Structured Programming with C Wrap-up

Programming Paradigms

- Programming Paradigm : Defines how to think when you are programming. Way of approaching the task.
- Common Paradigms;
  - Imperative : Describe all the steps required to perform a task. Which variables to declare, what to assign etc. Describe how to do it!
  - Declarative : Specify the result you want, now how to do it. E.g. SQL for databases
  - Functional : Programming with function calls, avoid global state
  - Object-Oriented: Structure the program with Objects and relationships between these objects
- C is an Imperative programming language.

Computing with Pen & Paper

- Think of computing by a person using pen and paper:
  - Task: Check if there are same number of 1s and 0s in a given String (e.g. 1001010011110010100)
  - Go through the String till the end, for each 1, add a new mark to the paper
  - Go through the String till the end, for each 0, strike out the leftmost mark without a strike.
  - If we can't find a mark to strike-out for a 0 then there are more 0s
  - If after processing all zeros there are marks that are not striked-out, there are more 1s.
  - Otherwise there are same number of 1s and 0s

States and Operations

- When designing a C program (applies to imperative programming in general) we think of the problem as a set of variables, and operations changing them
- Just like pen & paper example;
  - Paper stands for our variables (marking & striking are assignment)
  - The procedure executed by the person corresponds to our program
- Data types: To simplify our job, we define datatypes
  - Instead of a mark we can store integers, doubles or more complex datatypes (structs)
  - Decide on what data to store, and what type they should be
- Functions, Objects :
  - Simplify the algorithm description; break down the task into subtasks.

Design of Program

- Given a problem;
  - Program flow : Roughly sketch the execution of the program, what are the decisions, loops and sequence of the operations.
  - Identify: Data types, data structures. Will using a struct make this task easier?
  - Top-down and/or Bottom-up design: How to structure the problem. (More on this later)
  - Coding: Use all tools provided by programming language to implement the idea. As you realize the problem, you can always change things in previous steps. Code changes design, design changes code (Refactoring).
  - Testing and Debugging : Will always have errors, establish habits and procedures to avoid them.

Example: Tic-Tac-Toe Specs (1)

The program is to print a welcome message to introduce the game and its rules.

```
Welcome to TIC-TAC-TOE.
-----------------------
The object of this game is to get a line of X’sbefore the computer gets a line of O’s. A line maybe across, down, or diagonal.The board is labelled from 1 to 9 as follows:
1|2|3
-----
4|5|6
-----
7|8|9
```

Tic-tac-toe Specs (2)

This is followed by a request for whether the user wishes to start, with the integer 1 meaning ‘yes’and 0 for ‘no’.

```
Do you wish to go first (1-Yes, 0-No) ?
```

Each time the user is to make a move, a request is made for which location he wishes to choose,designated by a number between 1 and 9.

```
Your turn (1 - 9):
```

After the user or the computer has made a decision, the resulting table is printed in ASCII. For example, after 5 turns, the board might look like

```
X| |

-----

X|O|

-----

|X|O
```

Tic-Tac-Toe Specs (3)

The game will terminate with a winner or a draw, and a comment is to be printed on account of theuser’s win, loss or draw, respectively.

44

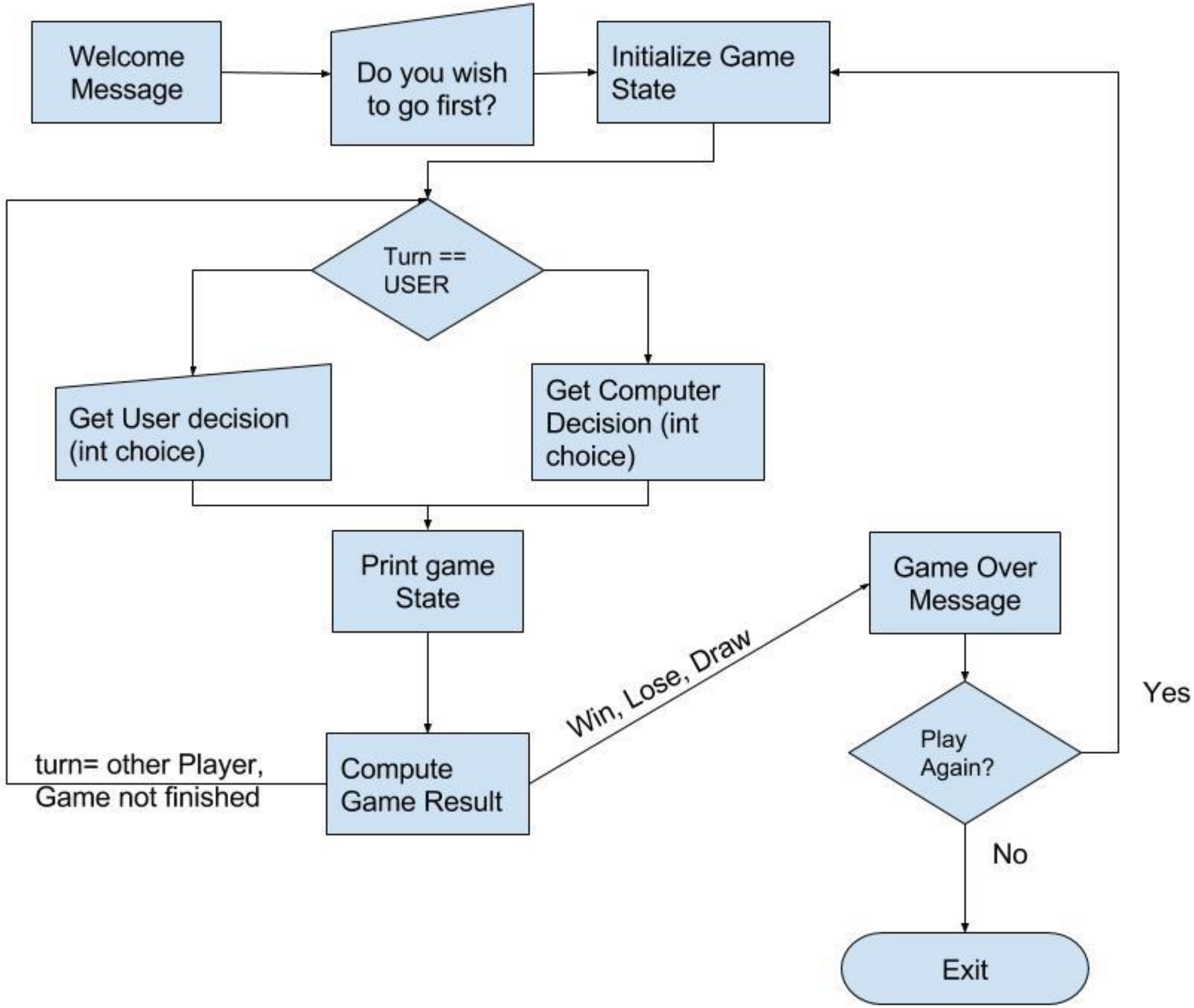
```
You win. Congratulations!!

You lose. Better luck next time.

Its a draw. How dull.
```

Finally, the user is asked if he wishes to play again and, if so, the game returns to the request ofwhether he wishes to have first go. Otherwise, the program terminates.

```
Do you wish to play again (1-Yes, 0-No) ?
```

Program Flow

- Model the flow of the program with respect to the specifications
- Flow will be a very simple flowchart showing the major tasks, decisions and loops
- You can probably spot the data needed and its types at this point.
- Try to picture where each variable is created and to which tasks it must be passed on to.
- You can always come-up with multiple flows and you can always revise your initial design.


Welcome Message
Do you wish to go first?
Initialize Game State
Turn == USER
Get User decision (int choice)
Get Computer Decision (int choice)
Print game State
Compute Game Result
turn= other Player, Game not finished
Win, Lose, Draw
Game Over Message
Play Again?
Yes
No
Exit

Top-down and Bottom-up Design

- Top-down: Start with a set of high level tasks (that will probably be called from main)
  - Split each task to manageable (you can code a function for it) recursively
  - Divide-and-conquer; will create a hiearchy of lower-level functions to implement
  - The functions you will identify might be too coupled with the task (not reusable)
- Bottom-up: identify various components that will be required
  - Implement first the required functions, more generically.

Pseudocode Design

- Another good idea is to convert the flow diagram to simple pseudocode
- Write the pseudocode as C comments
- Start implementing each item in pseudocode, and leave the comments
- Example:

```
Loop number of times
  Prompt user and get integer value
  Calculate factorial
  Print factorial
```

Tic-tac-toe: Data structure

- Think of what represents a state of the program
  - Whose Turn is it?
  - The tic-tac-toe table, which cells are marked with X or O?
- Types of the variables:

```
#define NUMSTATES 9
enum { NOTHING, CROSS, NOUGHT };
int state[NUMSTATES];
```

- We defined state as just int Nothing, Cross, Nought, another alternative is to define as struct:

```
struct tictactoe_state {
    int row; int col;
    int state  = NOTHING; };
```

Bottom-up Design

- Looking at the flow and specifications, we can determine the functions we need
- Print the message and get a value as input:

```
int getint_from_user(char* message);
```

- Plot the tic-tac-toe table;

```
void plot_state(int state[]);
```

- Let the computer decide on a play

```
void get_computer_decision(int state[]);
```

  can initially implement it as purely random, but can also try and test more sophisticated things.

Bottom-up Design

- Make your code more readable with enums, constants

```
enum Turn { USER, COMPUTER };
enum Result { PLAYING, WIN, LOSE, DRAW };
enum Turn turn;
enum Result result;
```

- Check if the game is over or still continuing

```
enum Result compute_result(int state[], enum Turn turn);
```

Bottom-up Design

- Implement each function independently

- Write a driver method (main method) to test your function.
  - Make sure to test edge cases with multiple inputs. Did you check the diagonal?

- The function prototypes are your checklist, cross-out an item when you have tested your implementation throughly.

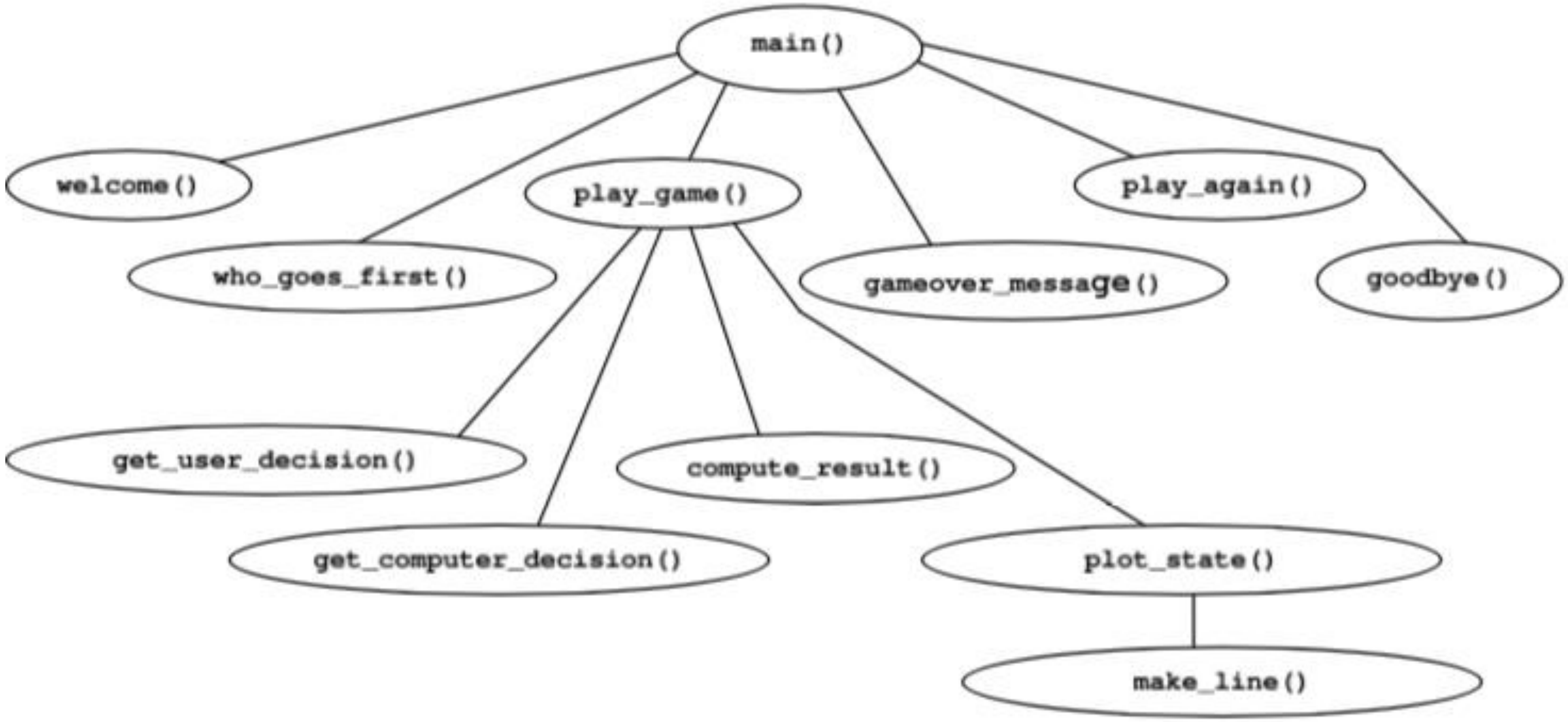
- Consider keeping your test codes around (configure multiple main methods in a target called test in your Makefile)

Top-down Design

- Start coding the main function:

```
int main(void){
  enum Turn turn; enum Result result; int newgame = 1;
  welcome();
  while (newgame) {
       turn = who_goes_first();
       result = play_game(turn);
       gameover_message(result);
       newgame = play_again();
  };
  goodbye();
  return 0;
}
```

Top-Down Hierarchy



Functions implementation

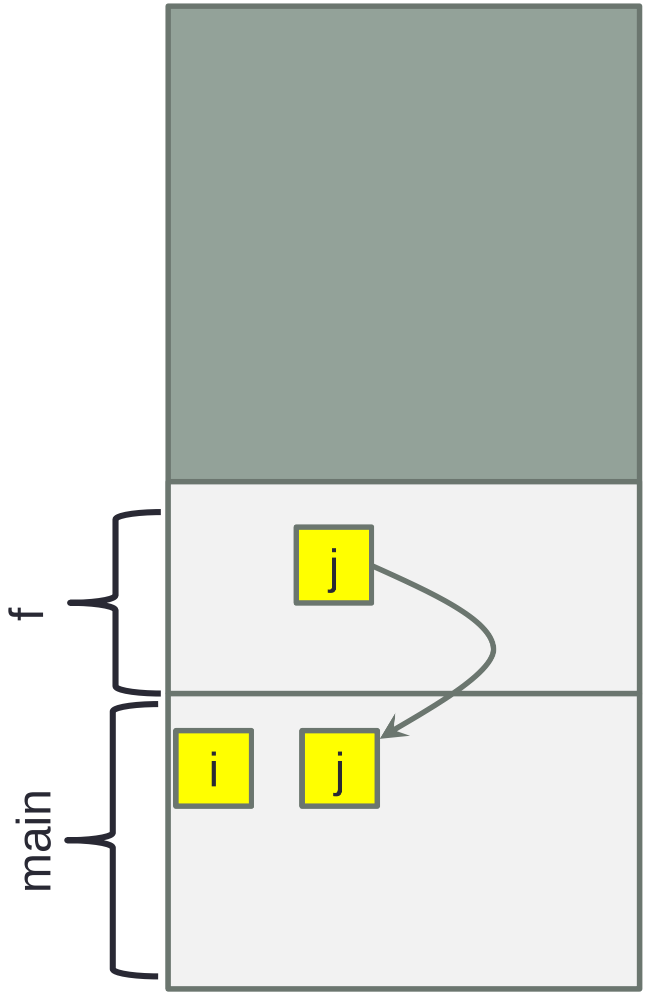
- It might be confusing at first to create functions and pass the required variables.

- Do not back down because of compiler errors, insist on correct modular design.
  - Never convert a variable to global to get rid of an error.

- We will see some example cases for function designs.

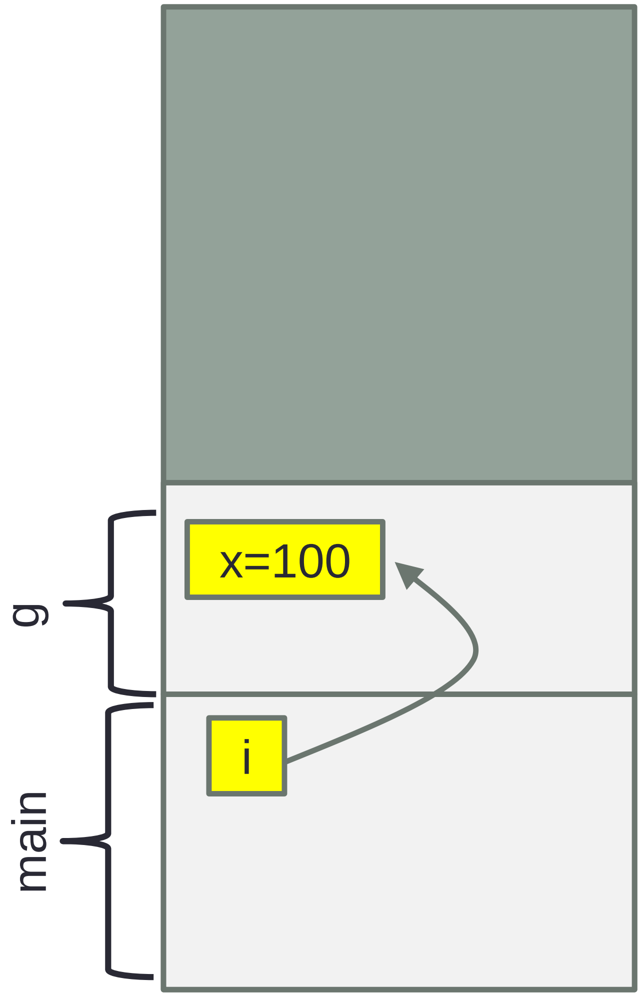
Return multiple values

```
int f(int* j) {
  *j=4;
  return 5;
}

int main(void) {
  int i=0;
  int j=0;
  i = f(&j);
  printf("%d %d \n", i, j);

  return EXIT_SUCCESS;
}
```



Beware of the reverse

```c
int* g() {
  int x = 100;
  return &x;
}
int main(void) {
  int* i;
  i = g();
  printf("%d \n", *i);

  return EXIT_SUCCESS;
}
```



- Returning stack variables is never a good idea
- After g terminates, x will be removed.
- If you have checked the warnings;

```
Test.c:21:2: warning: function returns address of local variable [-Wreturn-local-addr]
```

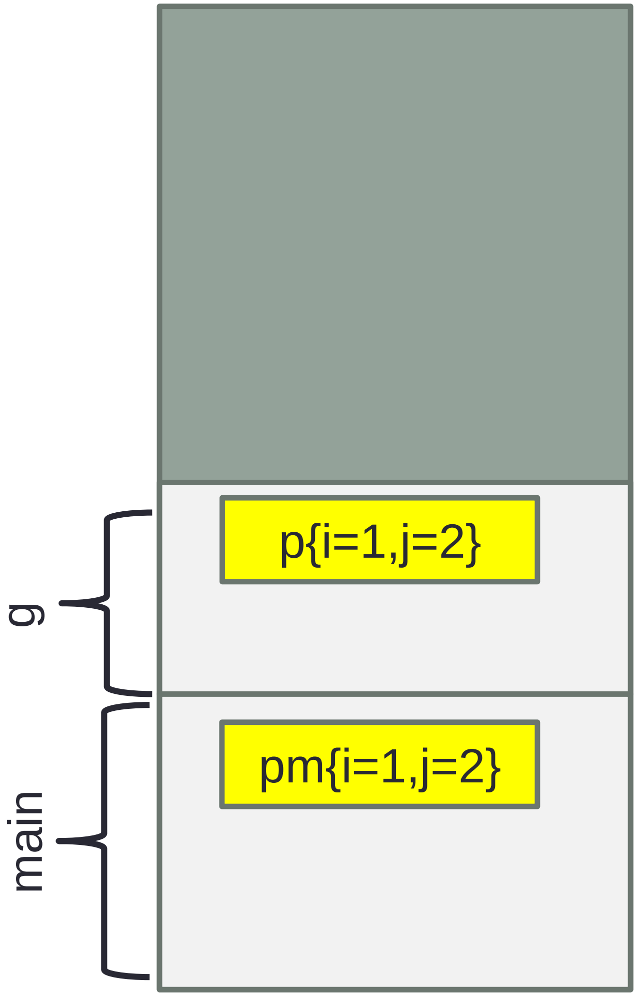
Returning with structs or arrays

- Instead of using pass-by-reference, you can use an array or a struct to return multiple values from a function
- Array solution: You should allocate the array from Heap!
- Struct solution: Your struct will be copied to another struct in the caller function.

Returning a struct

```
struct pair {
int i;
int j;
};

struct pair g() {
   struct pair p = {.i=1, .j=2};
   return p;
}


int main(void) {
   struct pair pm = g();
   printf("%d %d \n", pm.i, pm.j);
   return 0;
}
```

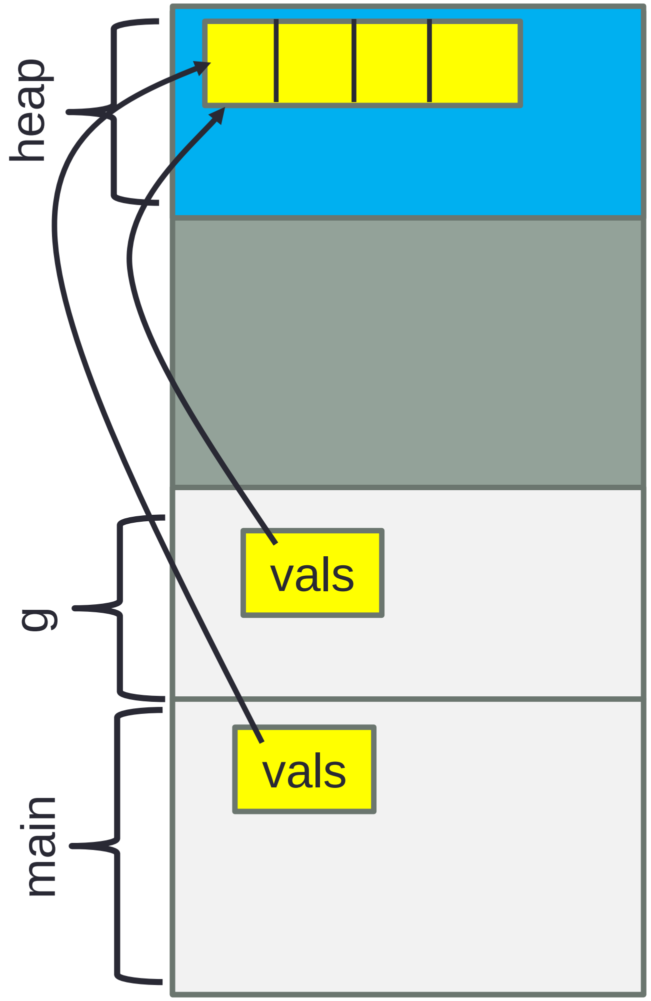
- The returned structs values are copied to variable in main. They have different addresses



Returning an array

```c
int* g() {
  int* values = malloc(sizeof(int)*2);
  values[0]=1;
  values[1]=2;
  return values;
}

int main(void) {
  int* vals = g();
  printf("%d %d \n", vals[0], vals[1]);
  free(vals);
  return 0;
}
```

- You can also create the array in main and pass its pointer to the called function
- It is always your responsibility to free the memory allocated in Heap



Laying traps for logical errors

- Logical errors are the worst type of errors
  - Program will not usually crash, but will not do what it is expected to
  - You don’t know where the problem is, so finding that will require debugging.
- By being defensive and writing code to prevent these errors you can save significant amount of time.
- Do not cut down from lines of code, cut down from debugging time.

Trick for comparisons

```
int main(void) {
  int i=0;
  if(i=0) {
    printf("It is zero");
  }
  return 0;
}
```

We wrote = instead of ==.
C compiler will not give an error, but just a warning.
You can trade this logical error with an compiler error.

- This program will not output anything.
- The problem is just one character (if(i=0)), C will return the value assigned to if statement
- One defensive programming strategy is to write it as;
  if(0=i)
  which will give an compiler error if you forget one = character.

Asserts

```c
double circle_area(double radius) {
  double pi=3.14;
  assert(radius>0);
  return pi*radius*radius;
}

int main(void) {
  printf("%f", circle_area(-1));
  return 0;
}
```

```
Assertion failed: radius>0, file
../src/Test.c, line 46
```

- Protect against logical errors by placing assert statements in your code.
- It will show the failed assertion and the line number.

Avoid overloading a line

```
int main(void) {
   int array[4] = {1,2,3,4};
   int i = 0;
   int sum = 0;
   while(i<4) {
      sum += array[i++];
   }

   return 0;
}
```

- It is tempting to use incrementation and other shortcuts to cut down the number of lines in your code
- Avoid this, keep your code simple and easy to read
- In the example above, increment i in a separate line.

Memory Access Violations

- One of the most common error sources for C is related to memory access violations
  - Even worse, they sometimes work depending on memory usage.
  - Cause of most; «But it was working on my computer» errors.
- Some reasons:
  - Fault in array index. Lower bound inclusive, upper bound exclusive
  - Not providing the correct size in malloc
  - Not using malloc but just defining a pointer variable
  - Early or multiple frees on a single heap allocation
- Finding them out is hard because the executable files just crash without leaving any clues

Memory Access Violations

```c
struct point_3d {
  double x;
  double y;
  double z;
};

int main(void) {
  struct point_3d* points;
  int num_points = 10;
  points = malloc(num_points);
  int i;
  for (i = 0; i < num_points; ++i) {
     points[i].x=1;
     points[i].y=2;
     points[i].z=3;
  }
  for (i = 0; i < num_points; ++i) {
     printf("%f, %f, %f \n", points[i].x, points[i].y, points[i].z);
  }

  return 0;
}
```

A ticking time bomb!!!
Might work for a while.
Some variable’s values can change for no reason.
Why?

Solution: Memory Debuggers

- Use a memory debugger.
- Valgrind is a great tool.

```
$ valgrind ./Test
==3300== Memcheck, a memory error detector
==3300== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et
al.
==3300== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright
info
==3300== Command: ./Test
==3300==
==3300== Invalid write of size 8
==3300==    at 0x10862B: main (in Test)
==3300==  Address 0x521c048 is 8 bytes inside a block of size 10
alloc'd
==3300==    at 0x4C2FB0F: malloc (in vgpreload_memcheck-amd64-linux.so)
==3300==    by 0x108604: main (in /Test)
==3300==
```

Recursive Algorithms

- Divide & Conquer methodology; divide the problem into sub-problems with the same structure.
- Nature of recursion:
  - One or more simple cases of the problem (the base case or stopping cases) have a trivial non-recursive solution
  - The cases of the problem can be reduced to sub-problems that are closer to stopping cases.
  - Eventually the problem can be reduced to stopping cases only.

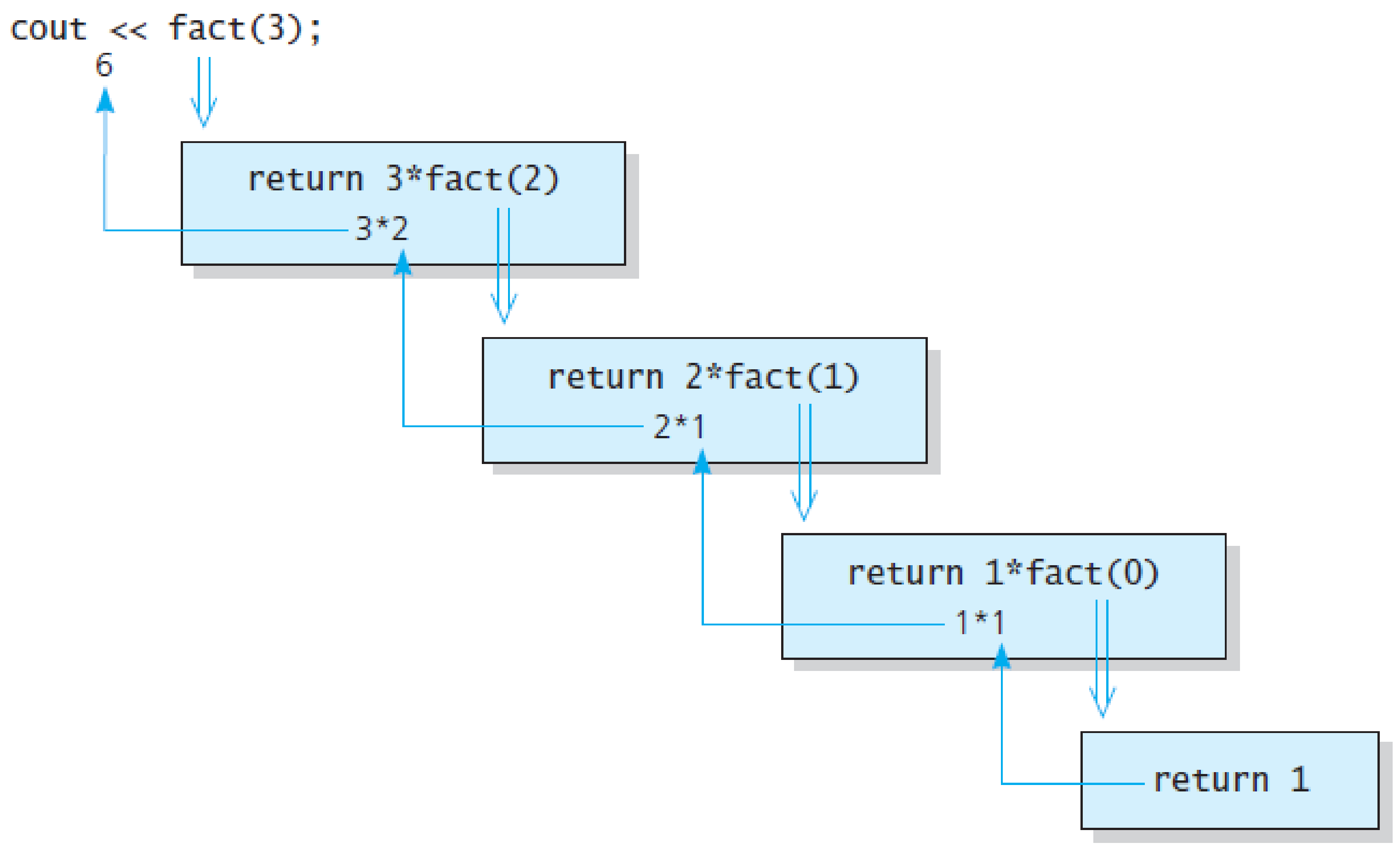
```
if (stopping case)
  solve it
else
  reduce the problem using recursion
```

Example Problem: Factorial

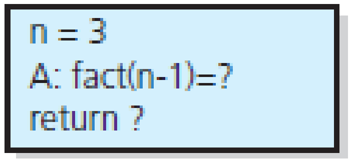
- $n! = n \times (n-1) \ldots 2 \times 1$
- 0! = 1
- $factorial(n) = \begin{cases} 1, & if\ n = 0 \\ n \times factorial(n-1) & if\ n > 0 \end{cases}$

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

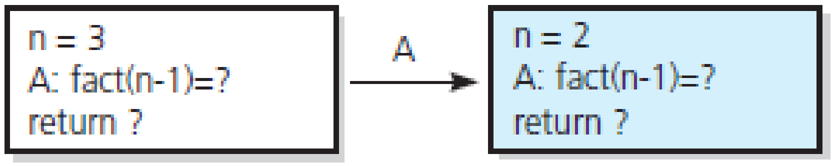
Factorial Example



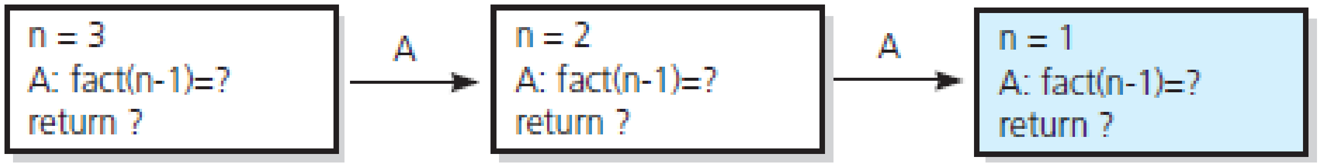
The initial call is made, and method `fact` begins execution:



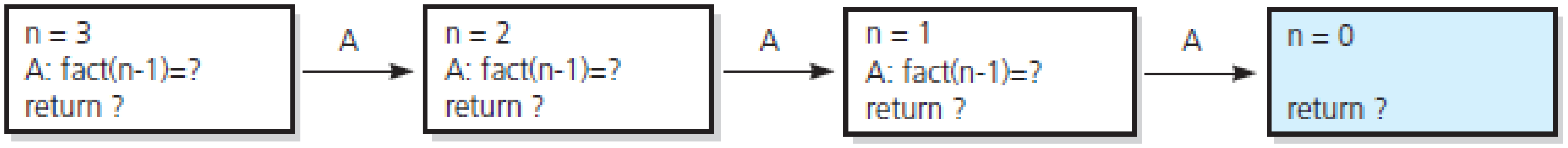
At point A a recursive call is made, and the new invocation of the method `fact` begins execution:



At point A a recursive call is made, and the new invocation of the method `fact` begins execution:

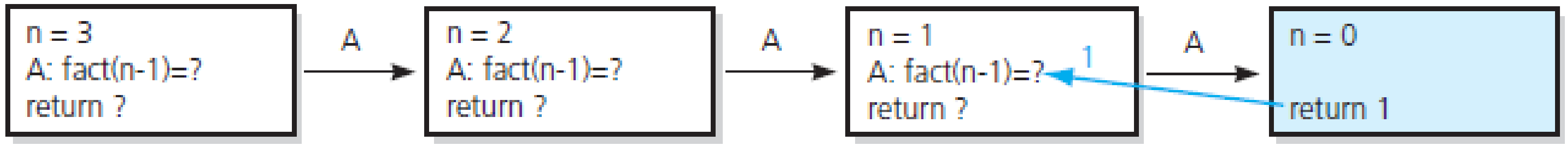


At point A a recursive call is made, and the new invocation of the method `fact` begins execution:

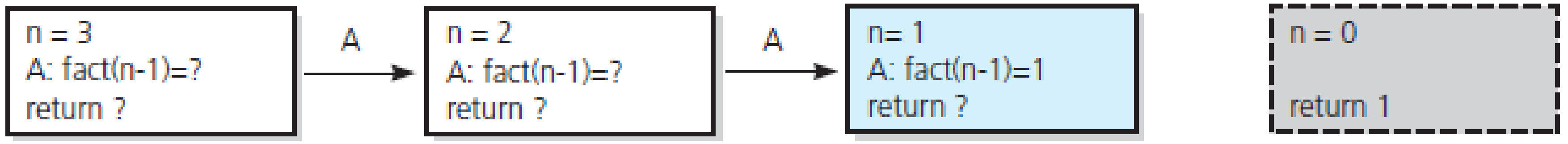


(*continues*)

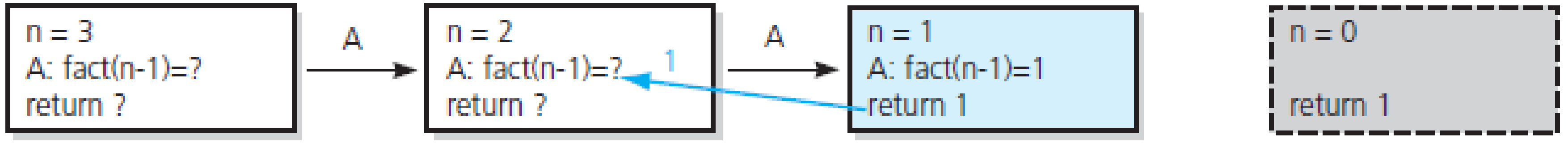
This is the base case, so this invocation of `fact` completes and returns a value to the caller:



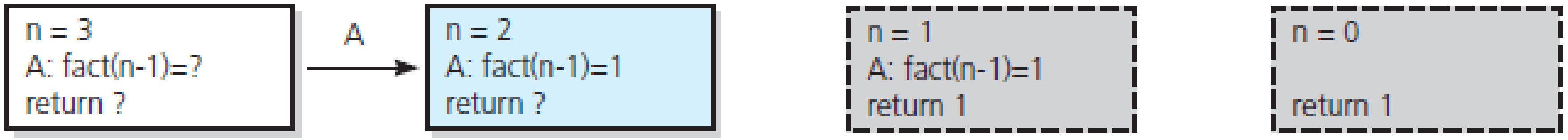
The method value is returned to the calling box, which continues execution:



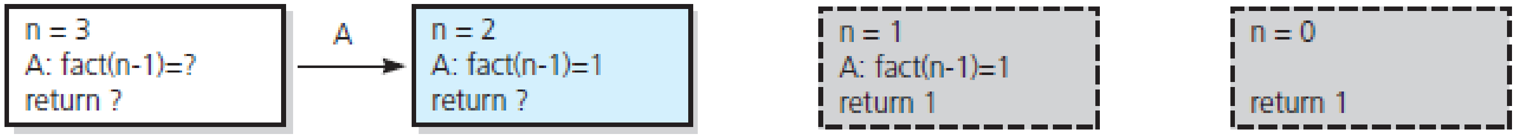
The current invocation of `fact` completes and returns a value to the caller:



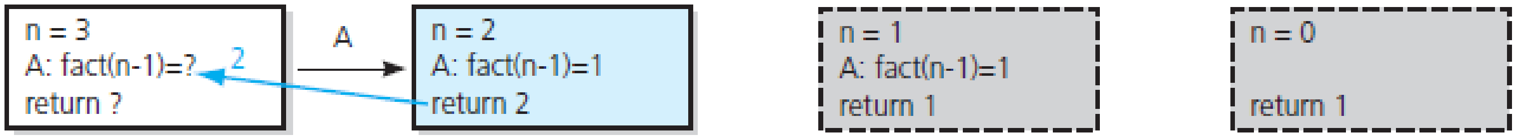
The method value is returned to the calling box, which continues execution:



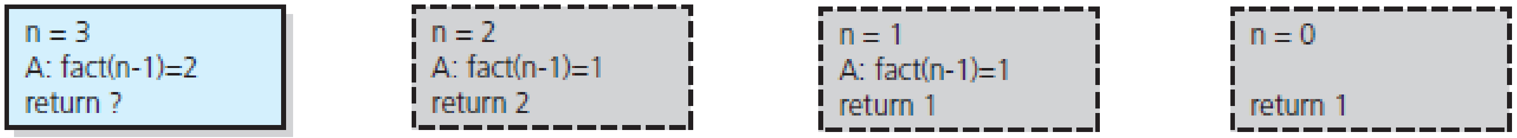
The method value is returned to the calling box, which continues execution:



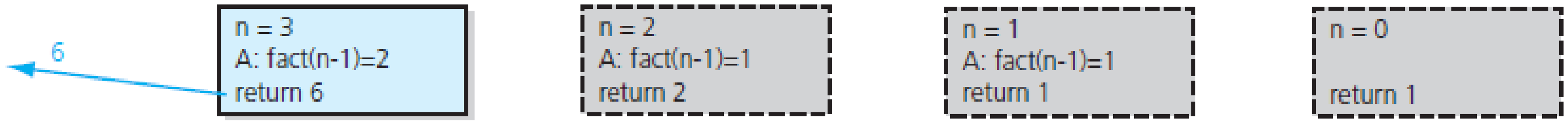
The current invocation of `fact` completes and returns a value to the caller:



The method value is returned to the calling box, which continues execution:



The current invocation of `fact` completes and returns a value to the caller:



The value 6 is returned to the initial call.

Writing a String Backward

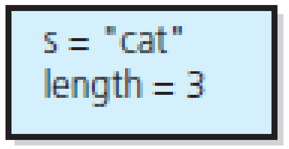
- Problem: Write the given string of characters in reverse order
- Recursive Solution:
  - Base case: Write the empty string backward
  - Recursive case: Print the last character, then solve the same problem for the remaining n-1 characters

```
writeBackward(in s:string)
       if(the string is empty)
          Do nothing // base case
       else
          Write the last character of s
          writeBackward(s minus its last character)
```

Write Backward

```
void writeBackward(char* s, int size) {
// Writes a character string backward.
// Precondition: The string s contains size characters,
// where size >= 0.
// Postcondition: s is written backward, but remains
// unchanged.
if (size > 0) {
printf("%c", s[size-1]);
    // write the rest of the string backward
    writeBackward(s, size-1); // Point A
}
    // size == 0 is the base case - do nothing
}
```

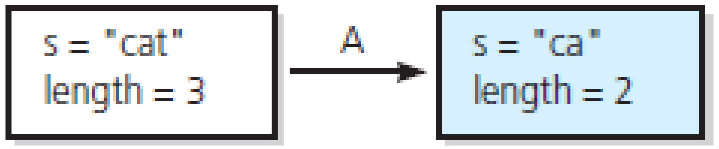
Write Backward Trace



Output line: **t**

Point A (`writeBackward(s)`) is reached, and the recursive call is made.

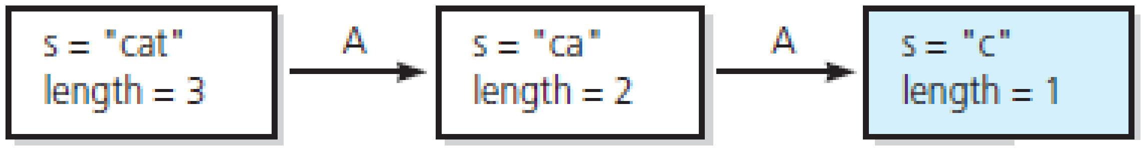
The new invocation begins execution:



Output line: **ta**

Point A is reached, and the recursive call is made.

The new invocation begins execution:

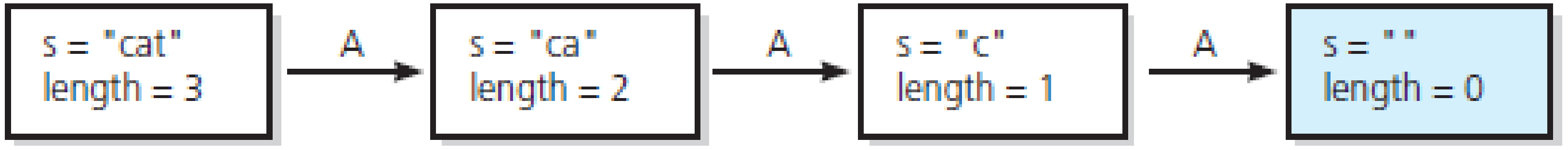


(continues)

Output line: **tac**

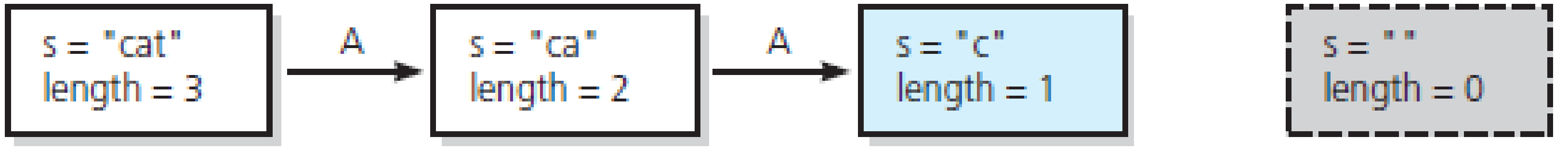
Point A is reached, and the recursive call is made.

The new invocation begins execution:

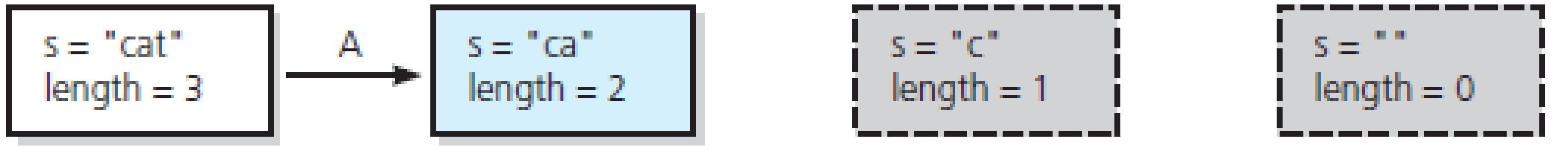


This is the base case, so this invocation completes.

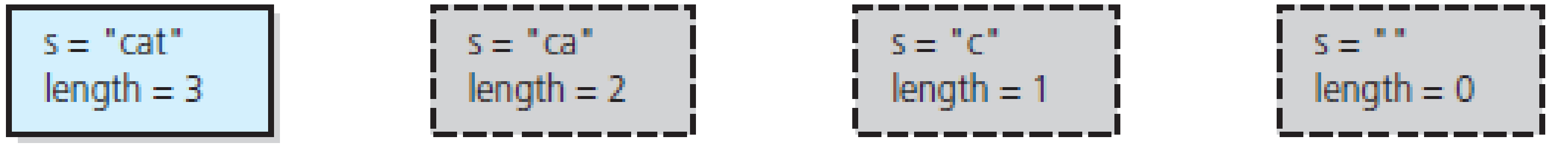
Control returns to the calling box, which continues execution:



This invocation completes. Control returns to the calling box, which continues execution:



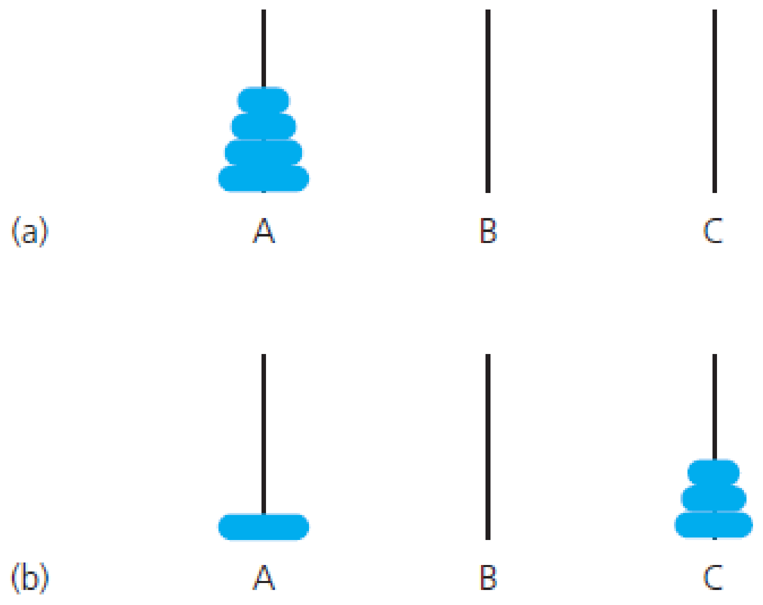
This invocation completes. Control returns to the calling box, which continues execution:



This invocation completes. Control returns to the statement following the initial call.

Towers of Hanoi

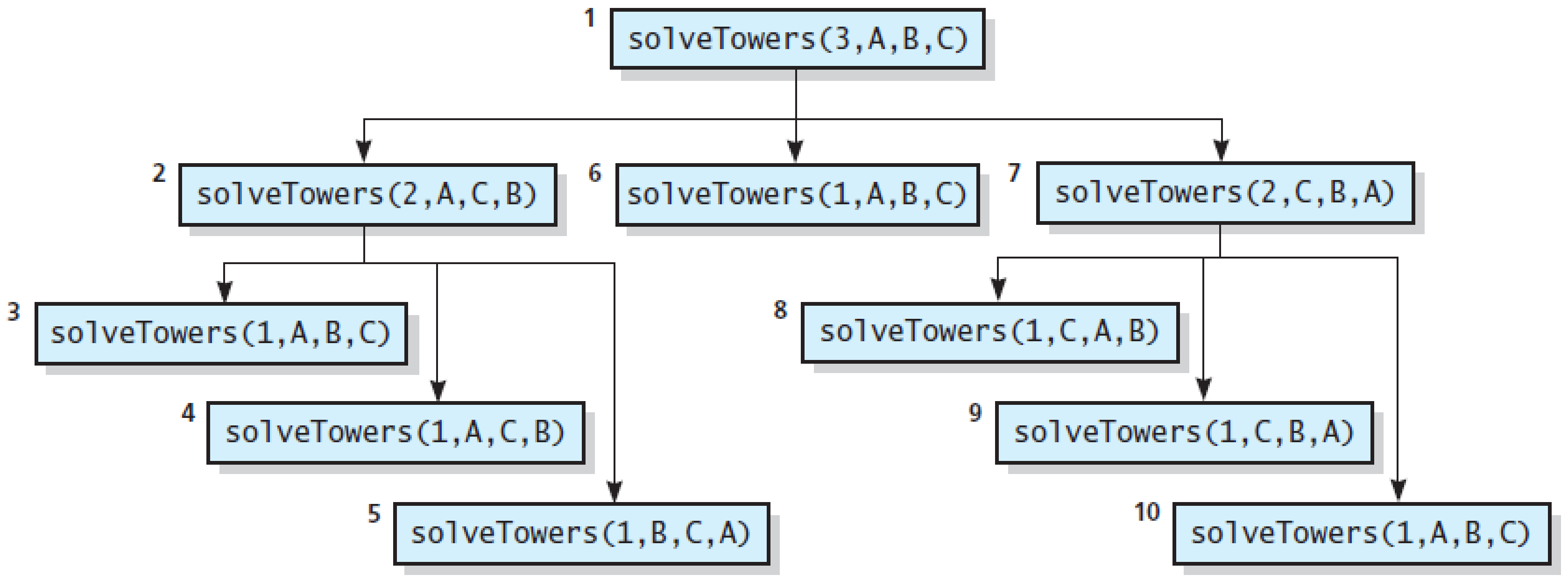
- There are n disks and three poles: A(source), B(destination), C(spare)
- Move all n disks from pole A to B.



Towars of Hanoi

```
void solveTowers(int count, char source,
                char destination, char spare) {
   if (count == 1) {
      printf("Move top disk from pole %c to pole %c nn",
                                   source, destination);
   } else {
      solveTowers(count-1, source, Spare , destination);
      solveTowers(1, source, destination, spare);
      solveTowers(count-1, spare,destination, source);

   }
}
```

Towers of Hanoi



1 solveTowers(3,A,B,C)
2 solveTowers(2,A,C,B)
6 solveTowers(1,A,B,C)
7 solveTowers(2,C,B,A)
3 solveTowers(1,A,B,C)
4 solveTowers(1,A,C,B)
5 solveTowers(1,B,C,A)
8 solveTowers(1,C,A,B)
9 solveTowers(1,C,B,A)
10 solveTowers(1,A,B,C)