Part II
Arrays, Pointers, and Dynamic Memory

Abstractions in Computer Systems

- Computer System
- Application Software
- Operating System
- Instruction Set Architecture
- Microarchitecture
- Logic and Building Blocks
- Digital Circuits
- Analog Circuits
- Devices and Physics

Networked Systems and Systems of Systems
Software
Hardware/Software Interface
Digital Hardware Design
Analog Design and Physics

Agenda

Part I
More on Control Structures and Functions

Part II
Arrays, Pointers, and Dynamic Memory

Part III
Floating-Point Numbers
Part I
More on Control Structures and Function Calls

Loops
do/while

```
int x = 0;
while(x < 10){
    x++;
    printf("%d\n",x);
}
```

```
int x = 0;
while(x < 10){
    do{
        x++;
        printf("%d\n",x);
    }while(x < 10);
}
```

What is the difference in result?

**Answer:** None. Both prints out numbers 1 to 10.

What is the difference in result if int x = 10;

while loop: no prints
do/while loop: prints out number 11

Conditional Statements
switch-statement

```
int op = 3;
int z = 0;
switch(op){
    case 1:
        z = 4;
        printf("case 1\n");
    break;
    case 2:
        printf("case 2\n");
    break;
    default:
        printf("default\n");
}
```

**A switch** is semantically equivalent to several if-then-else statements, but a switch is cleaner and can be implemented more efficiently.

After each case, we need to **break** out of the switch.

If no case matches, the **default** case is executed. In this case, the output will be "default" because the value 3 is not part of any case.

Floating-point numbers approximate the result.
Example: Word Count - Incorrect Implementation

```c
#include <stdio.h>
int main(){
    char c;
    int lines, words = 0;
    while((c = getchar()) != EOF){
        if(c == ' ')
            lines++;
        else
            words += in_space;
        in_space = 1;
    }
    printf("%8d%8d%8d\n", lines, words, chars);
    return 0;
}
```

This example code should give the same result as the UNIX command `wc` (word count).

For a text file `wctest.txt` (above), we should get:
```
$ cat wctest.txt | wc
lines  words  chars
  2       10       45
```

Exercise:
Find 4 errors in the code!

---

Example: Word Count - Correct Implementation

```c
#include <stdio.h>
int main(){
    char c;
    int lines = 0, words = 0;
    while((c = getchar()) != EOF){
        if(c == ' ')
            lines++;
        else
            if(c == 'n')
                words += in_space;
            in_space = 1;
    }
    printf("%8d%8d%8d\n", lines, words, chars);
    return 0;
}
```

---

Functions (1/3) Parameters and Arguments

Return type Two parameters and types Two arguments

```c
int sum(int x, int y){
    return x+y;
}
```

Expression return 105 (obviously)

Exercise:
Write a function called `expo` that computes the exponential value $x^n$. For instance `expo(4,3) = 64`

```c
int expo(int x, int n){
    int r = 1;
    for(int i=0; i<n; i++)
        r *= x;
    return r;
}
```

---

Functions (2/3) Local and Global Variables

Global variable. Should in general be avoided. Violates the principle of modularity.

```c
void expo_glob(int x){
    int i;
    r = 1;
    for(int i=0; i<n; i++)
        r *= x;
    ng += 2;
}
```

Local variable. Can only be used inside a function.

```c
int ng = 2;
int r;
```

```c
void expo_glob(int x){
    int i;
    r = 1;
    for(int i=0; i<ng; i++)
        r *= x;
    ng += 2;
}
```

void type means that it is a procedure, it does not return a value.

The function has side effects.

First called. `ng = 4, r = 4`
Second called. `ng = 6, r = 16`
Third called. `ng = 8, r = 64`
Note that this is not an example of good code design...
Exercise:
Create the factorial \( n! \), where \( n \) is an integer parameter. Create one imperative and one functional implementation. The latter one should use recursion.

```c
unsigned int fact(unsigned int n)
{
    if(n <= 1)
        return 1;
    return n * fact(n-1);
}
```

Exercise: Create the factorial function \( n! \), where \( n \) is an integer parameter. Create one imperative and one functional implementation. The latter one should use recursion.

```c
unsigned int fact(unsigned int n)
{
    int r = 1;
    while(n > 1){
        r = r * n;
        n--;
    }
    return r;
}
```

Part II
Arrays, Pointers, and Dynamic Memory

David Broman
dbro@kth.se

Arrays (1/3)
Declaration

An uninitialized array is declared by stating the element type, the array name, and its length (number of elements).

```c
int b[3];
b[0] = 10;
b[1] = 30;
b[2] = 20;
b[3] = 30;
```

An array can be initialized directly.

```c
int a[5] = {12,23,15,100,9};
```

The array can then be given values using assignment statements.

```c
int a[] = {12,23,15,100,9};
```

The last assignment is an out-of-bound error. The compiler can sometimes issue a warning, but not always.

```c
int a[] = {12,23,15,100,9};
```

Arrays (2/3)
Accessing Elements

The array can be accessed using index notation.

```c
printf("Total size in bytes: %d\n", sizeof(a));
```

Casting to integer (int) from unsigned long.

```c
sizeof(a)/sizeof(int)
```

The number of elements can be computed like this. Result: 5

```c
size_t sizeof();
```

The length can be inferred.

```c
int a[] = {12,23,15,100,9};
```

The number of elements can be computed like this. Result: 5

```c
sizeof(a)/sizeof(int)
```

Exercise: Which function(s) return correct answers. Stand for mean2 and mean4, sleep for mean3. Answer: mean2 and mean4

```
Exercise: Which function(s) return correct answers. Stand for mean2 and mean4, sleep for mean3. Answer: mean2 and mean4
```

Part II
Arrays, Pointers, and Dynamic Memory

David Broman
dbro@kth.se

Arrays (2/3)
Accessing Elements

Out of bound for the \( k5 \) case. An array of size \( N \) is indexing from 0 to \( N-1 \).

```c
int a[] = {1,3,2,4,3};
int k0 = a[0];
int k5 = a[5];
```

Needs to be a double before dividing.

```c
double mean1(int d[], int len){
    int i, sum = 0;
    for(i=0; i<len; i++)
        sum += d[i];
    return sum / len;
}
```

Casting has higher precedence than div.

```c
double mean2(int d[], int len){
    int i;
    double sum = 0;
    for(i=0; i<len; i++)
        sum += d[i];
    return sum / len;
}
```

```c
double mean3(int d[], int len){
    int i, sum = 0;
    for(i=0; i<len; i++)
        sum += d[i];
    return (double) (sum / len);
}
```

```c
double mean4(int d[], int len){
    int i, sum = 0;
    for(i=0; i<len; i++)
        sum += d[i];
    return (double) sum / len;
}
```

Exercise: Which function(s) return correct answers. Stand for mean2 and mean4, sleep for mean3. Answer: mean2 and mean4

```
Exercise: Which function(s) return correct answers. Stand for mean2 and mean4, sleep for mean3. Answer: mean2 and mean4
```
Arrays (3/3)
Multi-Dimensional Arrays

```c
void print_matrix(const int mtx[2][4])
{
    int i,j;
    for(i=0; i<2; i++)
        for(j=0; j<4; j++)
            printf("%2d ", mtx[i][j]);
    printf("\n");
}
```

```
#include <stdlib.h>
void random_matrix(int mtx[2][4])
{
    int i,j;
    for(i=0; i<2; i++)
        for(j=0; j<4; j++)
            mtx[i][j] = rand() % 100;
}
```

int m[2][4] = {{42, 77, 92, 10},
               {31, 21, 33, 61}};

int m2[2][4];
random_matrix(m2);
print_matrix(m2);

Pointers (1/3)
Swap values of two variables.

```c
int x = 3, y = 7;
int t;
t = x;
x = y;
y = t;
```

```c
void swap1(int x, int y)
{
    int t;
t = x;
x = y;
y = t;
}
```

```c
int a = 3, b = 7;
swap1(&a, &b);
```

Pointers (2/3)
Solution: Use pointers

A pointer is defined with the * symbol before the variable name in a variable definition.

```c
int a = 2;
int *p = &a;
*p = 3 + *p;
printf("p=0x%x a=0x%x\n",
       (unsigned int)p, a);
```

What is the output when executing this code?

Answer: p=0x104008 a=0x5

```c
#include <stdlib.h>
void swap2(int *x, int *y)
{
    int t;
t = *x;
    *x = *y;
    *y = t;
}
```

Back to the swap example...

```c
void swap2(int *x, int *y)
{
    int t;
t = *x;
    *x = *y;
    *y = t;
}
```

```c
int a = 3, b = 7;
swap2(&a, &b);
```

We define pointer parameters.

We dereference the pointer *x and get the value of a.

Finally, we dereference *y, and update b with the value of t.

A safer and simpler programming style with reference types is available in C++, but not in C.
Dynamic Memory Allocation

All examples so far have been using statically defined variables or allocating on the stack (local variables).

```c
int n = 100;
int *buf = malloc(sizeof(int)*n);
buf[4] = 10;
printf("%d\n", buf[4]);
free(buf);
```

We can access the array using array indexing.

When the buffer is not needed anymore, it must be deallocated using free()!

---

C/ASM Pointer Example (2/2)

```asm
.data
.align 2
numbers: .space 40
.align 2
.text
la $t1,numbers
addi $s0, $0, 10
loop:
sw $s0, 0($t1)
addi $s1, $t1, 4
addi $s0, $s0, -1
`bne $s0, $0, loop
la $t1,numbers
lw $s0,0($t1)
lw $s1,4($t1)
add $s2, $s0, $s1
stop: j stop
```

Home Exercise (fun todo after this lecture):
Write a C program (using pointers) that performs the same task.

```c
#include <stdio.h>
int numbers[10];
int main(){
int *p = numbers;
int i = 10;
do{
*p = i;
p++;
i--;
}while(i != 0);
p = numbers;
int result = *p + (*(p+1));
printf("Result: %d\n", result);
}
```

---

Layout of Memory – the Memory Map for MIPS

- Dynamic data contains the stack and the heap
  - Stack pointer $sp = 0x7FFFFFFC
- The global data contains global variables. ".data" in assembly code
  - Global pointer $gp = 0x10008000
- The text segment stores the code. ".text" in assembly code
- PC (Program Counter): Program starts at PC = 0x00400000

Reserved

Text

Global Data

Dynamic Data

Reserved

Stack

- Address
- Segments

Home Exercise (fun todo after this lecture):
Write a C program (using pointers) that performs the same task.

```c
#include <stdio.h>
int numbers[10];
int main(){
int *p = numbers;
int i = 10;
do{
*p = i;
p++;
i--;
}while(i != 0);
p = numbers;
int result = *p + (*(p+1));
printf("Result: %d\n", result);
}
```

---

Floating-Point Numbers

Note that when an integer pointer is incremented with 1, the pointer address value is increased by four (assuming that the word size is 4 bytes)
Floating-Point Numbers (1/2)
Basics

Floating point numbers can represent an approximation of real numbers in a computer. Used heavily in high performance scientific computing.

\[ 3.7 \times 10^{-3} = 0.0037 \]

C-code. Float 32-bit floating-point numbers.

\[
\begin{align*}
\text{float } x &= 3.7e-3; \\
\text{float } y &= 0.0037; \\
\text{printf}(&quot;%f,%f,%d&quot;,x,y,x==y);
\end{align*}
\]

Output: 0.003700, 0.003700, 1

Standard IEEE 754 defines
- 32-bit floating point number (float in C)
- 64-bit floating point number (double in C)

Special numbers
- + infinity
- - infinity
- NaN (Not a number)

Reading Guidelines

Next Module 2 (I/O Systems)
H&H Chapters 8.5-8.7
For the labs, focus on 8.6.2-8.6.5 (GPIO, Timers, and Interrupts).
The rest is useful for the project.

Reading Guidelines
See the course webpage for more information.
Summary

Some key take away points:

- **Arrays** and **pointers** are expressive, low-level data structures in C.

- **Floating-point numbers** are very useful, but should be used carefully when comparing numbers.

Thanks for listening!