment that implements the macro substitutes the macro call.
    - + Code executes faster.
    - - Code becomes larger.
    - + It is possible to do things that you cannot do with functions.
    - - Limited syntax check (Remember the `"square(a+1)"` example).
  - A function is implemented by performing a jump to a code segment.
    - ▪ Stack operations are performed for function call.
    - - Code executes slower.
    - + Code is smaller.
    - + More structured programming.
    - + Better syntax check.

Example: Macro substitution

- Define a macro for finding the maximum of two values.

```
#define max(A,B) (((A)>(B))?(A):(B))
```

- Find the maximum of three numbers. Four numbers?

```
result=maximum(a, b, c);

int maximum(int x, int y, int z){
    int max_num=x;
    if(max_num < y)
        max_num=y;
    if(max_num < z)
        max_num=z;
    return max_num;

}
```

```
temp=max(a,b);
result=max(temp,c);


temp1=max(a,b);
temp2=max(c,d);
result=max(temp1,temp2);
```

Example: Macro substitution

- Define a macro for finding the absolute value.

```
#define abs(A)  ((A)>0)?(A):-(A))
```

C libraries

Standart Functions in C

- Math library functions
- Header files
- Random number generation

Math Library Functions

- Math library functions
  - perform common mathematical calculations
  - `#include <math.h>`

- Format for calling functions
  - `FunctionName(` *`argument`* `);`
    - If multiple arguments, use comma-separated list
  - `printf( "%.2f", sqrt( 900.0 ) );`
    - Calls function `sqrt`, which returns the square root of its argument
    - All math functions return data type `double`
  - Arguments may be constants, variables, or expressions

Math Library Functions

Function	Description	Example

`sqrt(x)`	square root of *x*	`sqrt(900.0)` is `30.0`
		`sqrt(9.0)` is `3.0`

`cbrt(x)`	cube root of *x* (C99 and C11 only)	`cbrt(27.0)` is `3.0`
		`cbrt(-8.0)` is `-2.0`

`exp(x)`	exponential function e^x	`exp(1.0)` is `2.718282`
		`exp(2.0)` is `7.389056`

`log(x)`	natural logarithm of *x* (base *e*)	`log(2.718282)` is `1.0`
		`log(7.389056)` is `2.0`

`log10(x)`	logarithm of *x* (base 10)	`log10(1.0)` is `0.0`
		`log10(10.0)` is `1.0`
		`log10(100.0)` is `2.0`

`fabs(x)`	absolute value of *x* as a floating-point number	`fabs(13.5)` is `13.5`
		`fabs(0.0)` is `0.0`
		`fabs(-13.5)` is `13.5`

`ceil(x)`	rounds *x* to the smallest integer not less than *x*	`ceil(9.2)` is `10.0`
		`ceil(-9.8)` is `-9.0`

Math Library Functions (cont.)

Function	Description	Example

`floor(x)`	rounds x to the largest integer not greater than x	`floor(9.2)` is `9.0`
		`floor(-9.8)` is `-10.0`

`pow(x, y)`	x raised to power y (x^y)	`pow(2, 7)` is `128.0`
		`pow(9, .5)` is `3.0`

`fmod(x, y)`	remainder of x/y as a floating-point number	`fmod(13.657, 2.333)` is `1.992`
`sin(x)`	trigonometric sine of x (x in radians)	`sin(0.0)` is `0.0`
`cos(x)`	trigonometric cosine of x (x in radians)	`cos(0.0)` is `1.0`
`tan(x)`	trigonometric tangent of x (x in radians)	`tan(0.0)` is `0.0`

Header Files

- Header files
  - Contain function prototypes for library functions
  - `<stdlib.h>` , `<math.h>` , etc
  - Load with `#include <filename>`

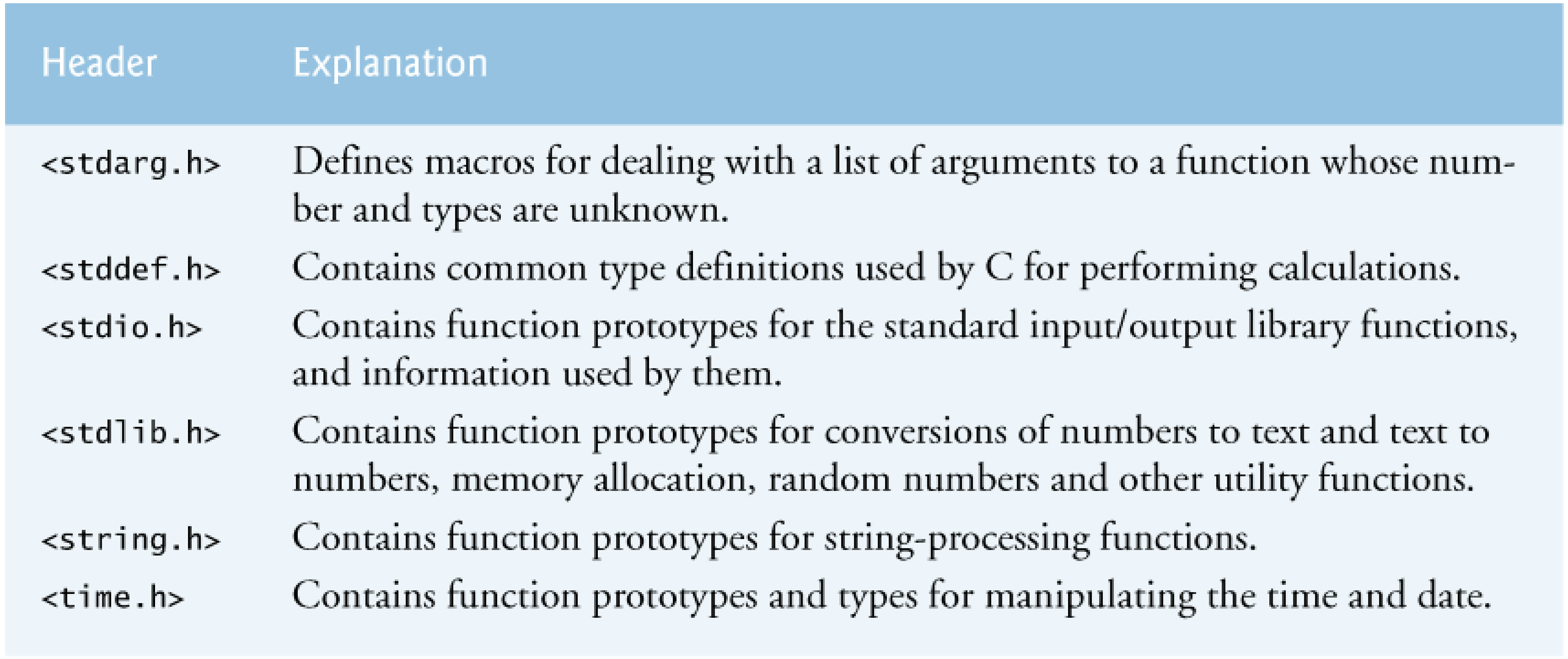
    `#include <math.h>`

- Custom header files
  - Create file with functions
  - Save as `filename.h`
  - Load in other files with `#include "filename.h"`
  - Reuse functions

Some of the standard library headers

Header	Explanation
`<assert.h>`	Contains information for adding diagnostics that aid program debugging.
`<ctype.h>`	Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa.
`<errno.h>`	Defines macros that are useful for reporting error conditions.
`<float.h>`	Contains the floating-point size limits of the system.
`<limits.h>`	Contains the integral size limits of the system.
`<locale.h>`	Contains function prototypes and other information that enables a program to be modified for the current locale on which it's running. The notion of locale enables the computer system to handle different conventions for expressing data such as dates, times, currency amounts and large numbers throughout the world.
`<math.h>`	Contains function prototypes for math library functions.
`<setjmp.h>`	Contains function prototypes for functions that allow bypassing of the usual function call and return sequence.
`<signal.h>`	Contains function prototypes and macros to handle various conditions that may arise during program execution.

Some of the standard library headers (cont.)



Header	Explanation
`<stdarg.h>`	Defines macros for dealing with a list of arguments to a function whose number and types are unknown.
`<stddef.h>`	Contains common type definitions used by C for performing calculations.
`<stdio.h>`	Contains function prototypes for the standard input/output library functions, and information used by them.
`<stdlib.h>`	Contains function prototypes for conversions of numbers to text and text to numbers, memory allocation, random numbers and other utility functions.
`<string.h>`	Contains function prototypes for string-processing functions.
`<time.h>`	Contains function prototypes and types for manipulating the time and date.

Random Number Generation – rand function

- `rand` function
  - Load `<stdlib.h>`
  - Returns "random" number between `0` and `RAND_MAX` (at least `32767`)

    ```
    i = rand();
    ```
  - Pseudorandom
    - Preset sequence of "random" numbers
    - Same sequence for every function call

- Scaling
  - To get a random number between `1` and `n`

    ```
    1 + ( rand() % n )
    ```
    - **`rand() % n`** returns a number between `0` and `n - 1`
    - Add `1` to make random number between `1` and `n`

      ```
      1 + ( rand() % 6)
      ```
      - number between `1` and `6`

Random Number Generation – srand function

- `srand` function
  - `<stdlib.h>`
  - Takes an integer seed and jumps to that location in its "random" sequence

    `srand(` *seed* **`);`**
  - `srand( time( NULL ) );/*  include <time.h> */`
    - `time( NULL )`
      - Returns the time at which the program was compiled in seconds
      - "Randomizes" the seed