

Lecture 2: **C Programming (Part II)**

01204212 Abstract Data Types and Problem Solving

Department of Computer Engineering
Faculty of Engineering, Kasetsart University
Bangkok, Thailand.



Department of
Computer Engineering
Kasetsart University



Outline

- Repetition Control Structure
- Library and User-defined Functions
- Recursion

Fundamental Flow Controls

- Sequence
 - Subroutine (function)
 - Selection: if, if-else, switch
- ➔ Repetition: for, while, do-while

Repetition / Loop

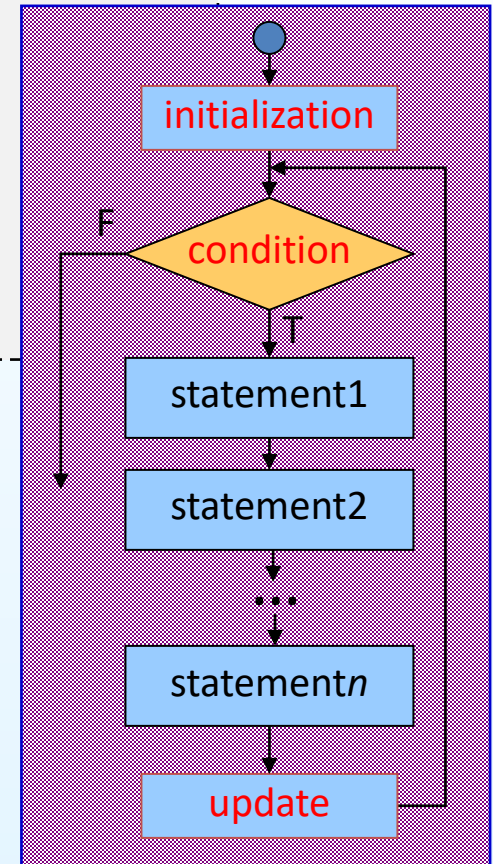
- A group of instructions that is executed **repeatedly** according to **some conditions**
- In C program, there are three types of loops:
 - for
 - while
 - do-while

Loop: The for Statement

Syntax:

```
for (initialization; condition; update)  
{  
    statement1;  
    statement2;  
    ...  
    statementn;  
}
```

- Use the **for** keyword
- There are three parts in the header
 - *initialization* is executed once at beginning
 - *condition* is checked for the next iteration
 - *update* is executed at the end of each iteration



Example: The for Statement

What is the output of the code fragment?

Code Fragment #1:

```
1: for (i=0; i<10; i++) {  
2:     printf("%d\n", i);  
3: }
```

Code Fragment #2:

```
1: for (i=0,j=0; i+j<10; i++,j+=2) {  
2:     printf("%d\n", i+j);  
3: }
```

Exercise 1 (5 mins.)

A person invests \$1000 in a saving account yielding 5% interest. Assuming that all interest is left on deposit in the account, write a program to calculate the amount of money in the account at the end of the 10th years.



for Statement: The Flexibility

- Each expression in the header of a **for** loop is **optional**
 - If **initialization** is omitted, no initialization is performed
 - If **condition** is omitted, it is **always** considered to be **True** so that the loop is **infinite**
 - If **update** is omitted, no increment/decrement is performed
- However, both semicolons are always required

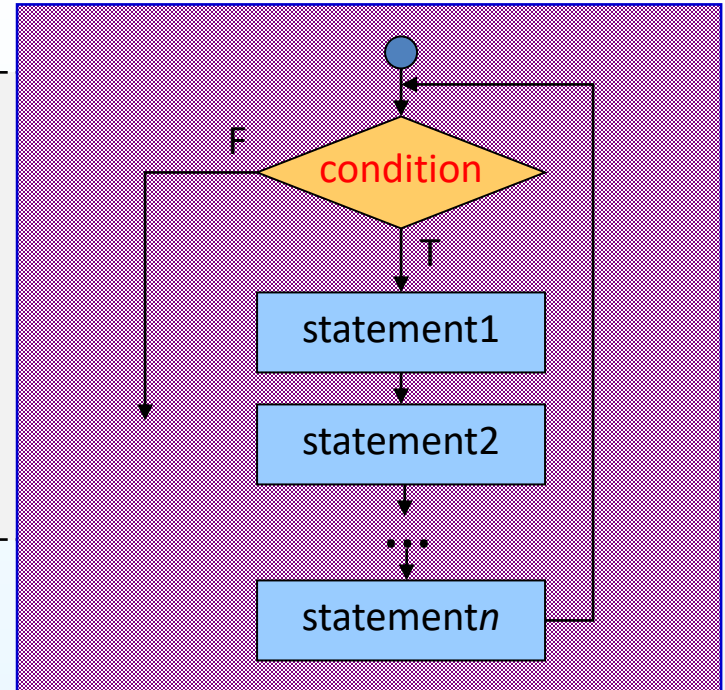
```
for ( ; ; )  
{  
    statements;  
}
```


Loop: The `while` Statement

Syntax:

```
while (condition)  
{  
    statement1;  
    statement2;  
    ...  
    statementn;  
}
```

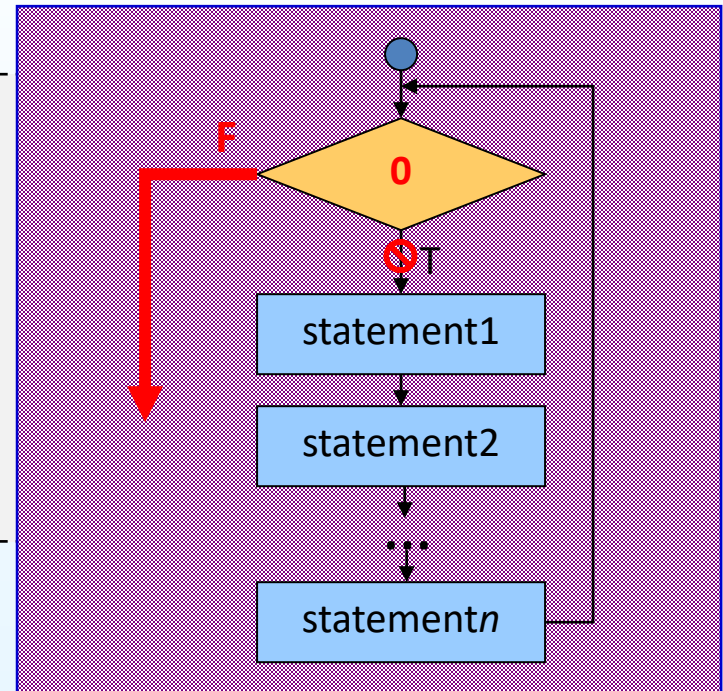
- Use the `while` keyword
- The `condition` is tested at the `top` of loop
- The loop statements are `repeatedly` executed if the `condition` is still `True`



while Statement: Zero Loop

- If the condition is **False initially**, the loop statements are **never** executed

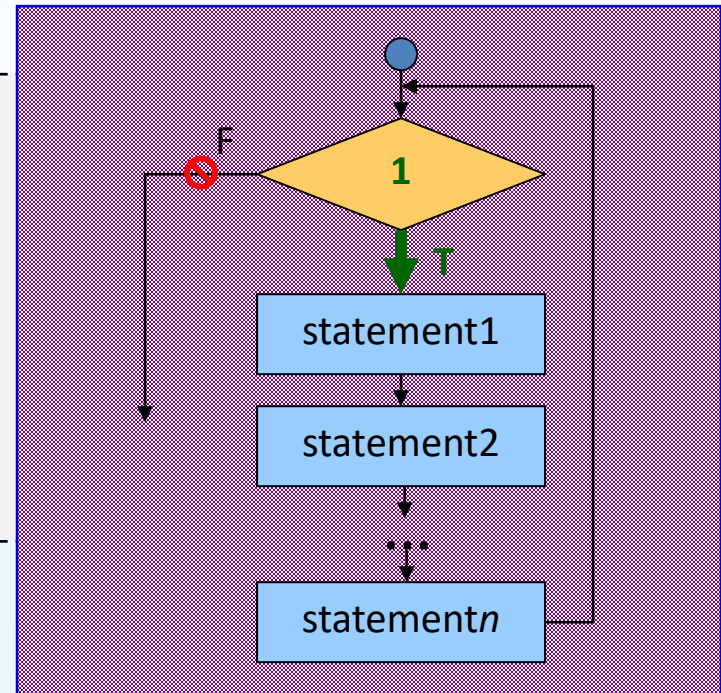
```
while (0)
{
    statement1;
    statement2;
    ...
    statementn;
}
```



while Statement: Infinite Loop

- If the condition is **always True**, the loop statements are **infinitely** executed

```
while (1)
{
    statement1;
    statement2;
    ...
    statementn;
}
```



Exercise 2 (5 mins.)

Write a program to get a **positive number** from the user, then print it out in the **reverse order**

Enter a number: 123456
654321



Exercise 3 (10 mins.)

Write a program to get a **message** from the user, then convert and print it out by the **uppercase letters**

```
Enter a message: Hello world !!!  
HELLO WORLD !!!
```

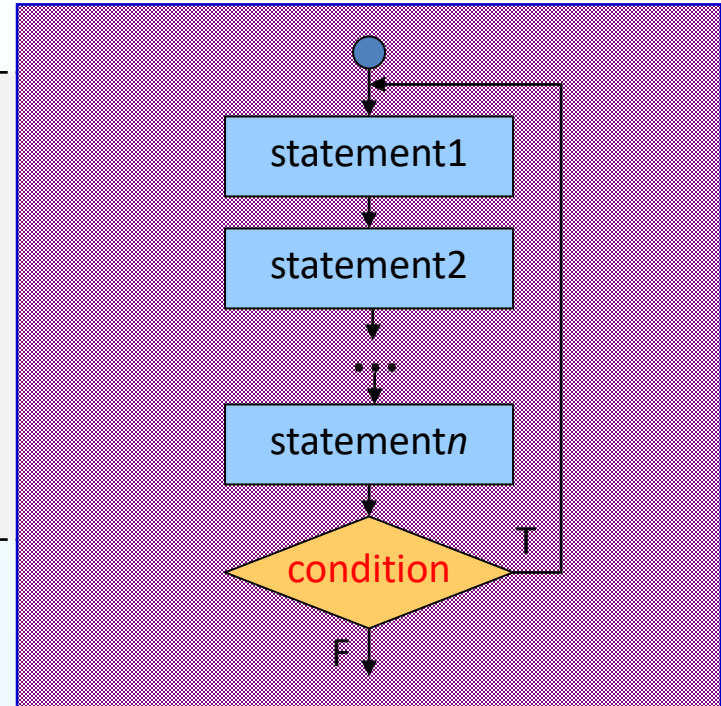
Hint: Try the `getchar()` function



Loop: The do-while Statement

Syntax:

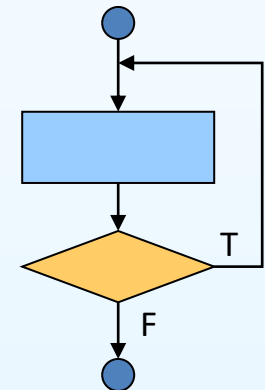
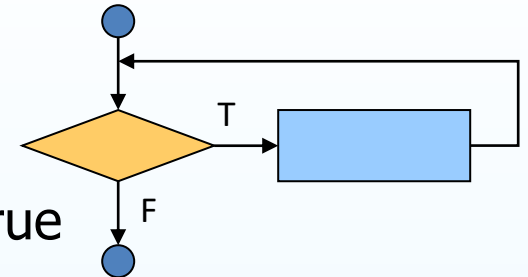
```
do
{
    statement1;
    statement2;
    ...
    statementn;
} while (condition);
```



- Use the **do** and **while** keywords
- The **condition** is tested at the **bottom** of loop
- Require a semicolon **;** at the end
- The loop statements are **repeatedly** executed if the **condition** is still **True**

The **while** vs. **do-while** Statements

- Use a **while** loop
 - Condition is first tested
 - Action is then performed if the condition is True
 - Loop can be skipped altogether
- Use a **do-while** loop
 - Action is first performed
 - Then, condition is tested until it is False
 - Loop exactly run at least once



Again, Exercise 3 (5 mins.)

Write a program to get a **message** from the user, then convert and print it out by the **uppercase letters**

```
Enter a message: Hello world !!!  
HELLO WORLD !!!
```

Hint: Try the `getchar()` function

Using the **do-while**
statement



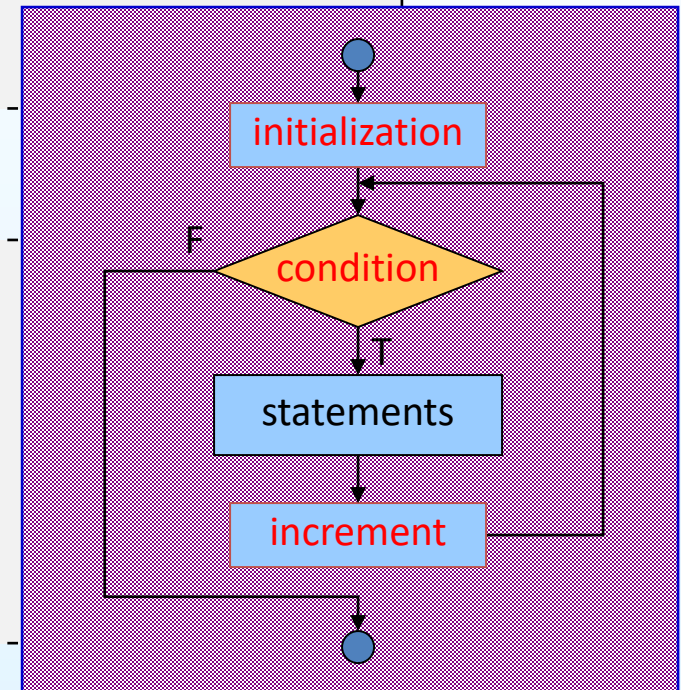
The *while* vs. *for* Statements

- A *while* loop may be equivalent to the *for* loop in some situation

```
for (initialization; condition; update)  
{  
    statements;  
}
```

↓ re-write

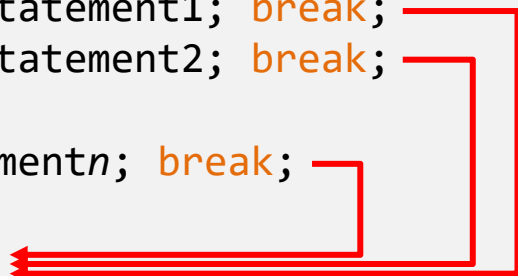
```
initialization;  
while (condition)  
{  
    statements;  
    update;  
}
```




The break Statement

- Cause the **immediate exit** from the control structure


```
switch (expression) {  
    case const1: statement1; break;  
    case const2: statement2; break;  
    ...  
    default: statementn; break;  
}  
statement_after;
```



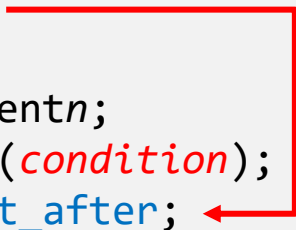
```
while (condition) {  
    statement1;  
    break;  
    ...  
    statementn;  
}  
statement_after;
```



```
for (initialization; condition; update) {  
    statement1;  
    break;  
    ...  
    statementn;  
}  
statement_after;
```



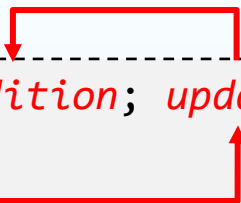
```
do {  
    statement1;  
    break;  
    ...  
    statementn;  
} while (condition);  
statement_after;
```



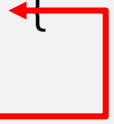
The continue Statement

- **Skip** the remaining statements of the current iteration and consider for the next iteration


```
for (initialization; condition; update) {  
    statement1;  
    continue; _____  
    ...  
    statementn;  
}  
statement_after;
```



```
while (condition) {  
    statement1;  
    continue; _____  
    ...  
    statementn;  
}  
statement_after;
```



```
do {  
    statement1;  
    continue; _____  
    ...  
    statementn;  
} while (condition);  
statement_after;
```

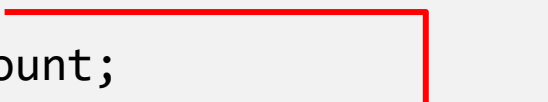


Example: break and continue

What is the output of the code fragment?

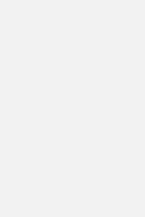
Code Fragment #1:

```
1: int count, sum = 0;
2: for (count=1; count<=10; count++) {
3:     if (count == 3)
4:         break;
5:     sum += count;
6: }
7: printf("%d\n", sum);
```



Code Fragment #2:

```
1: int count, sum = 0;
2: for (count=1; count<=10; count++) {
3:     if (count == 3)
4:         continue;
5:     sum += count;
6: }
7: printf("%d\n", sum);
```



Common Loop Patterns

- Counter-controlled loop
- Interactive loop
- Sentinel-controlled loop
- Loop and a half
- Forever loop with break
- Nested loop

Fundamental Flow Controls

- Sequence

➡ Subroutine (function)

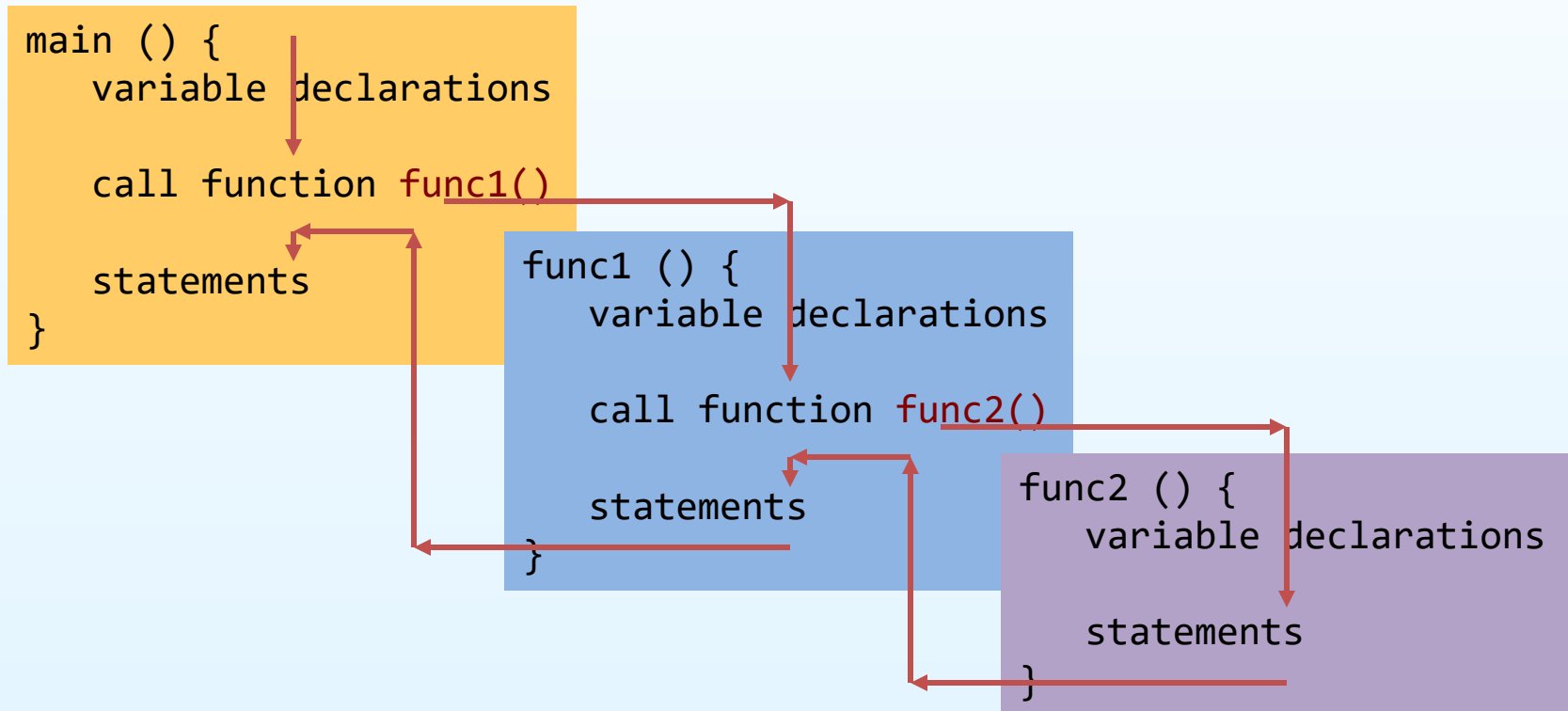
- Selection: if, if-else, switch
- Repetition: for, while, do-while

Concept of Function

- Construct a program from **smaller** pieces or components
 - These smaller pieces are called “**module**”
- Each piece **more manageable** than the original program
- **Encapsulate** some computation
- Allow **cleaner** and **smaller** programs
- Facilitate **reuse** of code parts
- Can be **parameterized** and return a value

Program Modules in C

- C standard libraries
 - A wide variety of functions
- User-defined functions



Math Library Functions

- Perform common mathematical calculations ... [more details](#)
- Use preprocessor `#include <math.h>`
- Calling function

Syntax:

```
function_name(var_list);
```

- All math functions return data type `double`

Math Functions

acos	asin	atan	atan2
cos	cosh	sin	sinh
tan	tanh	exp	frexp
ldexp	log	log10	modf
pow	sqrt	ceil	fabs
floor	fmod		

User-defined Functions

- Modularize a program
- All variables defined inside the functions are **local**
- Use **parameters** to communicate between functions

The Definition of a Function

Syntax:

```
return_type function_name(parameter_list)  
{  
    declarations_and_statements  
    return value;  
}
```

- The function header
 - *return_type*: data type of a result, use *void* if returns nothing
 - *function_name*: any valid identifiers
 - *parameter_list*: a list of data type and variable pairs separating with commas
- The function body
 - Sequence of statements
 - Use a *return* statement to indicate the end of the function and return the value; can be omitted if *return_type* is *void*

Example: A Function

```
1: #include <stdio.h>
2:
3: int adding(int a, int b) {
4:     return a+b;
5: }
6:
7: int main(void) {
8:     int a, b, sum;
9:
10:    printf("Enter a and b: ");
11:    scanf("%d %d", &a, &b);
12:    sum = adding(a, b);
13:    printf("The sum is %d\n", sum);
14:    return 0;
15: }
```

Return type

Function name

Data type of a parameter

Local variable

Return statement and return value

Calling the function; output of the function is returned to here

The Function Prototype

- A function prototype is the function header immediately ending with the semicolon
 - Locate at the beginning of the program
- The function prototype is needed when the **function comes after calling** or using
- Also, programmer can use the prototypes to validate functions

Example: The Function Prototype

```
1: #include <stdio.h>
2:
3: int adding(int a, int b);
4:
5: int main(void) {
6:     int a, b, sum;
7:
8:     printf("Enter a and b: ");
9:     scanf("%d %d", &a, &b);
10:    sum = adding(a, b);
11:    printf("The sum is %d\n", sum);
12:    return 0;
13: }
14:
15: int adding(int a, int b) {
16:     return a+b;
17: }
```

Function prototype (ending with ;)



Header Files

- A header file is a file containing function prototypes
- For C standard library:
 - e.g., `#include <stdio.h>`
`#include <stdlib.h>`
`#include <math.h>`
- Programmers' header files
 - Contain the defined function prototypes in *file.h*
 - e.g., `#include "file.h"`

Calling Functions

➔ Call by value

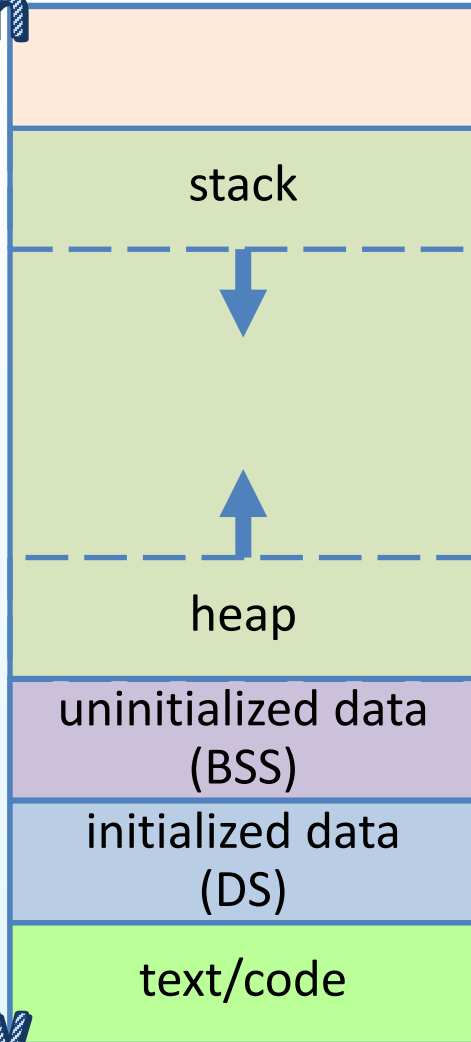
- Copy the **values** from arguments to parameters of the function
- Changing the value of parameters **does not effect** the original

• Call by reference

- Pass the **reference** of arguments to parameters
- Changing the value of parameters **effects** the original

Memory Layout

high



command-line arguments and environment variables

stack



heap

uninitialized data (BSS)

initialized data (DS)

text/code

initialized to zero by exec

read from program file by exec

low

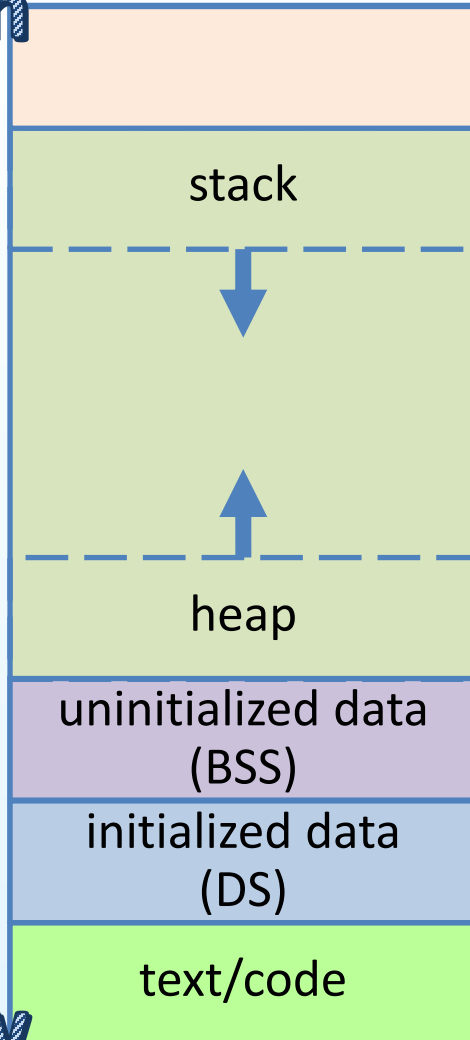
high...

low ...

Address	Value
0x7fffb5	1 byte
0x7fffb4	00000000 (0 = '\0')
0x7fffb3	00001010 (10 = '\n')
0x7fffb2	01101111 (111 = 'o')
0x7fffb1	01101100 (108 = 'l')
0x7fffb0	01101100 (108 = 'l')
0x7fffaf	01100101 (101 = 'e')
0x7fffae	01001000 (72 = 'H')
0x7fffad	
0x7fffac	4-byte integer
0x7fffab	
0x7fffaa	
0x7fffa9	
0x7fffa8	

Memory Layout

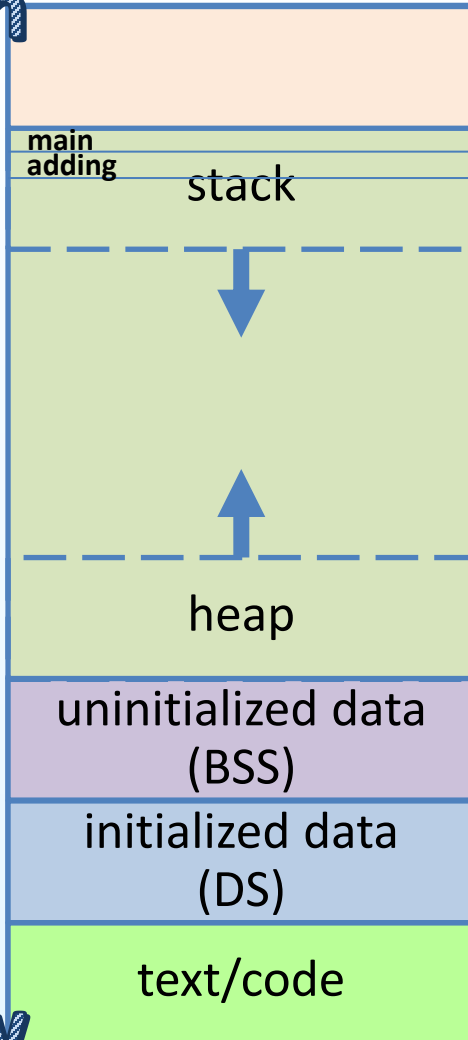
high



- Text or Code Segment
 - Program **instructions** (compiled binary)
 - Initialized Data Segment
 - All **global**, **static**, **constant** and **external** variable that are initialized beforehand
 - Uninitialized Data Segment
 - All **global** and **static** variables that do not explicit initialization; be initialized by the kernel
 - Heap
 - Dynamic memory **allocation**
 - Stack
 - All **local** variables are stored in the LIFO structure
 - Command Line Arguments
 - Arguments like argc and argv
- ... [more details](#)

Memory Layout

high



low

```
1:  #include <stdio.h>
2:
3:  int a = 1, b;
4:
5:  int adding(int a, int b) {
6:      return a+b;
7:  }
8:
9:  int main(void) {
10:     int sum;
11:
12:     b = 2;
13:     sum = adding(a, b);
14:     printf("The sum is %d\n", sum);
15:     return 0;
16: }
```

Example: Calling Function by Value

high

What is the output of the following program?

stack

```
1: #include <stdio.h>
2:
3: void increment(int a) {
4:     a = a + 1;
5: }
6:
7: int main(void) {
8:     int a = 1;
9:
10:    increment(a);
11:    printf("%d\n", a);
12:    return 0;
}
```

low

main

increment

a = 1

a = 2

Storage Classes

- Storage duration
 - How long a variable exists in the memory
- Scope
 - Where the variable can be referenced in the program
- Linkage
 - Specify the program files in which a variable is known

Storage Classes: Automatic

- A variable is created and destroyed within its block
- **auto** : default for local variables
- **register** : put the variable into a high-speed register

```
1: #include <stdio.h>
2:
3: int main(void) {
4:     register int i = 0;
5:     auto int sum = 0;
6:
7:     for (i=1; i<=10000; i++)
8:         sum += i;
9:     printf("The sum is %d\n", sum);
10:    return 0;
11: }
```



Storage Classes: Static

- A variable exists for the entire program execution
- **static** : keep the value of local variable of the function
- **extern** : use for multiple source files linked together

```
1: #include <stdio.h>
2:
3: int increment() {
4:     static int x = 0;
5:     return ++x;
6: }
7:
8: int main(void) {
9:     int i, sum = 0;
10:
11:     for (i=1; i<=10000; i++)
12:         sum = increment();
13:     printf("The sum is %d\n", sum);
14:     return 0;
15: }
```

Stored in
initialized data



Recursion

- Recursion is a method of problem solving by using calling a function from itself
- Solve the same problem with smaller input
- Eventually need to solve a base case

The Factorial Problem: Loop

Formula:

$$n! = n * (n-1) * (n-2) * \dots * 2 * 1$$

```
1: #include <stdio.h>
2:
3: long factorial(int n) {
4:     int i;
5:     long fac = 1;
6:
7:     for (i=n; i>=1; i--)
8:         fac *= i;
9:     return fac;
10: }
11:
12: int main(void) {
13:     int n = 0;
14:
15:     printf("Enter n: ");
16:     scanf("%d", &n);
17:     printf("%d! = %ld\n", n, factorial(n));
18:     return 0;
}
```

The Factorial Problem: Recursion

Formula:

$$n! = n * (n-1) * (n-2) * \dots * 2 * 1$$

$$n! = n * (n-1)!$$

$$(n-1)! = (n-1) * (n-2)!$$

$$(n-2)! = (n-2) * (n-3)!$$

...

$$2! = 2 * 1!$$

$$1! = 1$$



Base Case



The Factorial Problem: Recursion

Formula:

$$n! = n * (n-1) * (n-2) * \dots * 2 * 1$$

```
1: #include <stdio.h>
2:
3: long factorial(int n) {
4:     if (n == 1)
5:         return 1;
6:     else
7:         return n * factorial(n-1);
8: }
9:
10: int main(void) {
11:     int n = 0;
12:
13:     printf("Enter n: ");
14:     scanf("%d", &n);
15:     printf("%d! = %ld\n", n, factorial(n));
16:     return 0;
17: }
```

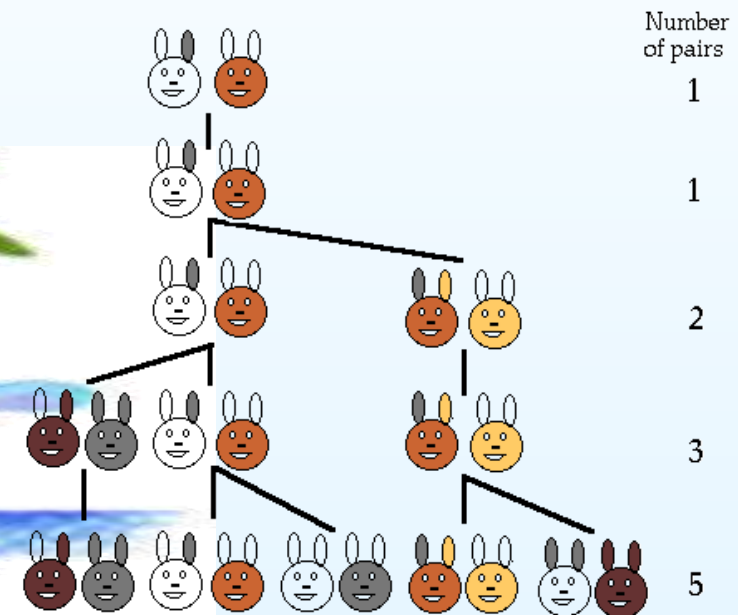
} Base Case

} Recursion Step



The Fibonacci Problem

- The rabbit tale
 - Begin with a pair of rabbits (male and female) in the islands
 - Rabbits can mate at the age of one month
 - At the second month, a female can produce another pair of rabbits
 - Suppose that rabbits never die



The Fibonacci Problem: Recursion

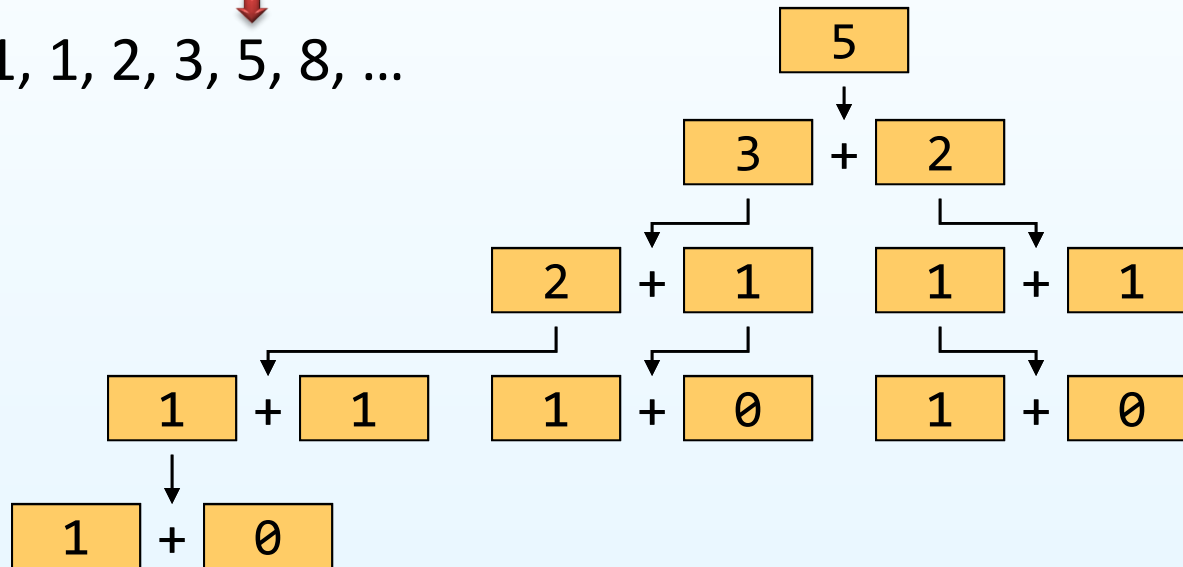
Formula:

$$\text{fib}_n = \text{fib}_{n-1} + \text{fib}_{n-2} \text{ where } \text{fib}_1 = 1, \text{fib}_0 = 0$$

Base Case

fib_5

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



The Fibonacci Problem: Recursion

Formula:

$$\text{fib}_n = \text{fib}_{n-1} + \text{fib}_{n-2} \text{ where } \text{fib}_1 = 1, \text{fib}_0 = 0$$

```
1: #include <stdio.h>
2:
3: long fibonacci(int n) {
4:     if (n == 0)
5:         return 0;
6:     else if (n == 1)
7:         return 1;
8:     else
9:         return fibonacci(n-1) + fibonacci(n-2);
10: }
11:
12: int main(void) {
13:     int n = 0;
14:
15:     printf("Enter n: ");
16:     scanf("%d", &n);
17:     printf("fib(%d) = %ld\n", n, fibonacci(n));
18:     return 0;
19: }
```

} Base Case

} Recursion Step

The Fibonacci Problem: Loop

Formula:

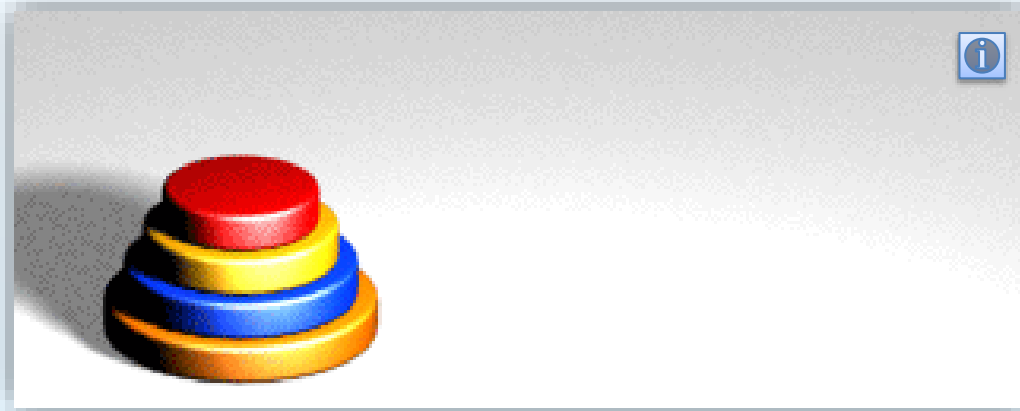
$$\text{fib}_n = \text{fib}_{n-1} + \text{fib}_{n-2} \text{ where } \text{fib}_1 = 1, \text{fib}_0 = 0$$

Please do it by yourself!

Towers of Hanoi (Brahma) - Lucas

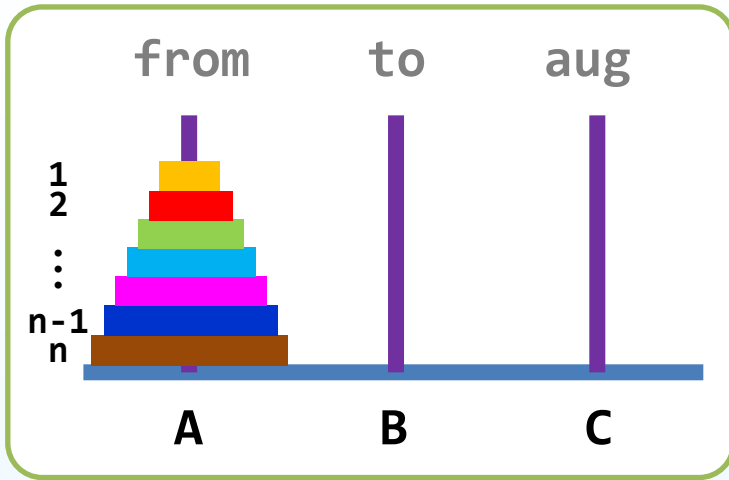
A mathematical game consists of three rods and a number of disks of different sizes. The puzzle starts with the disks in a stack in ascending order of size on one rod. The objective is to move the entire stack to another rod, obeying the following simple rules:

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
- No larger disk may be placed on top of a smaller disk.

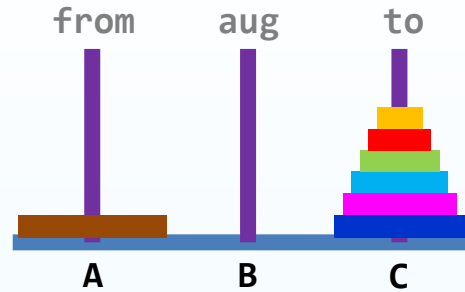


Source: https://en.wikipedia.org/wiki/Tower_of_Hanoi/

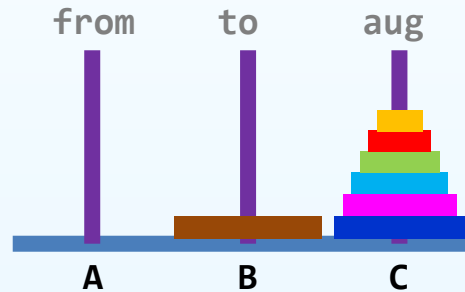
Towers of Hanoi: Recursion



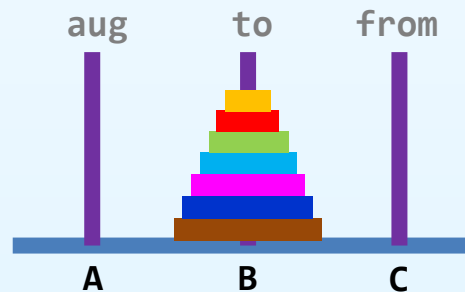
What is the
base case?



1. Move $1..(n-1)$: A \rightarrow C



2. Move n : A \rightarrow B

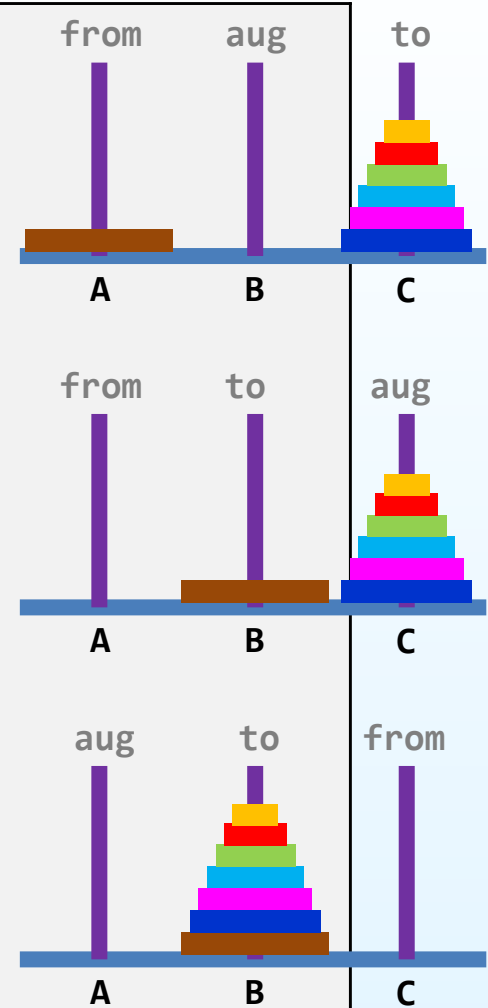


3. Move $1..(n-1)$: C \rightarrow B



Towers of Hanoi: Recursion

```
1: #include <stdio.h>
2:
3: void toh(int n, char from, char to, char aug) {
4:
5:
6:
7:
8:
9:
10:
11: }
12:
13: int main(void) {
14:     int n = 0;
15:
16:     printf("Enter n: ");
17:     scanf("%d", &n);
18:     toh(n, 'A', 'B', 'C');
19:     return 0;
20: }
```



Recursion vs. Loop

- Recursion
 - Easy to understand and implement
- Loop
 - Faster execution
 - Less memory allocation

Any Question?

Solution to Exercise 1

```
1: #include <stdio.h>
2:
3: int main(void) {
4:     double amount = 1000;
5:     double rate    = 0.05;
6:     int      year   = 10;
7:     int      i      = 0;
8:
9:     for (i=0; i<year; i++)
10:         amount += amount*rate;
11:     printf("The amount is $%.2lf\n", amount);
12:
13:     return 0;
14: }
```



Solution to Exercise 2

```
1: #include <stdio.h>
2:
3: int main(void) {
4:     int number = 0, lastdigit = 0, reverse = 0;
5:
6:     printf("Enter a number: ");
7:     scanf("%d", &number);
8:     while (number > 0) {
9:         lastdigit = number % 10;
10:        reverse = reverse * 10 + lastdigit;
11:        number = number / 10;
12:    }
13:    printf("%d\n", reverse);
14:
15:    return 0;
16: }
```



Solution to Exercise 3

```
1: #include <stdio.h>
2:
3: int main(void) {
4:     char c = '\0';
5:
6:     printf("Enter a message: ");
7:     while ((c = getchar()) != '\n')
8:         printf("%c", (c>='a') && (c<='z') ? c-'a'+ 'A' : c);
9:     printf("\n");
10:
11:     return 0;
12: }
```



Solution to Exercise 3 (do-while)

```
1: #include <stdio.h>
2:
3: int main(void) {
4:     char c = '\0';
5:
6:     printf("Enter a message: ");
7:     do {
8:         if ((c = getchar()) != '\n')
9:             printf("%c", (c>='a') && (c<='z') ? c-'a'+ 'A' : c);
10:    } while (c != '\n');
11:    printf("\n");
12:
13:    return 0;
14: }
```

