Course Structure

Module 1: C and Assembly Programming
LE4 → LE2 → LE3 → EX1 → LAB1 → S1 → LAB2

Module 2: I/O Systems
LE5 → LE6 → EX2 → LAB3

Module 3: Logic Design
LE7 → LE8 → EX3 → LD-LAB

Module 4: Processor Design
LE9 → LE10 → EX4 → S2 → LAB4

Module 5: Memory Hierarchy
LE11 → EX5 → S3

Module 6: Parallel Processors and Programs
LE12 → LE13 → EX6 → S4

Proj. Expo → LE14

Part I: More on Control Structures and Functions
Part II: Arrays, Pointers, and Dynamic Memory
Part III: Floating-Point Numbers
Abstractions in Computer Systems

- 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

foo(int boo)

Part II

Arrays, Pointers, and Dynamic Memory

Part III

Floating-Point Numbers

David Broman
dbro@kth.se

Part I

More on Control Structures and Function Calls

foo(int boo)

Part II

Arrays, Pointers, and Dynamic Memory

Part III

Floating-Point Numbers

David Broman
dbro@kth.se

Primitve Data Types
Integers and Floating Points

Type | Size(bits) | Min | Max
--- | --- | --- | ---
char | 8 | \(-2^7 = -128\) | \(2^7 - 1 = 127\)
unsigned char | 8 | 0 | \(2^8 - 1 = 255\)
short | 16 | \(-2^{15} = -32768\) | \(2^{15} - 1 = 32767\)
unsigned short | 16 | 0 | \(2^{16} - 1 = 65535\)
long | 32 or 64 | machine dependent (signed)

int | machine dependent (signed) |
unsigned int | machine dependent (unsigned) |

float | 32 |
double | 64 |

Floating-point numbers approximate the result.
Loops
do/while

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

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

What is the difference in result?

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

Do/while makes the check at then end.

What is the difference in result if

```c
int x = 10;
```

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

---

Conditional Statements

**switch**-statement

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

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 in not part of any case.
Example: Word Count - Incorrect Implementation

```
#include <stdio.h>
int main(){
    char c;
    int lines, words = 0;
    int chars, in_space = 1;
    while((c = getchar()) != EOF){
        chars++;
        if(c == '\n')
            lines++;
        if(c == ' ' || c == '\t' || c == '\n')
            in_space = 1;
        else
            words += in_space;
            in_space = 0;
    }
    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).

```
Example: Word Count - Correct Implementation

```
#include <stdio.h>
int main(){
    char c;
    int lines = 0, words = 0;
    int chars = 0, in_space = 1;
    while((c = getchar()) != EOF){
        chars++;
        if(c == '\n')
            lines++;
        if(c == ' ' || c == '\t' || c == '\n')
            in_space = 1;
        else{
            words += in_space;
            in_space = 0;
        }
    }
    printf("%8d%8d%8d\n", lines, words, chars);
    return 0;
}
```

1. Must define values. Dangerous to define several variables on one line.
2. Should compare for equality `==`, not perform an assignment `=` (most compilers issue a warning)
3. Whitespace includes tab: ‘\t’
4. Need to define a block for the else-if construct.

1. Must define values. Dangerous to define several variables on one line.
2. Should compare for equality `==`, not perform an assignment `=` (most compilers issue a warning)
3. Whitespace includes tab: ‘\t’
4. Need to define a block for the else-if construct.
Functions (1/3)
Parameters and Arguments

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

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

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

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

```c
int sum(35, 40) + sum(10, 20)
```

Expression return 105 (obviously)

Functions (2/3)
Local and Global Variables

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

```c
type ng = 2;
type r;

void exp_glob(type x){
    type i;
    r = 1;
    for(i=0; i<ng; i++)
        r *= x;
    ng += 2;
}
```

Local variable.
Can only be used inside a function.

```c
exp_glob(2);
exp_glob(2);
expo_glob(2);
```

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 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;
}
```  

Functional, Recursive

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

Imperative

Part II

Arrays, Pointers, and Dynamic Memory
Arrays (1/3)
Declaration

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

The array can then be given values using assignment statements.

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

An array can be initialized directly.

The length can be inferred.

sizeof() can be used to get its size in bytes. Result: 20

Casting to integer (int) from unsigned long.

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

Arrays (2/3)
Accessing Elements

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

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

(double) (sum / len);

Casting has higher precedence than div.

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

Needs to be a double before dividing.
### 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");
    }
}
```

```c
#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;
    }
}
```

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

// Can declare two dimensional arrays.

Row Columns

Note that print function can have a const parameter (read only), but not the function with side effect.

The random function rand() is part of the standard library.

```c
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;
```

Store one of the values in a temporary variable.

**How can we write a function that performs the swap?**

**Problem 1:** A function can only return one value.

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

Arguments are passed as values. Variables `x` and `y` are only local variables.

**Problem 2:** We cannot modify the content of variables outside a function, if they passed as value arguments.

```c
int a = 3, b = 7;
swap1(a,b);
```
Part II
Arrays, Pointers, and Dynamic Memory

David Broman
dbro@kth.se

Part I
More on Control Structures and Functions

Part III
Floating-Point Numbers

Pointers (2/3)

Solution: Use pointers

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

An & symbol is used in an expression for getting the memory address of a variable.

*A pointer variable dereferences a pointer, i.e., returns or assigns the value that the pointer points to.

What is the output when executing this code?

Answer:
p=0x104008 a=0x5

Memory content after executing the 3 first lines of code.

0x0010 4008 0x0000 0002 a
0x0010 4004 0x0000 0002
0x0010 4000 0x0010 4008 p

Pointers (3/3)

Back to the swap example...

We define pointer parameters.

The pointer *x is dereferences, that is, we get the value of a.

We dereference the pointer *y and get the value of b, followed by dereferencing *x and updating the value of a.

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

The memory address of a and b are passed as values, not the content of a and b.

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).

 malloc dynamically allocates N number of bytes, where N is the argument. It returns a pointer to the new data.

```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().

Layout of Memory – the Memory Map for MIPS

<table>
<thead>
<tr>
<th>Address</th>
<th>Segments</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>Reserved</td>
</tr>
<tr>
<td>0x10000000</td>
<td>Global Data</td>
</tr>
<tr>
<td>0x10008000</td>
<td>Global pointer $gp = 0x10008000</td>
</tr>
<tr>
<td>0x7FFFFFFC</td>
<td>Stack pointer $sp = 0x7FFFFFFFC</td>
</tr>
<tr>
<td>0x00400000</td>
<td>Text</td>
</tr>
<tr>
<td>0x80000000</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

The reserved memory may be used for memory mapped I/O.

Dynamic data contains the stack and the heap.

Stack pointer $sp = 0x7FFFFFFFC

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
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);
}
```

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 x 10^{-3} = 0.0037

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

float x = 3.7e-3;
float y = 0.0037;
printf("%f,%f,%d\n",x,y,x==y);

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)

Floating-Point Numbers (2/2) 
Rounding

Surprising fact about floating point numbers:
0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 

Note: The printed output is the same, but the internal value is not exactly the same due to rounding errors. The equality test becomes false!

Exercise: What is the output of the program

Solution:
1.000000,1.000000,0
Yes, there will be coffee in just second…
(please do not fumble with the bags)

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!

<table>
<thead>
<tr>
<th>Part I</th>
<th>Part II</th>
<th>Part III</th>
</tr>
</thead>
<tbody>
<tr>
<td>More on Control</td>
<td>Arrays, Pointers, and Dynamic Memory</td>
<td>Floating-Point Numbers</td>
</tr>
<tr>
<td>Structures and Functions</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

David Broman
dbro@kth.se