Lab 1: C/C++ Pointers and time.h

# EPL231 – Data Structures and Algorithms

# TODO

- Send email to totis@cs.ucy.ac.cy
- Subject: EPL231-Registration
- Body:
  - Surname Firstname
  - ID
  - Group A or B?

# Guidelines

- Code Guidelines:
  - Clean Code
  - Comments
  - Error control
  - Don't Overcomplicate
  - Make it work, THEN optimize
- Trouble?
  - Email
  - Room 122, Tuesday and Thursday mornings
- 10% of Grade - Programming Exercises

# C/C++ Pointers

# Pointers

- Pointer: a variable that stores the **memory address** of another variable
- Why:
  - Direct Memory Access
  - Speed
  - Memory
  - Handling of complex data types:
    - Strings, Arrays

# Pointers - Syntax

- Syntax:
  - `int *p1;`
  - `float *p2;`
  - `char **p3;`
  - `struct myStruct *p4;`
  - `void *p5;`

# Pointers - Operators

- Operators:
  - "Address of" operator:
    - &var: Returns the memory address of variable ***var***
  - "Indirection" operator (Dereference a pointer):
    - *ptr: Access the memory contents at the address pointed by pointer ***ptr***.
    - Can we dereference all pointers?
      - NO: think of void pointers!
      - Why: Every variable occupies different amount of memory.

# Pointers - Arithmetic

- Addition:
  - `Type *ptr = 1000;`
  - `Type *ptr2 = ptr + 10;`
    - → `ptr2 = ptr + 10 * sizeof(Type)`
- Subtraction:
  - `Type *ptr = 1000;`
  - `Type *ptr2 = ptr - 10;`
    - → `ptr2 = ptr - 10 * sizeof(Type)`
- Valid for every type except void.

# Dynamic Memory Allocation

- malloc, calloc, etc./free
- new / delete

# Pointers - Example

```
int main() {
  int x = 23;
  float y = 1.33f;
  int *p_x;
  p_x = &x;
}
```

| Address | Data |
|---|---|
| **1000** x | 23 |
| **1004** y | 1.33f |
| **1008** p_x | 1000 |
| **100C** | |
| **100F** | |

# Pointers – Function Calls

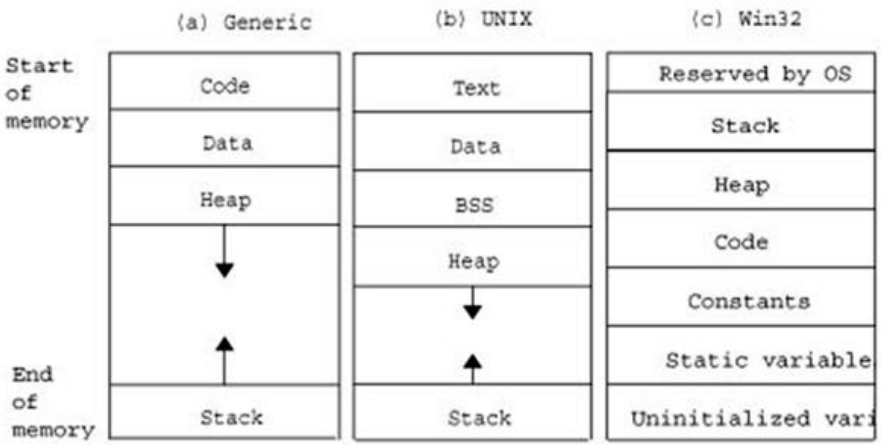
```
#include <stdio.h>
#include <stdlib.h>

void swap_1(int x, int y);
void swap_2(int *x, int *y);

void swap_1(int x, int y)
{
    int tmp = y;
    y = x;
    x = tmp;

    fprintf(stdout, "swap_1() called\n");
}

void swap_2(int *x, int *y)
{
    int tmp = *y;
    *y = *x;
    *x = tmp;

    fprintf(stdout, "swap_2() called\n");
}
```

```
int main()
{
    int x = 3;
    int y = 10;

    fprintf(stdout, "x = %d, y = %d\n", x, y);
    swap_1(x, y);
    fprintf(stdout, "x = %d, y = %d\n", x, y);
    swap_2(&x, &y);
    fprintf(stdout, "x = %d, y = %d\n", x, y);

    return 0;
}
```

x = 3, y = 10
swap_1() called
x = 3, y = 10
swap_2() called
x = 10, y = 3

# Process Memory

**PROCESS MEMORY ORGANIZATION**




**SEGMENTS:**

- Code, Text: The binary code
- Data: Global variables
  - BSS: Static Variables
- Heap: Dynamically allocated memory
- Stack: Local storage, function calls, function parameters, return address

# What is the difference?

```
int array_1[100];
int *array_2 = (int *)malloc(100 * sizeof(int));
```

- Both can be accessed using array notation:

```
array_1[5] = 20;
array_2[12] = 23;
```

- Both can be accessed using pointer notation:

```
*(array_1 + 5) = 20;
*(array_2 + 12) = 23;
```

- array_1 is stored in the Stack → Local Storage → Automatically destroyed after leaving scope
- array_2 is stored in the Heap → Dynamic Storage → HAS to be freed explicitly:
  - `free(array_2);`

# Timing your algorithms: time.h

# Data Types

- `#include <time.h>`
- **`clock_t`**:
  - Used for counting clock ticks.
  - Data type returned by **clock**().
- **`time_t`**:
  - Used for counting seconds.
  - Data type returned by **time**().
- **`struct tm`**:
  - A non-linear, broken-down calendar representation of time.

# Important Functions

- `clock_t` **`clock`** `(void);`
  - Returns the number of clock ticks elapsed since the program was launched.
  - We can get seconds if we divide this with CLOCKS_PER_SEC (in some systems CLK_PER_SEC)
- `time_t` **`time`** `(time_t *timer);`
  - Returns the number of seconds that elapsed since 00:00, 01/01/1970

# Algorithm Run Time

```
// Variable declarations
clock_t start = 0;
clock_t finish = 0;
float duration = 0.0f;

// Time our algorithm
start = clock();
runMyAlgorithm();
finish = clock();

// Duration of algorithm in seconds
duration = (float)(finish - start) / CLOCKS_PER_SEC;

// Print results
printf("Start:%u\n", start);
printf("Finish:%u\n", finish);
printf( "Duration:%5.1f seconds\n", duration);
printf( "Duration:%5.0f miliseconds\n", duration*1000);
```

# Time a real scenario

# Linear Search

- *Δεδομένα Εισόδου*: Πίνακας X με n στοιχεία, ταξινομημένος από το μικρότερο στο μεγαλύτερο, και ακέραιος k.
- *Στόχος*: Να εξακριβώσουμε αν το k είναι στοιχείο του X.
- *Γραμμική Διερεύνηση*: εξερευνούμε τον πίνακα από τα αριστερά στα δεξιά.

```
int linear( int X[], int n, int k){
 int i=0;
 while ( i < n )
   if (X[i] == k) return i;
   if (X[i] > k)  return -1;
   i++;
 return -1;
}
```

- *Χρόνος εκτέλεσης*: Εξαρτάται από το που (και αν) ο k βρίσκεται στον X[n].
- *Χείριστη περίπτωση*: O(n)

# Binary Search

- *Δυαδική Διερεύνηση:* βρίσκουμε το μέσο του πίνακα και αποφασίζουμε αν το k ανήκει στο δεξιό ή το αριστερό μισό. Επαναλαμβάνουμε την ίδια διαδικασία στο "μισό" που μας ενδιαφέρει.

```
int binary( int X[],int n,int k){
 int low = 0, high = n-1;
 int mid;
 while ( low < high ){
   mid = (high + low)/2;
   if (X[mid] < k) low = mid + 1;
   else
   if (X[mid] > k) high = mid - 1;
   else return mid;
  }
 return -1;
}
```

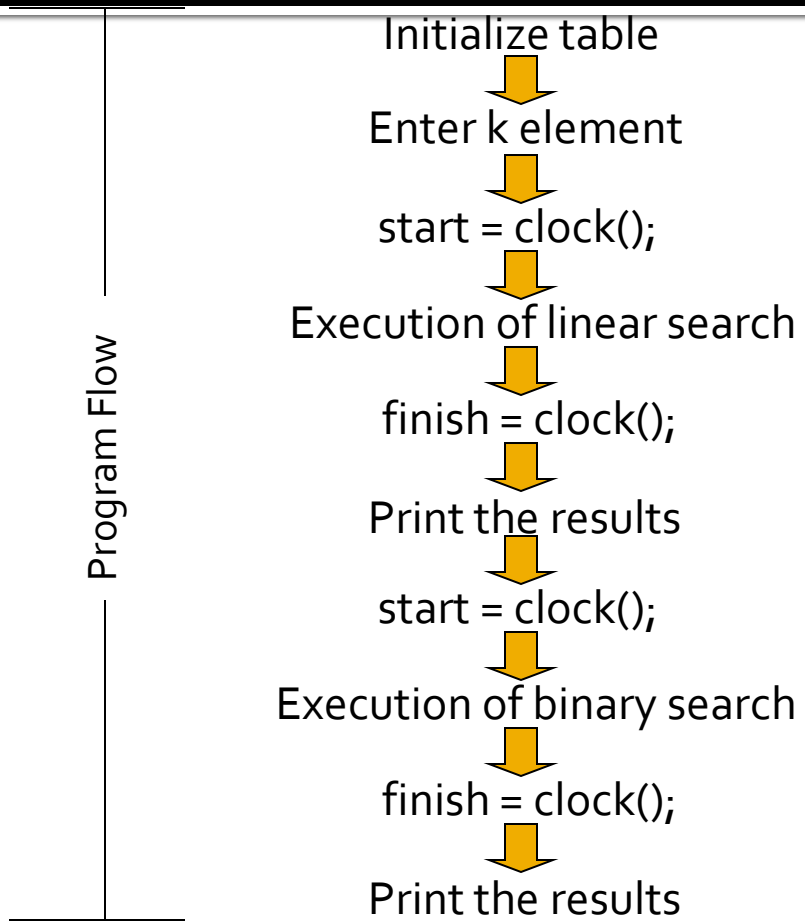
O(lg(n))

- *Χρόνος εκτέλεσης*;

# Let's start the implementation

- Define a variable MAX=1000
- Create a table of size MAX
- Initialize table from 0-(MAX-1)
- Implement both linear and binary search
- Create a menu for entering the k-element

# Program Flow



# Tryouts

- k=
  - 1001
  - 500
  - 2
  - Change MAX to 1.000.000 or more!
- Compare and explain the results

# Thanks

Questions?

totis@cs.ucy.ac.cy