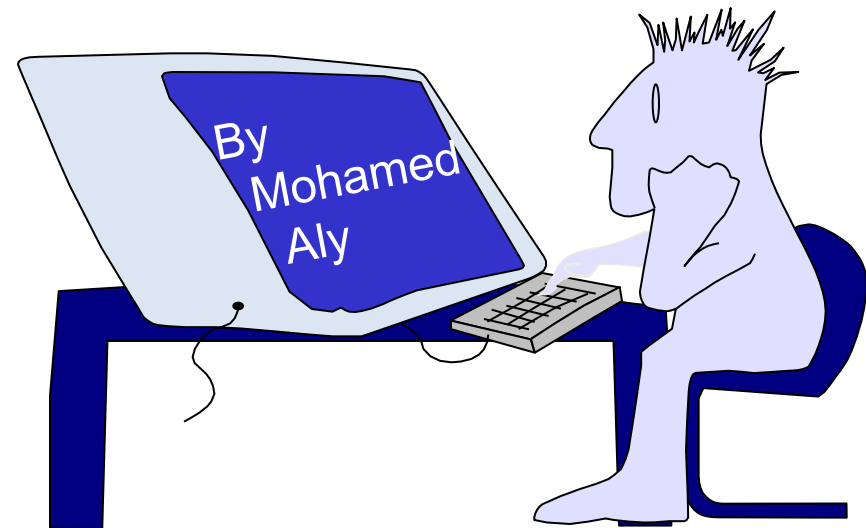


Embedded C

C Programming Part 1



Compilation

- C compilers take C and convert it into an architecture specific machine code (string of 1s and 0s)
- Unlike other languages which convert to architecture independent code

Advantages

- Great run-time performance
 - Generally much faster than other languages for comparable code (because it optimizes for a given architecture)

Compilation(Cont.)

Disadvantages

- All compiled files (including the executable) are architecture specific, depending on both the CPU type and the operating system.
- Executable must be rebuilt on each new system.
 - Called “porting your code” to a new architecture
- The “change → compile → run [repeat]” iteration cycle is slow

C program

- C programs consist of pieces/modules called functions, exactly one of which must be main
 - A programmer can create his own functions
 - Advantage: the programmer knows exactly how it works
 - Disadvantage: time consuming
 - Programmers will often use the C library functions
 - Use these as building blocks
 - Avoid re-inventing the wheel
 - If a pre-made function exists, generally best to use it rather than write your own
 - Library functions carefully written, efficient, and portable

First program in C

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

```
/* function main begins program execution */
```

```
int main( void )
```

```
{
```

```
    printf( "Welcome to C! \n" );
```

```
    return 0; /* indicate that program ended successfully */
```

```
} /* end function main */
```

Comments

- Text surrounded by `/*` and `*/` is ignored by compiler

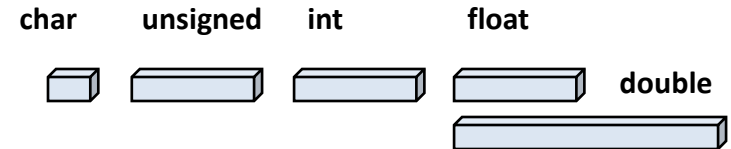
Variable declarations

- All variable declarations must go before they are used (at the beginning of the block)
- A variable may be initialized in its declaration; if not, it holds garbage
- Examples of declarations:
 - **correct:**

```
{  
    int a = 0, b = 10;  
    ...  
}
```
 - **Incorrect:**

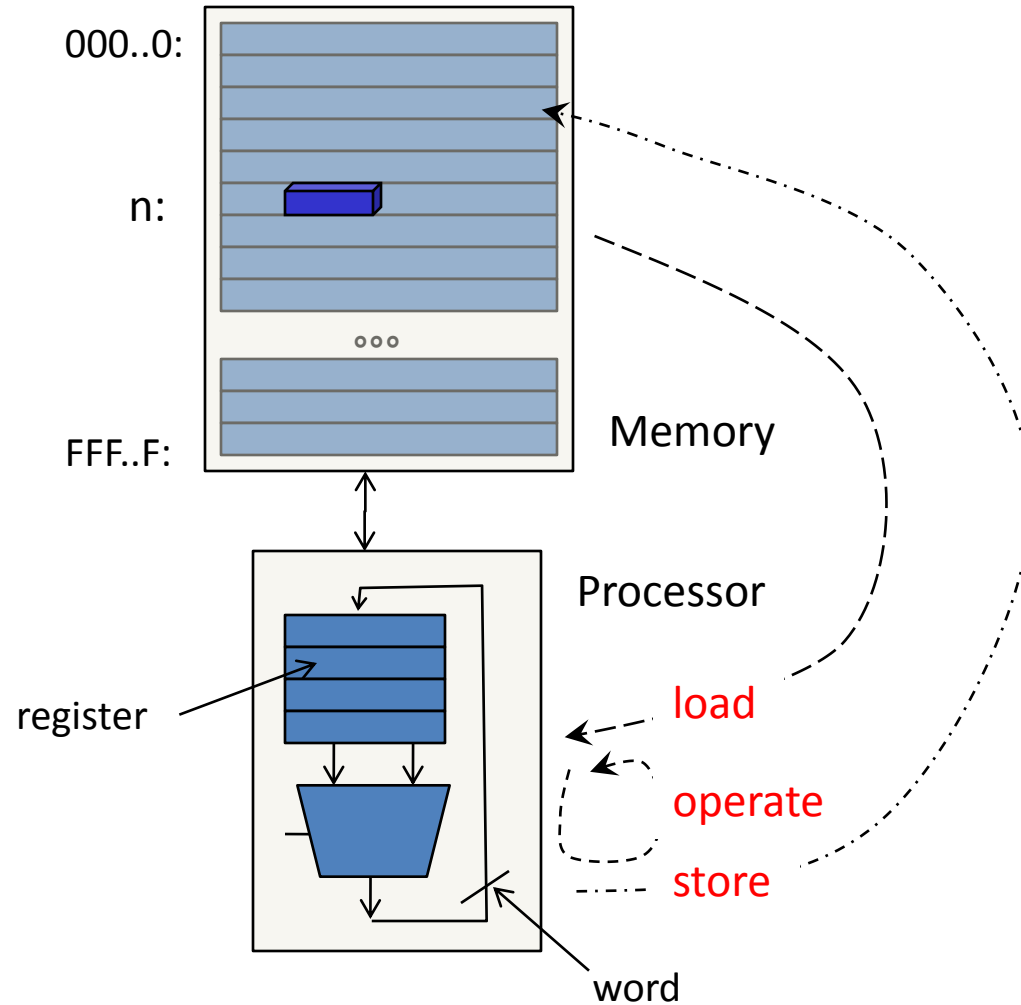
```
for (int i = 0; i < 10; i++)
```


Data/Variable types



- **char** – 1 byte
 - Sufficient to represent the local character set
 - Typically ASCII
- **int** – signed integer values
 - Range dictated by natural word width of the machine
- **float** – single precision floating point number
 - A lot like real numbers
 - Specific representation defined by IEEE
- **double** – double precision floating point number
 - Even more like real numbers

Where do variables live and work?



#include

- Preprocessor directive
- Tells computer to load contents of a certain file
- **#include <stdio.h>**
 - <stdio.h> allows standard input/output operations

printf

- **printf("Welcome to C \n");**
- Instructs computer to perform an action
 - Specifically, prints the string of characters within quotes (" ")
- Entire line called a statement
 - All statements must end with a semicolon ;
- Escape character (\)
 - Indicates that printf should do something out of the ordinary
 - \n is the newline character

```
printf( "Welcome " );  
printf( "to C \n" );
```

\n :Newline. Position the cursor at the beginning of the next line.

\t : Horizontal tab. Move the cursor to the next tab stop.

\a : Alert. Sound the system bell.

**** :Backslash. Insert a backslash character in a string.

\" :Double quote. Insert a double-quote character in a string.

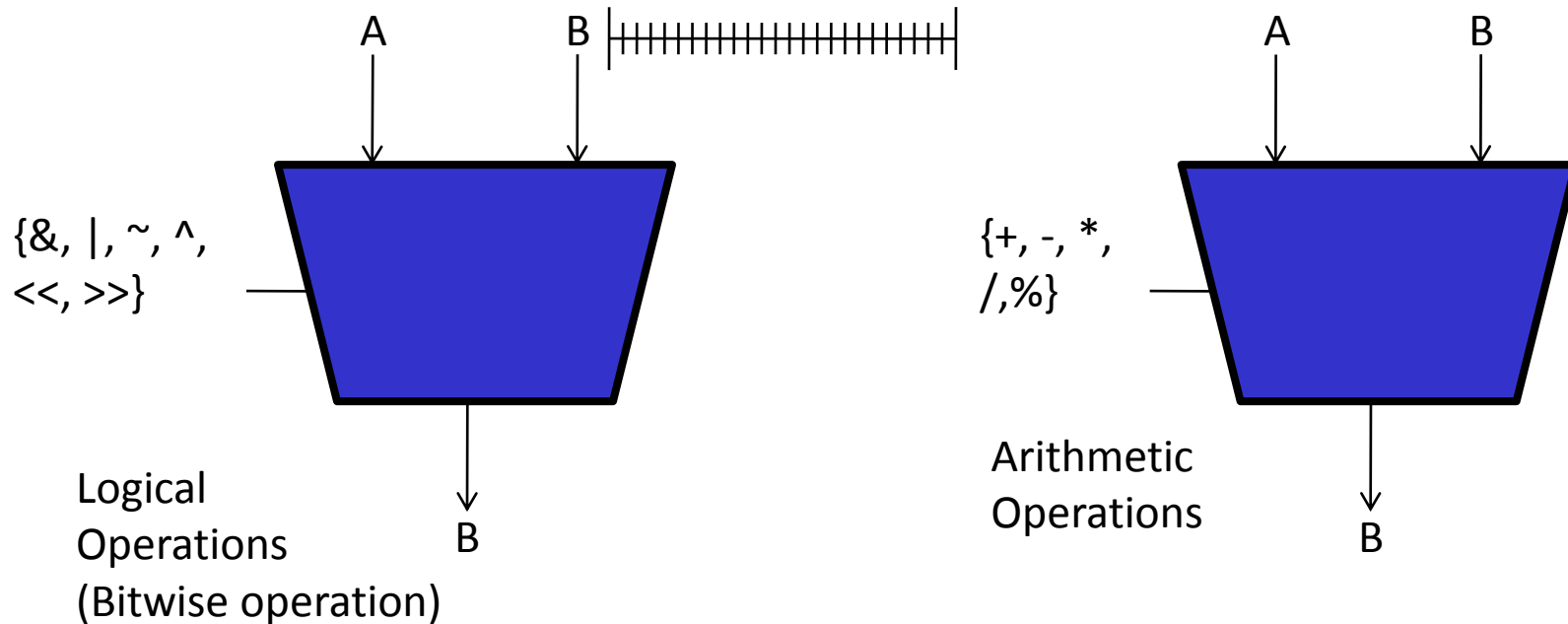
scanf

- **scanf("%d", &integer);**
- Obtains a value from the user
 - **scanf** uses standard input (usually keyboard)
- This **scanf** statement has two arguments
 - **%d** indicates data should be a decimal integer
 - **&integer** location in memory to store variable
 - **&** is confusing in beginning
 - For now, just remember to include it with the variable name in **scanf** statements
- When executing the program the user responds to the **scanf** statement by typing in a number, then pressing the *enter* key

```
int integer;  
printf( "Enter first integer\n" );  
scanf( "%d", &integer );
```

Bitwise and arithmetic operators

- Computers provide direct hardware support for manipulating certain basic types



- An attempt to divide by zero is normally undefined on computer systems and generally results in a fatal error
- Bitwise operation are machine dependent
 - How will it deal with right shift for signed numbers?

Assignment operators

- $c = c + 3;$
- Destructive *read-in* and non-destructive *read-out*
- Assignment operators abbreviate assignment expressions
 $c = c + 3;$
 $c += 3;$
- Examples of other assignment operators:
 - $d -= 4$ ($d = d - 4$)
 - $e *= 5$ ($e = e * 5$)
 - $f /= 3$ ($f = f / 3$)
 - $g \% = 9$ ($g = g \% 9$)

Increment and decrement operators

- Increment operator (**++**)
 - Can be used instead of **c+=1**
- Decrement operator (**--**)
 - Can be used instead of **c-=1**
- Preincrement
 - Operator is used before the variable (**++c** or **--c**)
 - Variable is changed before the expression it is in is evaluated
- Postincrement
 - Operator is used after the variable (**c++** or **c--**)
 - Expression executes before the variable is changed

Increment and decrement operators(Cont.)

- If c equals 5

```
printf( "%d", ++c ); /*Prints 6*/  
printf( "%d", c++ ); /*Prints 5*/  
/*In either case, c now has the value of 6*/
```

```
++c;  
printf( "%d", c );
```

```
c++;  
printf( "%d", c );
```

```
/*In either case, c now has the value of 6 and prints 6*/
```

Equality and relational operators

- ==(Equal)
- !=(Not equal)
- >(Greater than)
- >=(Greater than or equal)
- <(Less than)
- <=(Less than or equal)

Do not mix

- Do not be confused equality (==) and assignment (=) operators

Logical operators

- **&&** (logical AND)
 - Returns true if both conditions are true
- **||** (logical OR)
 - Returns true if either of its conditions are true
- **!** (logical NOT, logical negation)
 - Reverses the truth/falsity of its condition
 - Unary operator, has one operand
- What evaluates to FALSE in C?
 - 0 (integer)
 - **NULL** (pointer: more on this later)
- What evaluates to TRUE in C?
 - everything else...

Conditional operator (?:)

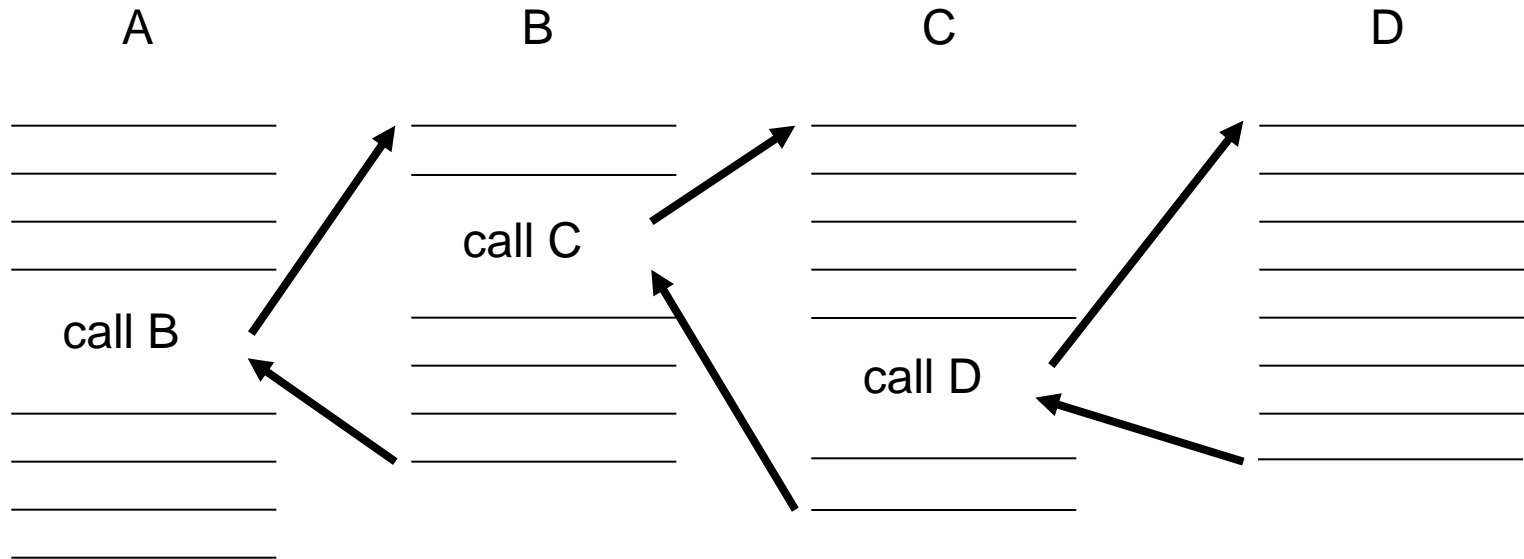
```
printf( "%s\n", grade >= 50 ? "Passed" : "Failed" );
```

```
grade >= 50 ? printf("Passed\n") : printf("Failed \n");
```

Operators	Associativity	Type
() [] . ->	left to right	highest
- ++ -- ! & * ~ size of (type)	right to left	unary
+ * / %	left to right	multiplicative
+ -	left to right	additive
<< >>	left to right	shifting
< <= > >=	left to right	relational
!=	left to right	equality
==		
&	left to right	bitwise AND

Operator precedence and associativity. (Part 1 of 2.)

Functions



Functions(Cont.)

```
return-value-type function-name( parameter-list )  
{  
    declarations and statements  
}
```

- Function-name: any valid identifier
- Return-value-type: data type of the result (default int)
 - void – indicates that the function returns nothing
- Parameter-list: comma separated list, declares parameters
 - A type must be listed explicitly for each parameter unless, the parameter is of type int
- Definitions and statements: function body (block)
 - Variables can be defined inside blocks (can be nested)
 - Functions can not be defined inside other functions
- Returning control
 - If nothing returned return; or, until reaches right brace
 - If something returned return *expression*

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

```
int x =10;  
Printf(“%0d”,  
        square( x ) );
```

Functions(Cont.)

Calling functions

- 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

```
int addone( int y )  
{  
    return y+1;  
}
```

```
void addone( int y )  
{  
    y = y+1;  
}
```

Functions(Cont.)

Function prototype

- Function name, Parameters, Return type
- Used to validate functions
- Prototype only needed if function definition comes after use in program

```
#include<stdio.h>
int square( int y );
int main( void )
{
    int x =10;
    Printf(“%d”, square( x ) );
    return 0;
}
int square( int y )
{
    return y * y;
}
```

Functions(Cont.)

Recursive functions

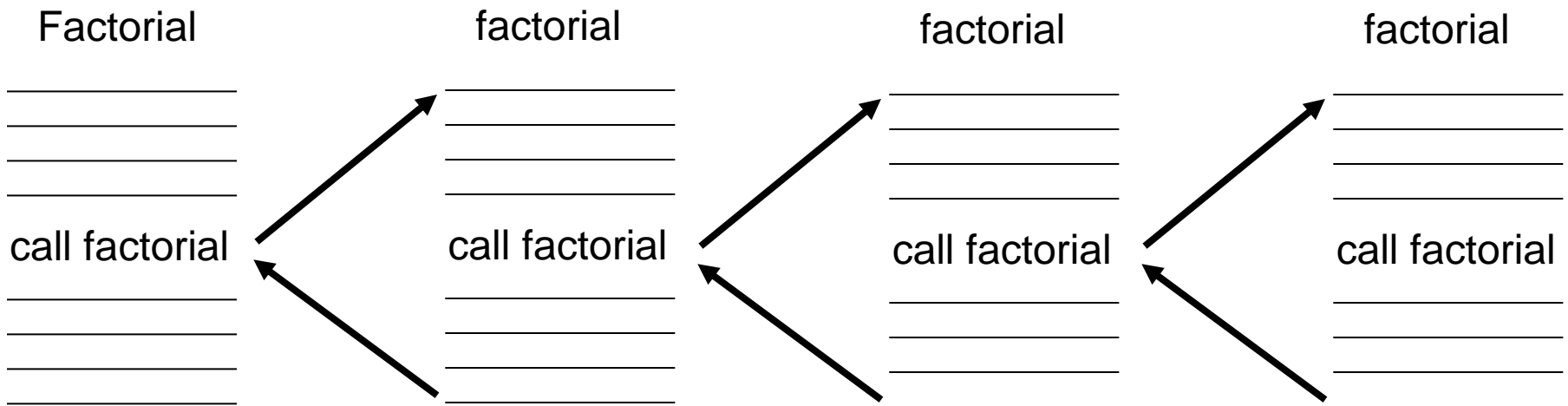
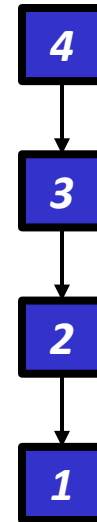
- Functions that call themselves
- Can only solve a base case
- Balance between performance (iterations/loops) and good coding style (recursion)

Functions(Cont.)

Recursive functions(Cont.)

```
long factorial( long number )
{
    if ( number <= 1 )
    {
        return 1;
    }
    else
    {
        return ( number * factorial( number - 1 ) );
    }
}
```

$$5! = 5 * 4 * 3 * 2 * 1 = 5 * (4!)$$



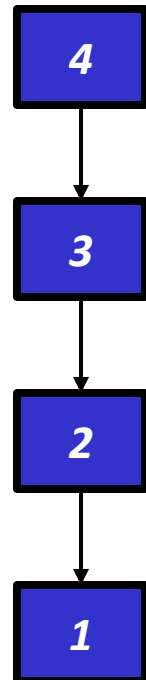
Functions(Cont.)

Recursive functions(Cont.)

Factorials

$$5! = 5 * 4 * 3 * 2 * 1 = 5 * (4!)$$

```
long factorial( long number )  
{  
    if ( number <= 1 )  
    {  
        return 1;  
    }  
    else  
    {  
        return ( number * factorial( number - 1 ) );  
    }  
}
```

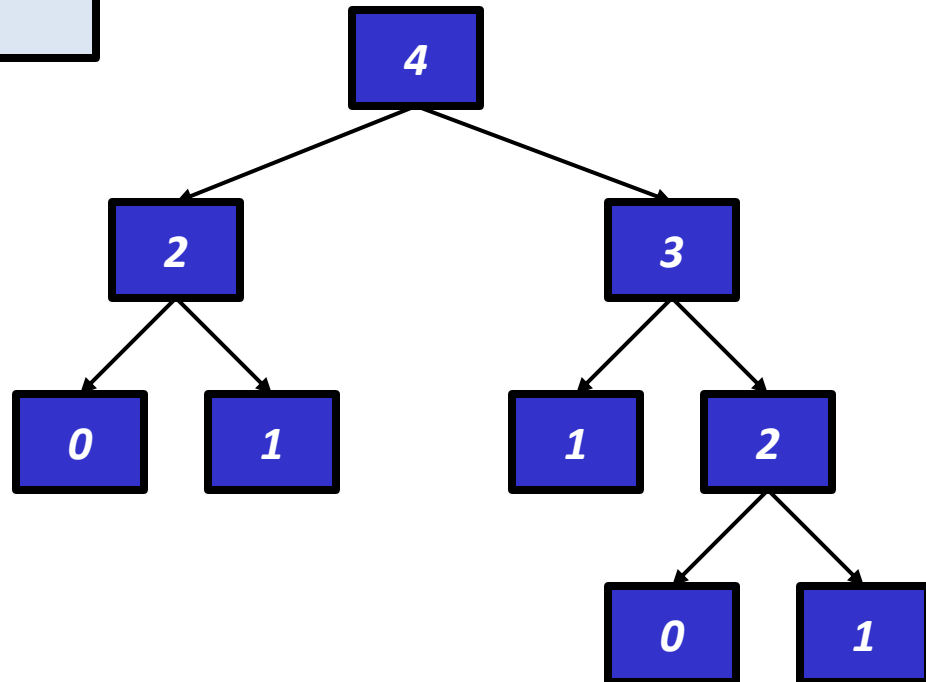


Functions(Cont.)

Recursive functions(Cont.)

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

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

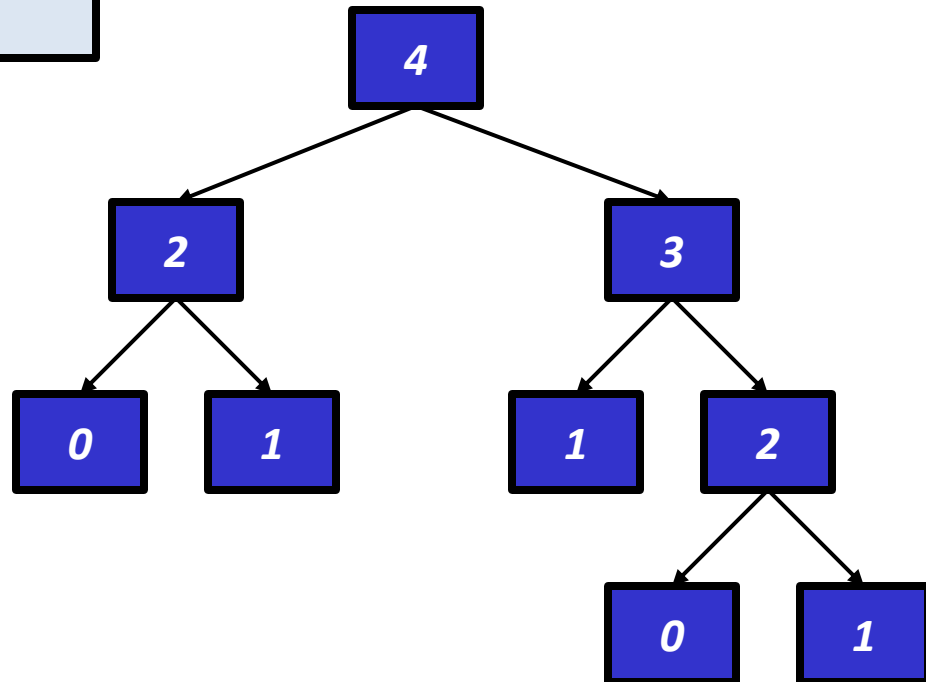


Functions(Cont.)

Recursive functions(Cont.)

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

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



Thanks

Embedded C