# ENGN2219/COMP6719
# Computer Systems & Organization

Convener: Shoaib Akram

shoaib.akram@anu.edu.au

Australian National University

# Pointers: Example

- Guess the output of the `printf()` statements

```
int A = 19;
int B = 1;
int C = 8;
int D = 17;
....
int *P = &B;
char *Q = &B;
// Both P and Q contain
   00000004
printf("%i\n",*P); ??
printf("%i\n",*Q); ??
```

| Address | Data | Variable |
|---|---|---|
| ⋮ | ⋮ | ⋮ |
| 00000010 | 00000004 | P |
| 0000000C | 17 | D |
| 00000008 | 8 | C |
| 00000004 | 1 | B |
| 00000000 | 19 | A |

4 Bytes

# Answer

- `printf("%i\n",*P);` Output is always 1
- `printf("%i\n",*Q);` Big Endian: 0, Little Endian: 1

```
int A = 19;
int B = 1;
int C = 8;
int D = 17;
....
int *P = &B;
char *Q = &B;
// Both P and Q contain
   00000004
printf("%i\n",*P); ??
printf("%i\n",*Q); ??
```

| Address | Data | Variable |
|---|---|---|
| ⋮ | ⋮ | ⋮ |
| 00000010 | 00000004 | P |
| 0000000C | 17 | D |
| 00000008 | 8 | C |
| 00000004 | 1 | B |
| 00000000 | 19 | A |

4 Bytes

# malloc and free

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

void useless_func() {
    int *array1 = malloc(10 * sizeof(int));
    for (int i = 0; i < 10; i++)
       array1[i] = i * i;
    printf("%i\n", sizeof(array1));
    free(array1);
    printf("Done ....\n");
    return;
}
```

include for malloc()

use sizeof(int) operator rather than guessing how big is an int on a machine

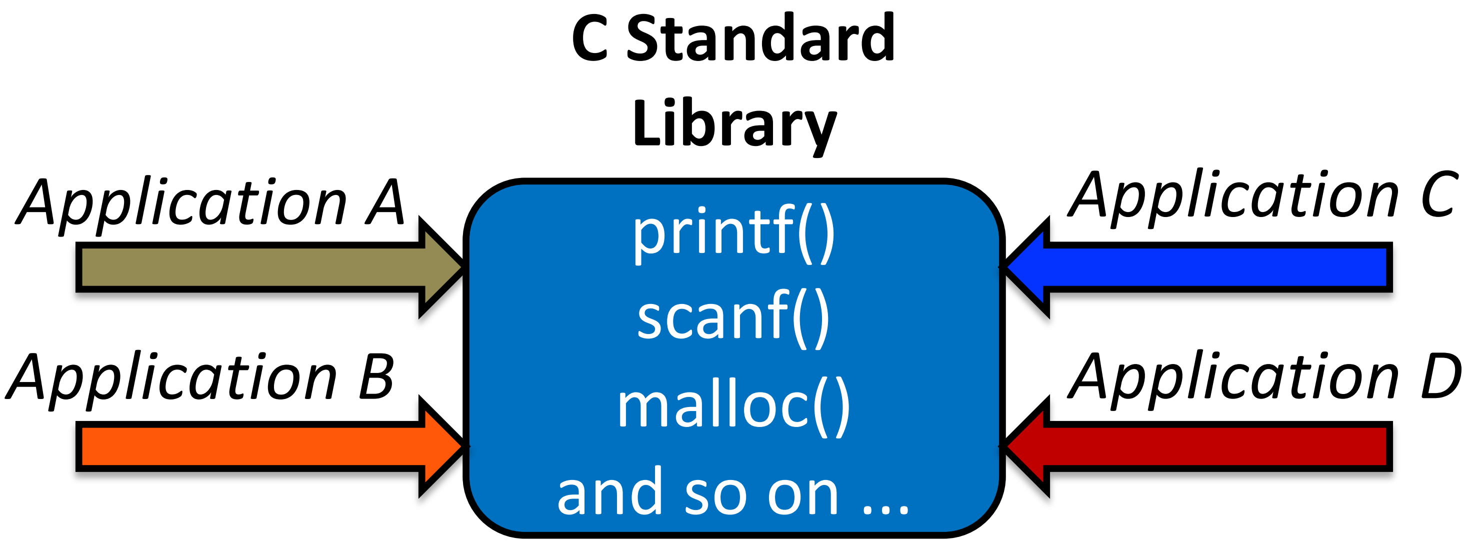
16 or 32 or 64 depending on the machine architecture

# API

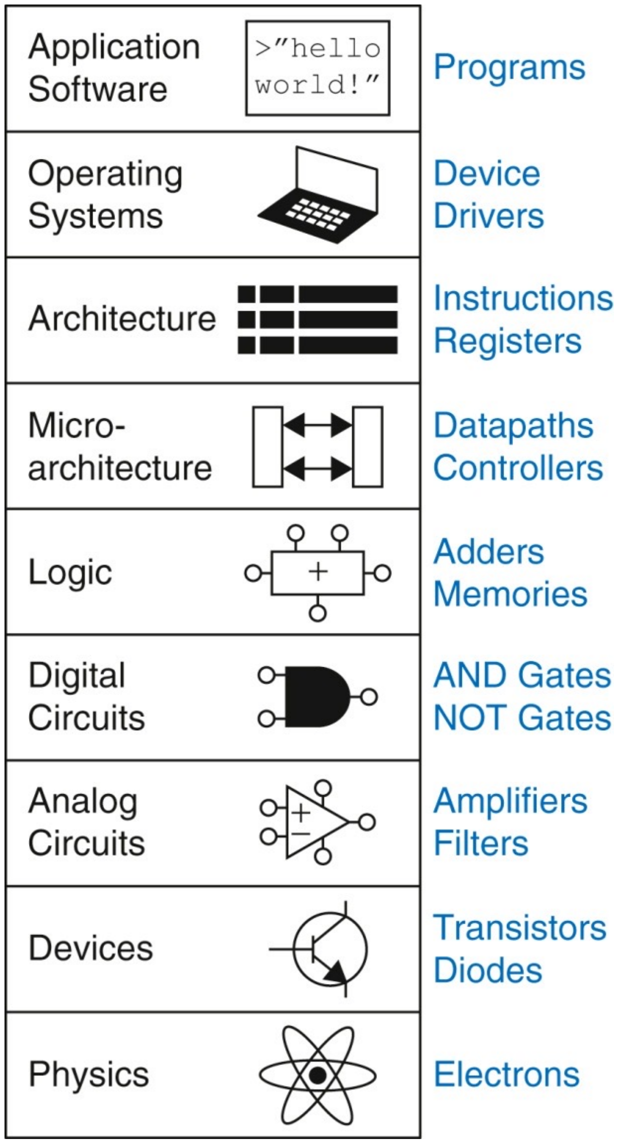
- *Application Programming Interface (API)*
  - Defines the interfaces by which one software program communicates with another at the source code level
  - **Abstraction:** API provides a standard set of interfaces that many different users (writing programs) can invoke
- *API defines the interface only*
  - The user of the API can ignore the implementation
  - Many implementations of an API can exist

# API: Example

- The C standard library (`libc`) defines a family of basic and essential functions: memory mgmt. and string manipulation




- `libc` hides a lot of operating system details by interacting with the OS on behalf of the programmer


Application Software
>"hello world!"
Programs
Operating Systems
Device Drivers
Architecture
Instructions Registers
Micro-architecture
Datapaths Controllers
Logic
+
Adders Memories
Digital Circuits
AND Gates NOT Gates
Analog Circuits
+
−
Amplifiers Filters
Devices
Transistors Diodes
Physics
Electrons

# ABI

- *Application Binary Interface*
  - Defines the binary interface b/w two or more pieces of software on a particular architecture
- *ABI ensure binary compatibility*
  - Calling convention, byte ordering, register use, linking, binary object format
  - Enforced by the toolchain (not a programmer's worry)
- ABI is a function of both the architecture (x86, ARM) and operating system (Linux, Windows)

# Common Memory Errors

- *Forgetting to allocate memory*

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

void useless_func() {
    char *src = "Hello!";
    char *dst;
    strcpy(dst, src);
}
```

- *Causes a segmentation fault*
  - *System is angry at you, because you did something wrong with memory*

### Description

The C library function **char *strcpy(char *dest, const char *src)** copies the string pointed to, by **src** to **dest**.

### Declaration

Following is the declaration for strcpy() function.

```
char *strcpy(char *dest, const char *src)
```

### Parameters

- **dest** − This is the pointer to the destination array where the content is to be copied.
- **src** − This is the string to be copied.

# Common Memory Errors

- *Forgetting to allocate memory*

```
//Fixed program

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

void useless_func() {
   char *src = "Hello!";
   char *dst = (char*) malloc(strlen(src) + 1);
   strcpy(dst, src);
}
```

# Common Memory Errors

- *Not allocating sufficient memory (a.k.a. buffer overflow)*

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

void useless_func() {
    char *src = "Hello!";
    char *dst = (char*) malloc(strlen(src));
    strcpy(dst, src);
}
```

# Common Memory Errors

- *Not allocating sufficient memory (a.k.a. buffer overflow)*
    - There maybe an unused variable at the end of allocated memory (no harm)
    - Memory manager might have allocated a little extra space
    - Fault and crash and security vulnerability is likely

- *Just because a program runs, does not mean it is correct*

# Common Memory Errors

- *Forgetting to initialize allocated memory*
  - Zero initialization is important for avoiding hard to debug problems

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

void useless_func() {
    int *n = (int*) malloc(sizeof(int));
    ...
    cond = *n;
    ...
    if (cond == 0) {
        ...;
    }
}
```

# Common Memory Errors

- *Forgetting to free memory (memory leak)*
  - Huge problem in long-running programs
- When the process exits, the operating system automatically reclaims all allocated memory by the process
  - Still a bad habit to not call `free()` in short-running programs

# Example

- Memory leak example (from last lecture)

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

void useless_func() {
    int *array1 = malloc(10 * sizeof(int));
    for (int i = 0; i < 10; i++)
       array1[i] = i * i;
    array1 = malloc(5 * sizeof(int));
    for (int i = 0; i < 10; i++)
       array1[i] = i * i;
    free(array1);
    printf("Done ....\n");
    return;
}
```

memory leak
Original 10-element array still on heap (address is gone, not saved)

out of bounds

# Common Memory Errors

- *Freeing memory before the program is done with it*
  - Known as a dangling pointer
  - The subsequent use can crash the program

```
// example 1
void useless_func() {
    int n = 100;
    int *ptr = (int*) malloc(sizeof(int));
    if (n == 100) {
        int v = 10;
        ptr = &v;
    }
    // ptr is now a dangling pointer
}
```

# Common Memory Errors

- *Freeing memory before the program is done with it*
  - Known as a dangling pointer
  - The subsequent use can crash the program

```
// example 2
void useless_func() {
    int *ptr = (int*) malloc(sizeof(int));
    ...
    ...
    free(ptr);
    ...
    ...
    *ptr = 2; // ptr is now a dangling pointer
}
```

# Common Memory Errors

- *Freeing memory before the program is done with it*
  - Known as a dangling pointer
  - The subsequent use can crash the program

```
//example 2 fix, you will now get a runtime error
void useless_func() {
    int *ptr = (int*) malloc(sizeof(int));
    ...
    ...
    free(ptr);
    ptr = NULL; // good practice
    ...
    ...
    *ptr = 2; // ptr is no longer a dangling pointer
}
```

# Common Memory Errors

- *Freeing memory before the program is done with it*
  - Known as a dangling pointer
  - The subsequent use can crash the program

```
// example 3
int *useless_func() {
   int x = 100;
   return &x;
}

// when the function returns, it's stack is
   deallocated, and we must not use the address
   returned by useless_func()
```

# Common Memory Errors

- *Freeing memory multiple times*
  - double free (typical name)

```
void useless_func() {
    int *x = (int *) malloc(100 * sizeof(int));
    free(x);
    ...
    free(x); // confuses the memory manager
}
```

# Common Memory Errors

- *Incorrectly calling `free()`*
  - Avoid these so-called **invalid frees**

```
void useless_func() {
    int *x = (int *) malloc(100 * sizeof(int));
    ...
    x = x + 4;
    ...
    free(x);
}
```

# Multi-Dimensional Arrays

- *Statically allocated arrays and stack-allocated 2-dimensional arrays are simple*

```
#define R 5
#define C 4
int matrix[R][C];

for (int i = 0; i < R; i++) {
    for (int j = 0; j < C; j++) {
        matrix[i][j] = i + j;
    }
}
```

| | | | |
|---|---|---|---|
| [0][0] | [0][1] | [0][2] | [0][3] |
| [1][0] | [1][1] | [1][2] | [1][3] |
| [2][0] | [2][1] | [2][2] | [2][3] |
| [3][0] | [3][1] | [3][2] | [3][3] |
| [4][0] | [4][1] | [4][2] | [4][3] |

# Multi-Dimensional Arrays

- *Dynamically allocated 2-dimensional arrays*

```
#define R 5
#define C 4

int **matrix;

void useless_func() {
   matrix = (int **) malloc(R * sizeof(int*));
   for (int i = 0; i < R; i++) {
      matrix[i] = malloc(C * sizeof(int));
   }
}
```

# Multi-Dimensional Arrays

- *You can have jagged arrays, where each row has a different # columns*

```
#define R 3

int **matrix;

void useless_func() {
   matrix = (int **) malloc(R * sizeof(int*));
   matrix[0] = malloc(2 * sizeof(int));
   matrix[1] = malloc(5 * sizeof(int));
   matrix[2] = malloc(3 * sizeof(int));
}
```

# Structs

- Lab 10 handout introduces structs in C
- How can we create a dynamically allocated array of structs?

```
#define TOTAL 100

struct student {
    char name[16];
    int id;
};

typedef struct student student_t;

student_t *students;
void useless_func() {
    students = (student_t *) malloc(TOTAL * sizeof(student_t));
    students[0].name = “Shane”;
    students[0].id = 10;
}
```

can access members with . operator

# Structs and –> Operator

- If we have a pointer to a struct, we can use the arrow operator ( –> ) to access the members of a struct

```
struct student {
    char name[16];
    int id;
};

typedef struct student student_t;

student_t *student;
void useless_func() {
    student = (student_t *) malloc(sizeof(student_t));
    student->name = “Shane”;
    students->id = 10;
}
```

# Enumerations in C

- Enumeration (or enum) is a user defined data type in C
  - Used to assign names to integral constants (code readability)

```c
// An example program to demonstrate working
// of enum in C
#include<stdio.h>

enum week{Mon, Tue, Wed, Thur, Fri, Sat,
Sun};

int main()
{
    enum week day;
    day = Wed;
    printf("%i",day);
    return 0;
}
```

# Principle of Locality

- Programs tend to reference (access) data items (words) that:
  - Reside near other recently accessed items
  - Were recently accessed themselves
- Temporal Locality
  - A memory location that is referenced once is likely to be referenced again multiple times soon
- Spatial Locality
  - If a memory location is referenced once, then a nearby location is likely to be referenced soon
- Generally, programs with good locality are likely to run faster than programs with poor locality

# Locality is Everywhere

- Computer designers speed up main memory accesses by keeping most recently accessed data items in small fast memories called cache memories
  - Main memory is slow but high capacity (DRAM technology uses capacitors)
  - Cache is fast but low capacity (SRAM uses cross-coupled inverters)
- Web browsers exploit temporal locality by caching recently referenced documents on disks
- One of the enduring ideas in computer systems

# Example

- Does the program below exhibit good locality?

```
int sumarray(int a[n]) {
   int i;
   int sum = 0;
   for (i = 0; i < n; i++)
      sum = sum + a[i];
   return sum;
}
```

| Address | 0 | 4 | 8 | 12 | 16 | 20 | 24 | 28 |
|---|---|---|---|---|---|---|---|---|
| Contents | a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] |
| Access Order | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

# Example

- Does the program below exhibit good locality?

```
int sumvec(int a[n]) {
   int i;
   int sum = 0;
   for (i = 0; i < n; i++)
      sum = sum + a[i];
   return sum;
}
```

| Variable | Spatial | Temporal |
|---|---|---|
| sum | Poor | Good |
| a | Good | Poor |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Address | 0 | 4 | 8 | 12 | 16 | 20 | 24 | 28 |
| Contents | a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] |
| Access Order | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

# Explanation

- `array a`
  - Each element is accessed only once
  - Neighboring element is accessed soon in the next loop iteration
- `sum`
  - `sum` is a scalar, so no spatial locality
  - `sum` has good temporal locality, as it is accessed in each iteration

# Stride

- Visiting each element of an array sequentially is an example of a *stride-1* reference pattern
  - Stride-1 reference pattern is called sequential access pattern
- Visiting every k-th element of a contiguous array is a *stride-k* reference pattern
- As stride increases, the spatial locality decreases
- Sequential accesses are generally highly desirable

# Row Major Order

- C arrays are laid out in memory row-wise
  - The entire row is stored contiguously (elements are next to each other, 0, 4, 8 ....0)
- $a_{mn}$: m is row, and n is column

Row 1

| Address | 0 | 4 | 8 | 12 | 16 | 20 |
|---|---|---|---|---|---|---|
| Contents | $a_{00}$ | $a_{01}$ | $a_{02}$ | $a_{10}$ | $a_{11}$ | $a_{12}$ |
| Access Order | 1 | 2 | 3 | 4 | 5 | 6 |

# Example

- Sum the elements of the array in row-major order

```
int sumarrayrows(int a[M][N]) {
   int i, j, sum = 0;

   for (i = 0; i < M; i++)
      for (j = 0; j < N; j++)
         sum = sum + a[i][j];
   return sum;
}
```

good spatial locality

| Address | 0 | 4 | 8 | 12 | 16 | 20 |
|---|---|---|---|---|---|---|
| Contents | $a_{00}$ | $a_{01}$ | $a_{02}$ | $a_{10}$ | $a_{11}$ | $a_{12}$ |
| Access Order | 1 | 2 | 3 | 4 | 5 | 6 |

# Example

- What is the impact of interchanging `i` and `j` loops?

```
int sumarrayrows(int a[M][N]) {
   int i, j, sum = 0;

   for (j = 0; j < N; j++)
      for (i = 0; i < M; i++)
         sum = sum + a[i][j];
   return sum;
}
```

poor spatial locality

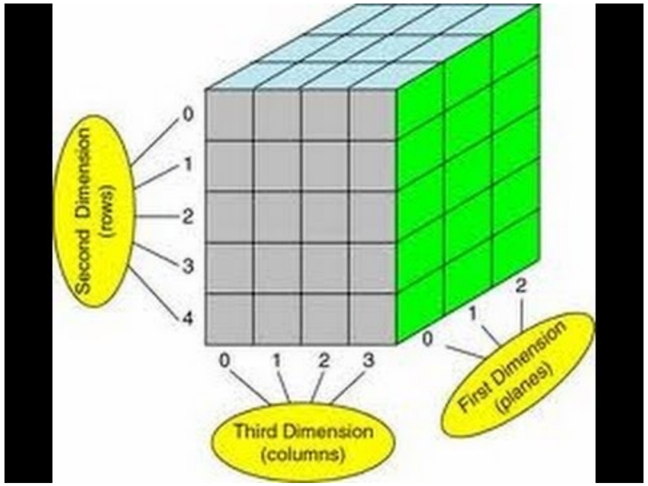
| Address | 0 | 4 | 8 | 12 | 16 | 20 |
|---|---|---|---|---|---|---|
| Contents | $a_{00}$ | $a_{01}$ | $a_{02}$ | $a_{10}$ | $a_{11}$ | $a_{12}$ |
| Access Order | 1 | 3 | 5 | 2 | 4 | 6 |

# Practice Problem

Permute the loops in the following function so that it scans the 3-dimensional array a with a stride-1 reference pattern.

```
int sumarray3d(int a[N][N][N]) {
   int i, j, k, sum = 0;

   for (i = 0; i < N; i++)
      for (j = 0; j < N; j++)
         for (k = 0; k < N; k++)
            sum = sum + a[k][i][j];
   return sum;
}
```



# Locality of Instructions

- Program instructions are stored in memory and must be fetched by CPU
  - Locality is also relevant to instruction accesses
- Instructions in the loop body have high locality
  - Good spatial locality because instructions next to each other are executed in sequential order
  - Good temporal locality because the loop body is executed multiple times

# Assignment 1 Grade Summary


*Something was submitted by the deadline*

# Issues I Observed ☹

- Work in a group, but do not submit the group survey
- Dump the submission in the lecturer's inbox
  - Which gets ~100 emails on a slow-paced day
- Do not fork the assignment
- Submit only the top-level circuit and nothing else
- Do not submit report
- Do not respect the deadline, don't ask for extension
- Do not submit Statement of Originality

# Good News

- Many reports were a pleasure to read
- Highest: 99.5
- Constantly surprised with the ambitious extensions
  - Multi-Cycle
  - Pipelining
  - New instructions for condensing the code
- On average, neatness and clarity was clearly seen in submissions that were pushed on time