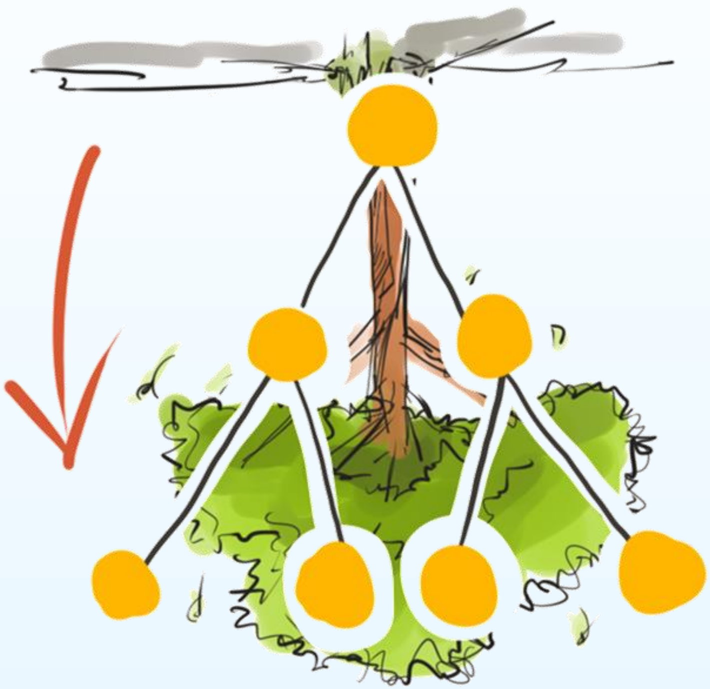


Lecture 2:
C Programming (Part II)

01204212 Abstract Data Types and Problem Solving

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

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

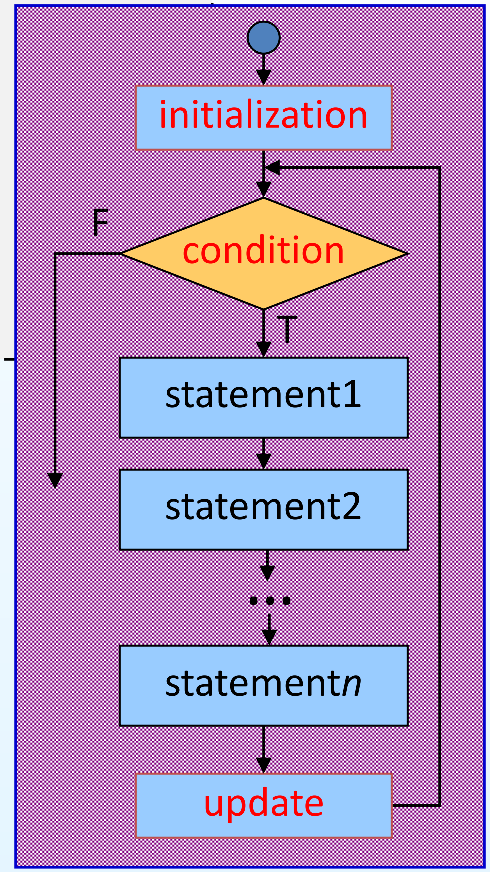
- A group of instructions that is executed repeatedly according to some conditions

- 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:

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

Code Fragment #2:

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

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

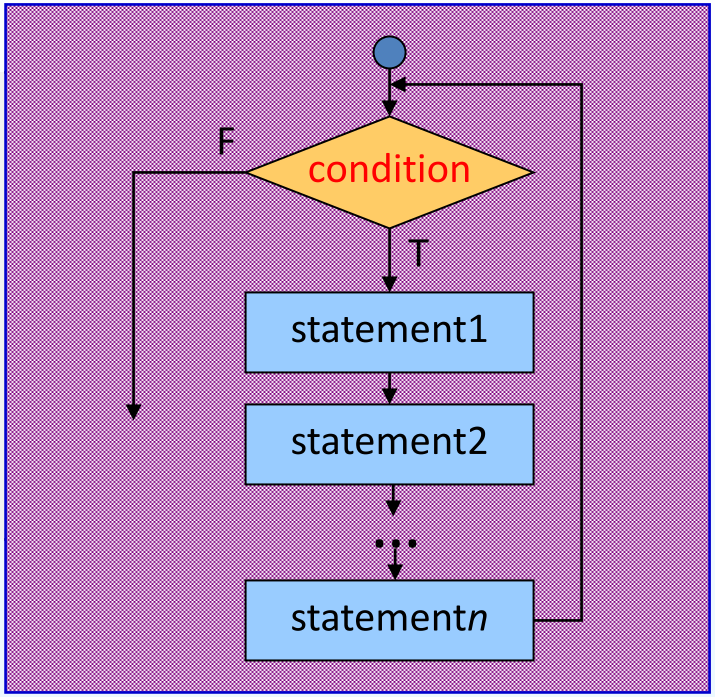
- However, both semicolons are always required

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

Loop: The `while` Statement

Syntax:

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

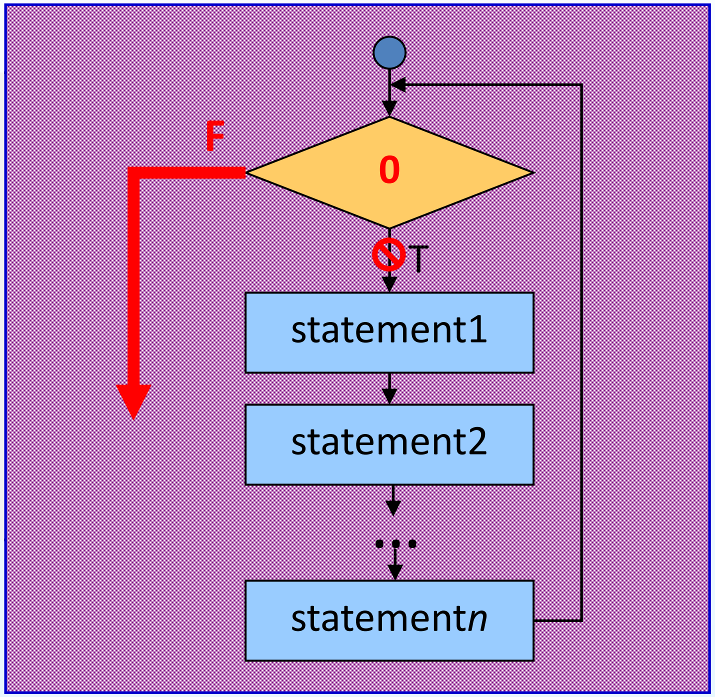


- Use the **`while`** keyword
- The condition is tested at the top of loop
- The loop statements are repeatedly executed if the condition is still True

while Statement: Zero Loop

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

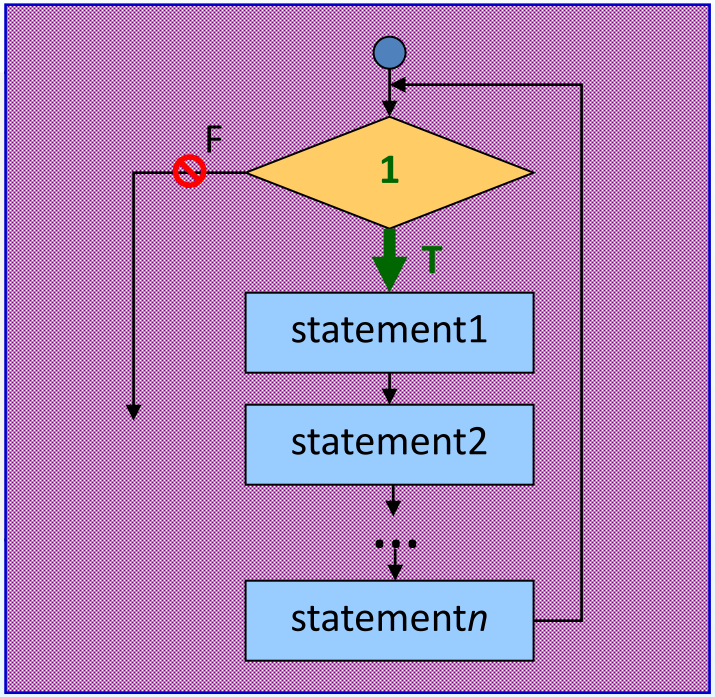
```
while (0)
{
   statement1;
   statement2;
   ...
   statementn;
}
```



`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

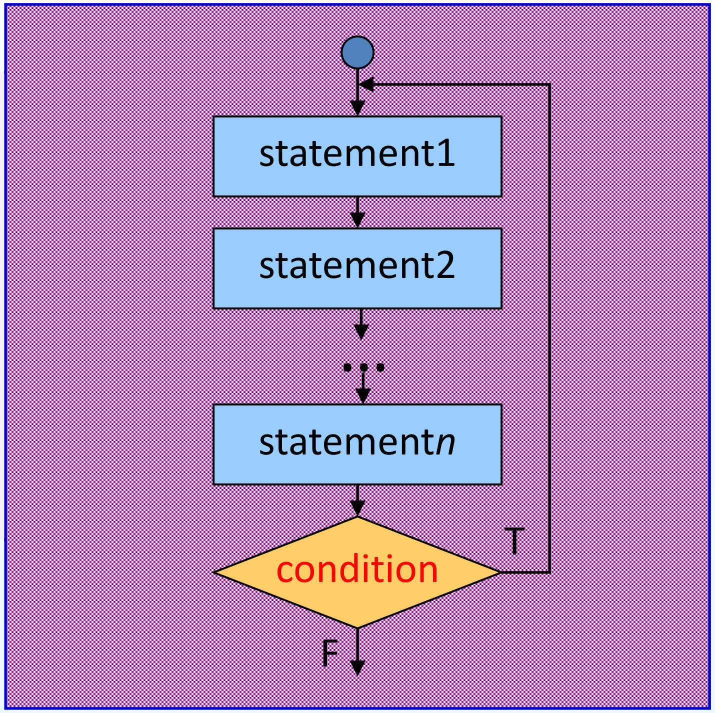
```
Enter a message: Hello world !!!
HELLO WORLD !!!
```

Hint: Try the `getchar()` function

Loop: The `do-while` Statement

Syntax:

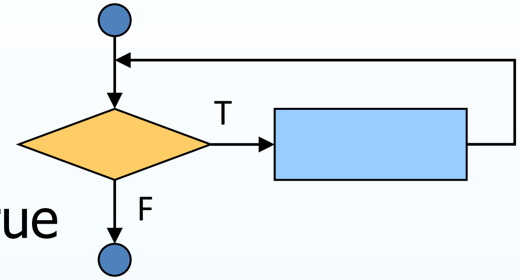
```
do
{
  statement1;
  statement2;
  ...
  statementn;
} while (condition);
```



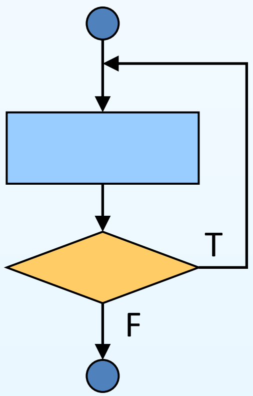
- Use the **do** and **while** keywords
- The condition is tested at the bottom of loop
- Require a semicolon ; at the end
- The loop statements are repeatedly executed if the condition is still True

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

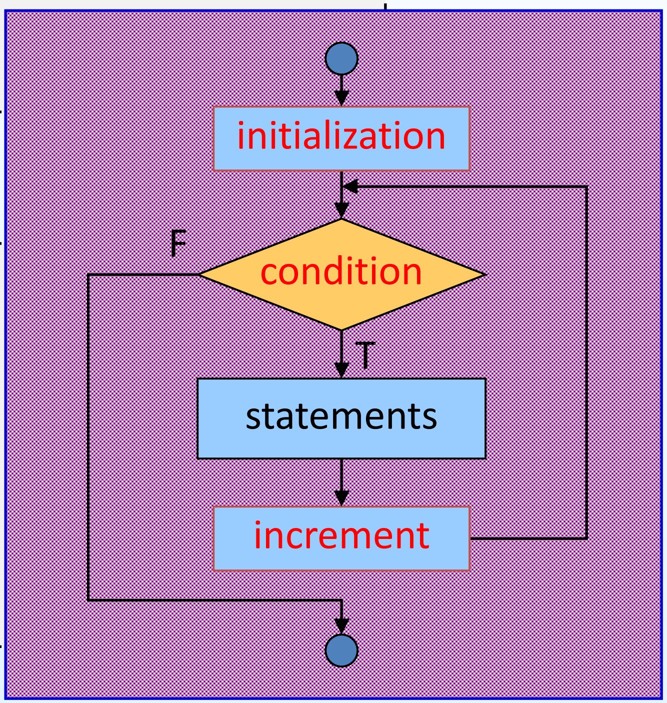
The `while` vs. `for` Statements

- 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:

```c
int count, sum = 0;
for (count=1; count<=10; count++) {
  if (count == 3)
    break;
  sum += count;
}
printf("%d\n", sum);
```

Code Fragment #2:

```c
int count, sum = 0;
for (count=1; count<=10; count++) {
  if (count == 3)
    continue;
  sum += count;
}
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

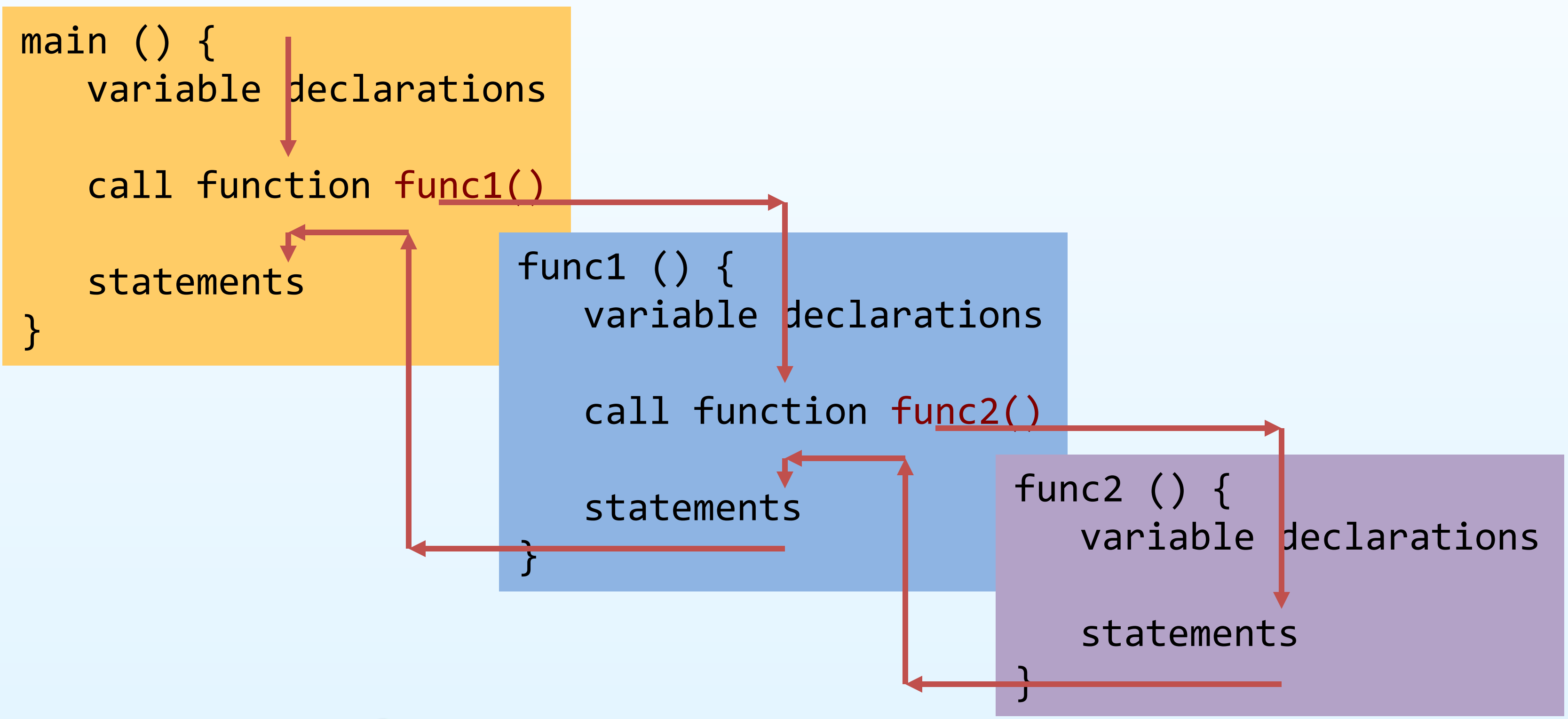
- Sequence
- Subroutine (function)
- Selection: `if`, `if-else`, `switch`
- Repetition: `for`, `while`, `do-while`

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

```c
#include <stdio.h>

int adding(int a, int b) {
  return a+b;
}

int main(void) {
  int a, b, sum;

  printf("Enter a and b: ");
  scanf("%d %d", &a, &b);
  sum = adding(a, b);
  printf("The sum is %d\n", sum);
  return 0;
}
```

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

```c
#include <stdio.h>

int adding(int a, int b);

int main(void) {
  int a, b, sum;

  printf("Enter a and b: ");
  scanf("%d %d", &a, &b);
  sum = adding(a, b);
  printf("The sum is %d\n", sum);
  return 0;
}

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

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>`

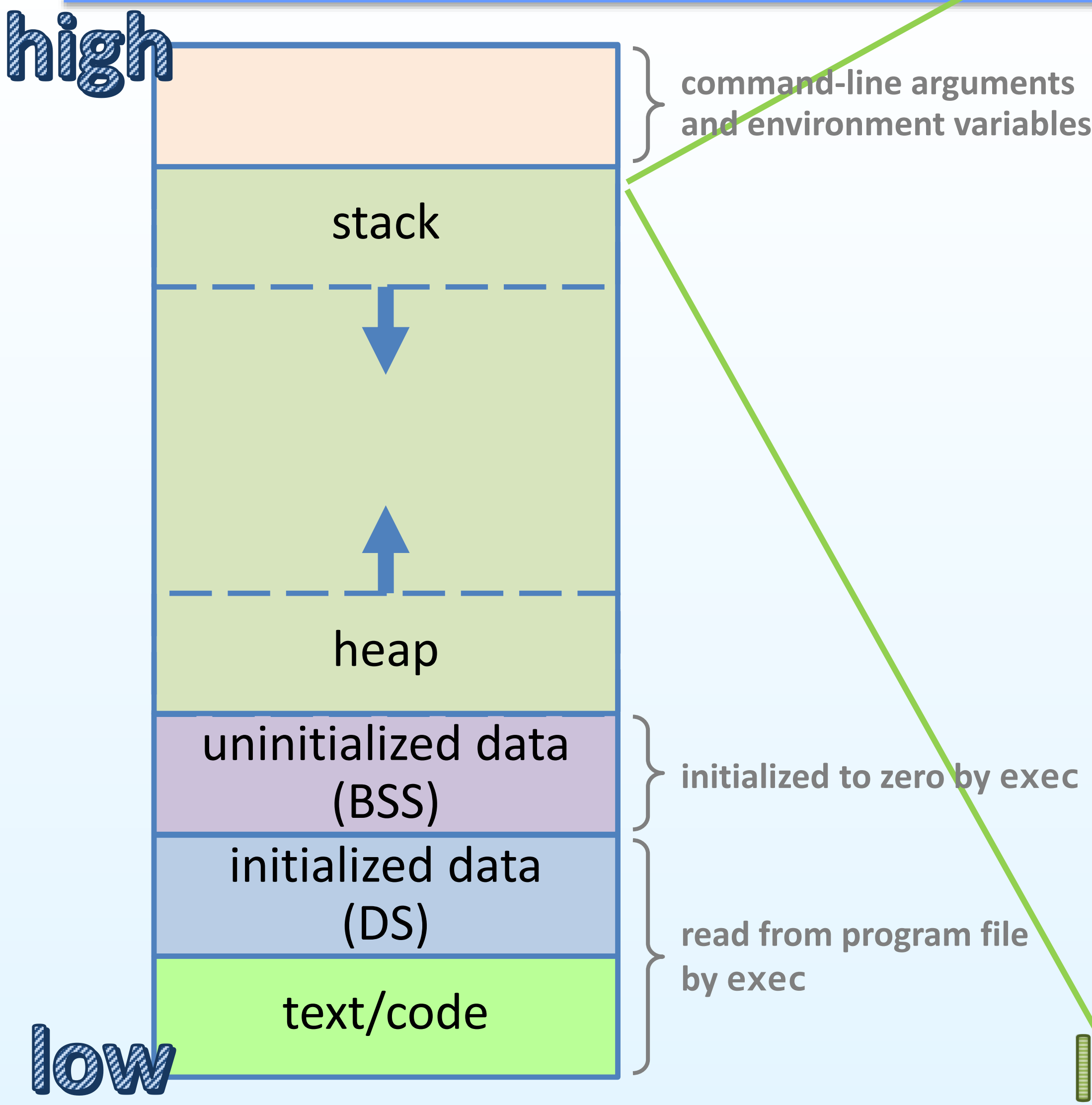
    `#include <math.h>`
- Programmers’ header files
  - Contain the defined function prototypes in `file.h`
  - e.g., `#include "file.h"`

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

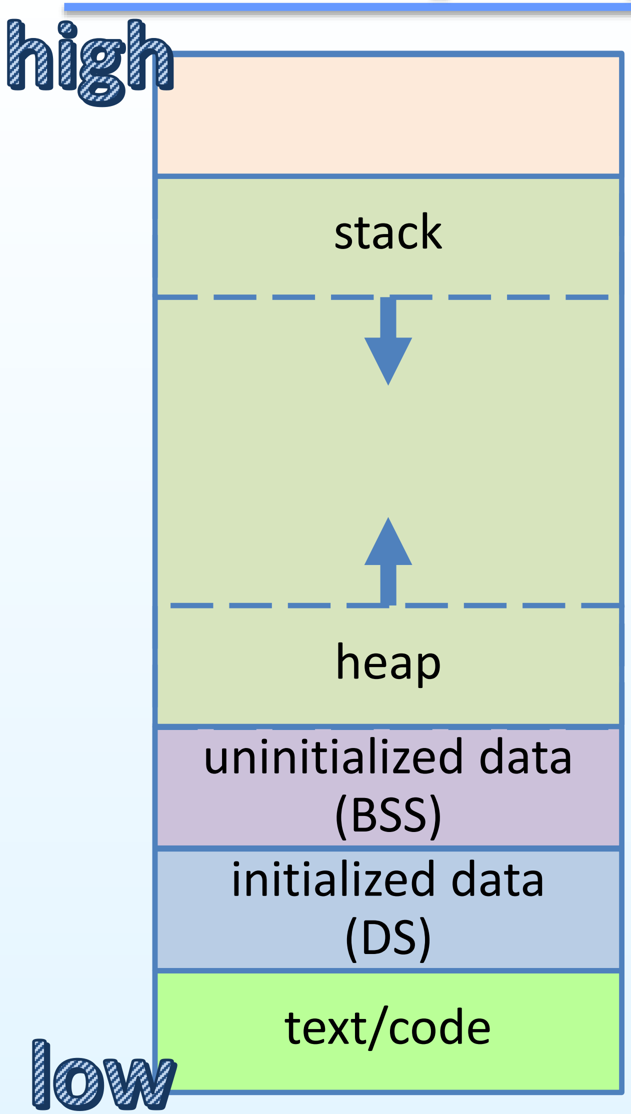
- Call by reference
  - Pass the reference of arguments to parameters
  - Changing the value of parameters effects the original

Memory Layout



Address	Value
high…	
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	
low …	

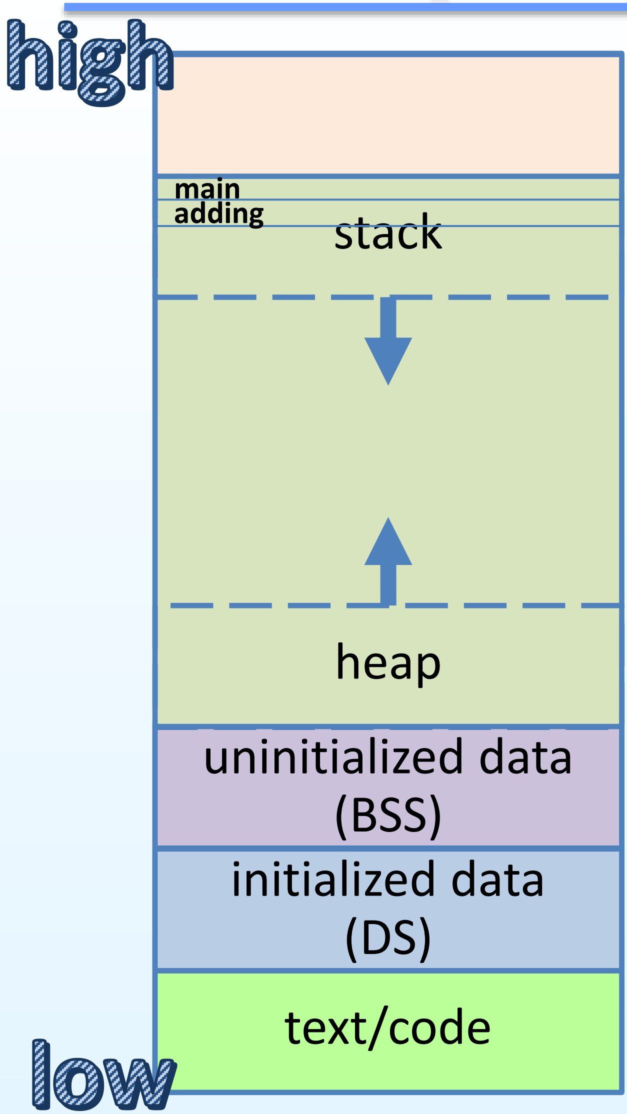
Memory Layout



- 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




```c
#include <stdio.h>

int a = 1, b;

int adding(int a, int b) {
  return a+b;
}

int main(void) {
  int sum;

  b = 2;
  sum = adding(a, b);
  printf("The sum is %d\n", sum);
  return 0;
}
```

Example: Calling Function by Value

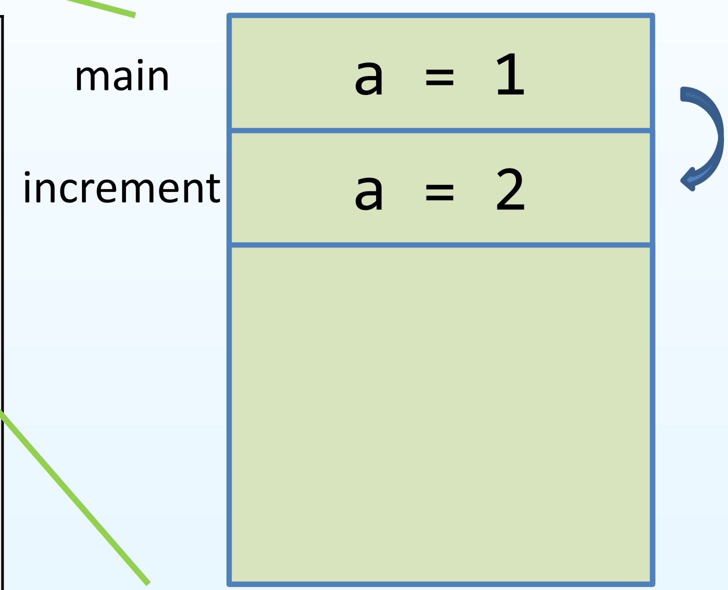
high

stack

What is the output of the following program?

```c
#include <stdio.h>

void increment(int a) {
  a = a + 1;
}

int main(void) {
  int a = 1;

  increment(a);
  printf("%d\n", a);
  return 0;
}
```

low



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

```c
#include <stdio.h>

int main(void) {
  register int i = 0;
  auto int sum = 0;

  for (i=1; i<=10000; i++)
    sum += i;
  printf("The sum is %d\n", sum);
  return 0;
}
```

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

```c
#include <stdio.h>

int increment() {
  static int x = 0;
  return ++x;
}

int main(void) {
  int i, sum = 0;

  for (i=1; i<=10000; i++)
    sum = increment();
  printf("The sum is %d\n", sum);
  return 0;
}
```

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) * ... * 2 * 1
```

```c
#include <stdio.h>

long factorial(int n) {
  int i;
  long fac = 1;

  for (i=n; i>=1; i--)
    fac *= i;
  return fac;
}

int main(void) {
  int n = 0;

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

The Factorial Problem: Recursion

Formula:

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

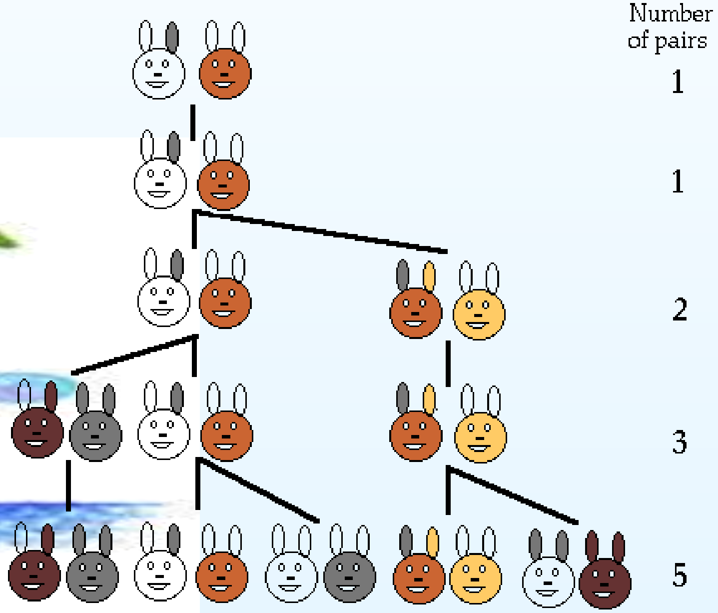
```c
#include <stdio.h>

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

int main(void) {
  int n = 0;

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

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

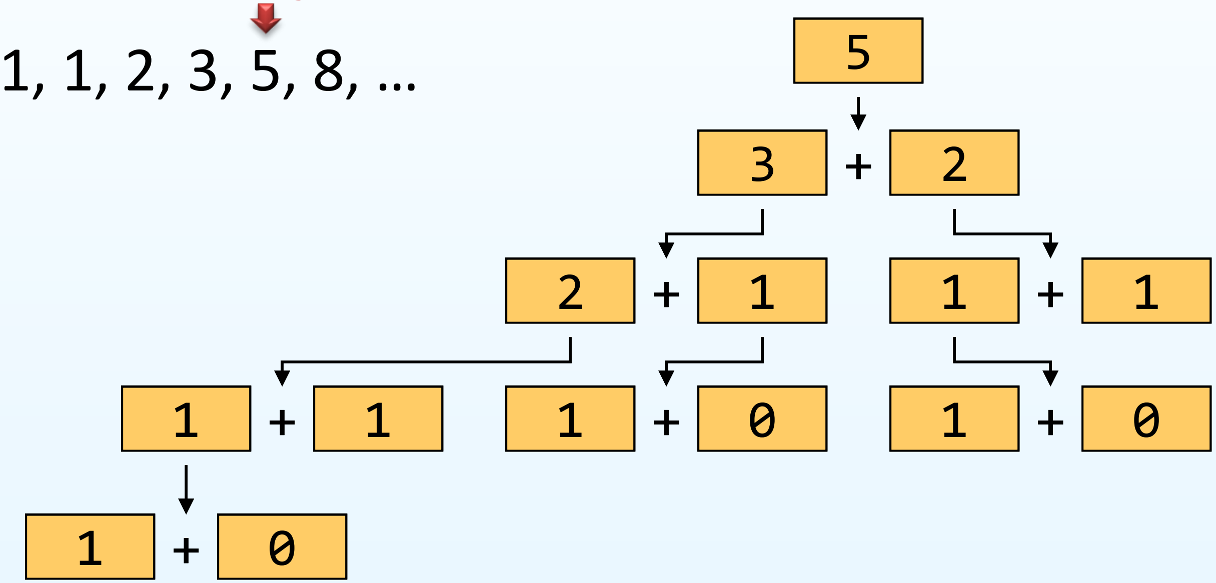
Formula:

$$fib_n = fib_{n-1} + fib_{n-2} \text{ where } fib_1 = 1,\ fib_0 = 0$$

Base Case

fib_5

series: 0, 1, 1, 2, 3, 5, 8, …



The Fibonacci Problem: Recursion

Formula:

$$fib_n = fib_{n-1} + fib_{n-2} \text{ where } fib_1 = 1,\ fib_0 = 0$$

```c
#include <stdio.h>

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

int main(void) {
  int n = 0;

  printf("Enter n: ");
  scanf("%d", &n);
  printf("fib(%d) = %ld\n", n, fibonacci(n));
  return 0;
}
```

Base Case

Recursion Step

The Fibonacci Problem: Loop

Formula:

$$fib_n = fib_{n-1} + fib_{n-2} \text{ where } fib_1 = 1,\ fib_0 = 0$$

Please do it by yourself!

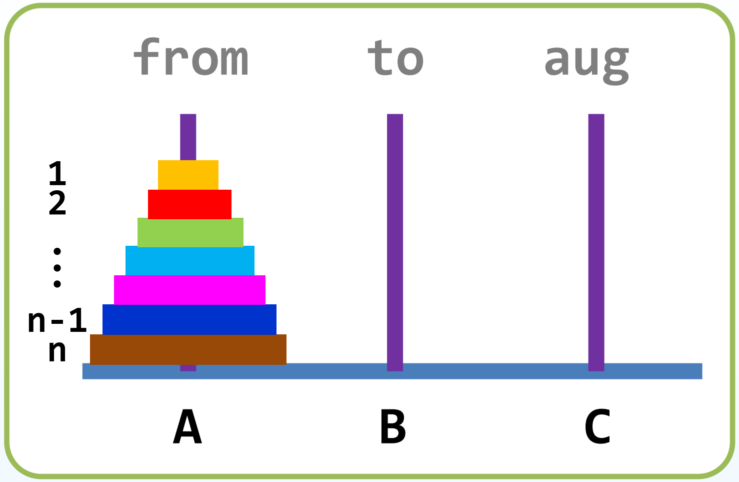
Towers of Hanoi (Brahma) - Lucas

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:

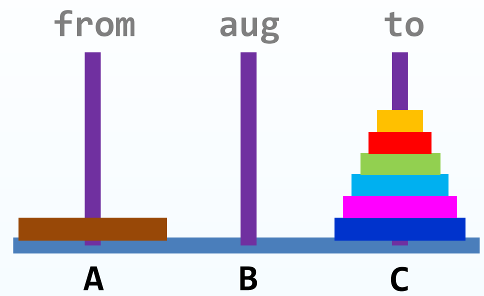
- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
- No larger disk may be placed on top of a smaller disk.

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

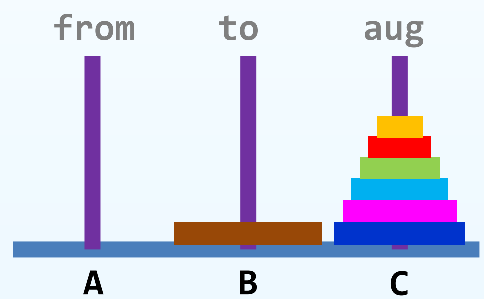
Towers of Hanoi: Recursion



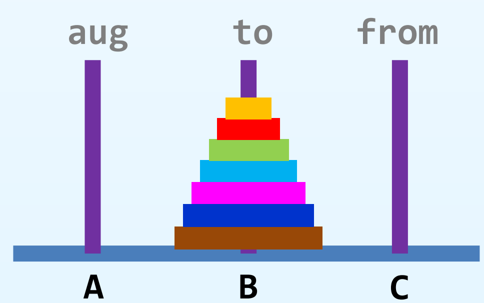
What is the base case?



1. Move **1..(n-1)**: **A -> C**



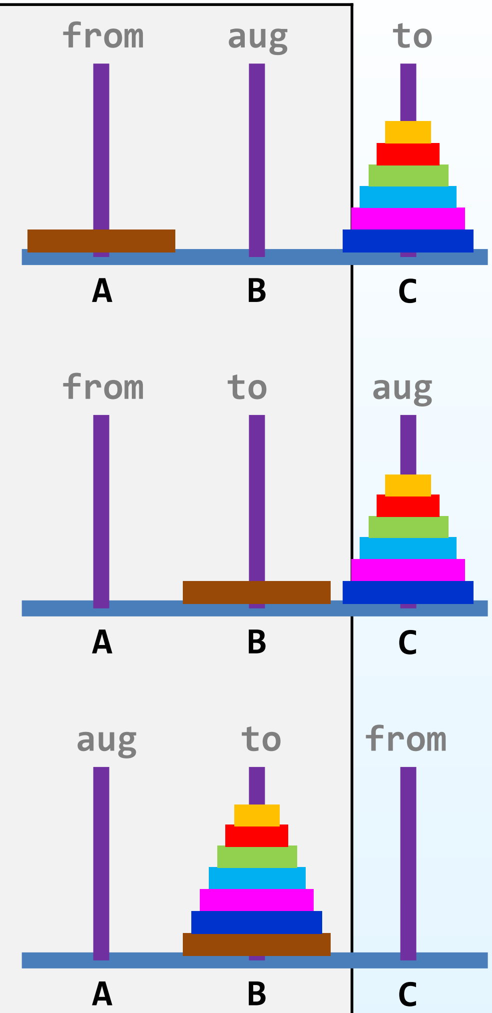
2. Move **n**: **A -> B**



3. Move **1..(n-1)**: **C -> B**

Towers of Hanoi: Recursion

```c
#include <stdio.h>

void toh(int n, char from, char to, char aug) {







}

int main(void) {
  int n = 0;

  printf("Enter n: ");
  scanf("%d", &n);
  toh(n, 'A', 'B', 'C');
  return 0;
}
```



Recursion vs. Loop

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

Any Question?