



# Computer Hardware Engineering (IS1200)

## Computer Organization and Components (IS1500)

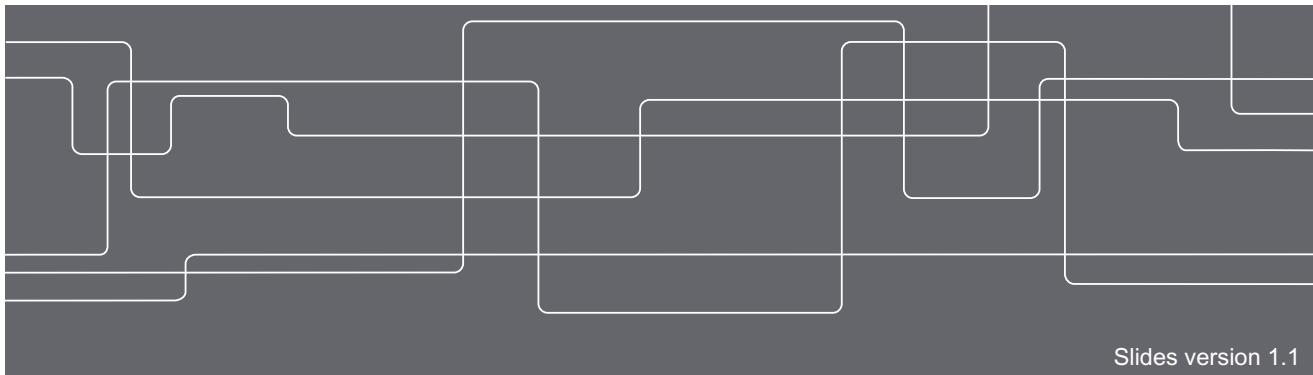
Spring 2019

### Lecture 4: The C Programming Language Continued

Elias Castegren

Postdoc, KTH Royal Institute of Technology

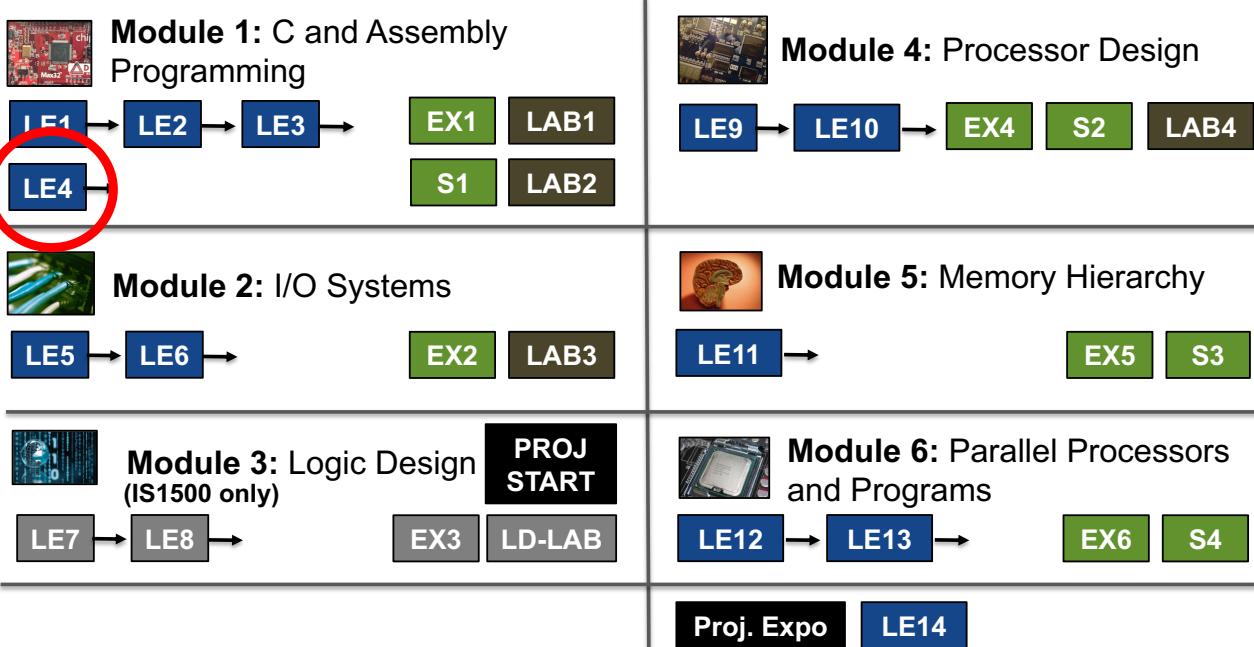
Slides by David Broman



2

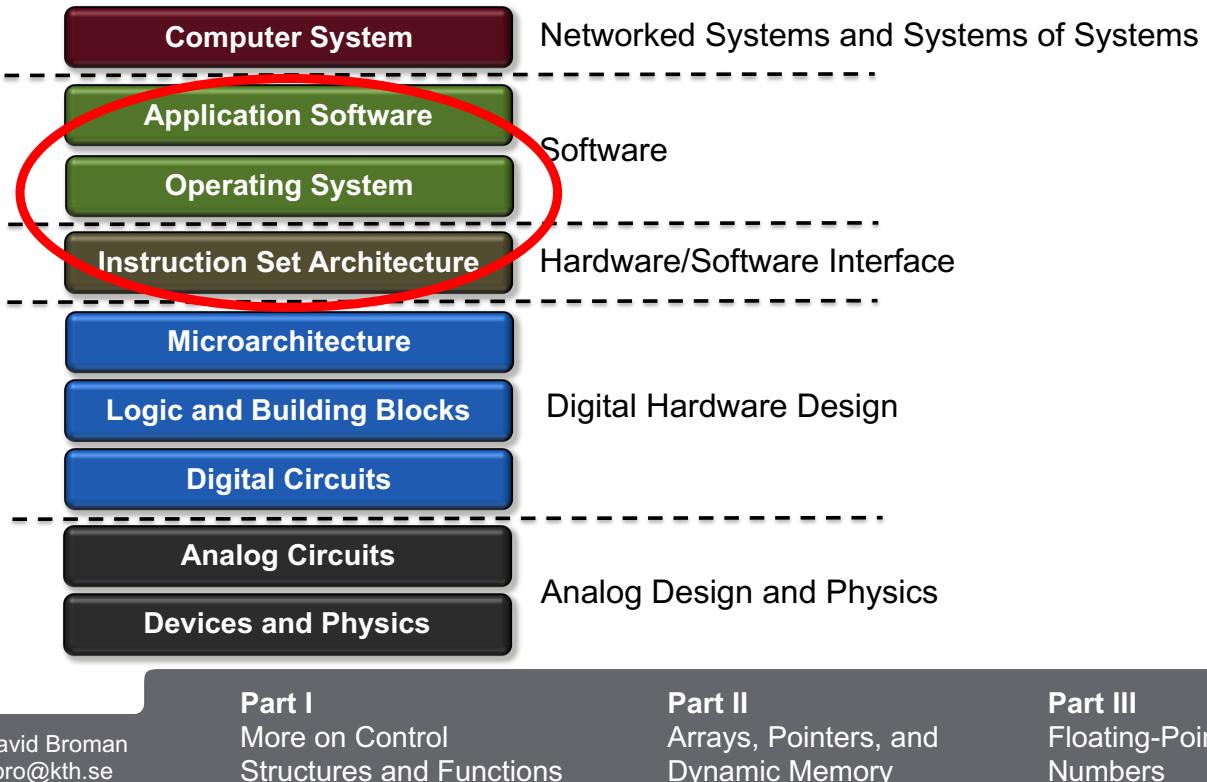


## Course Structure

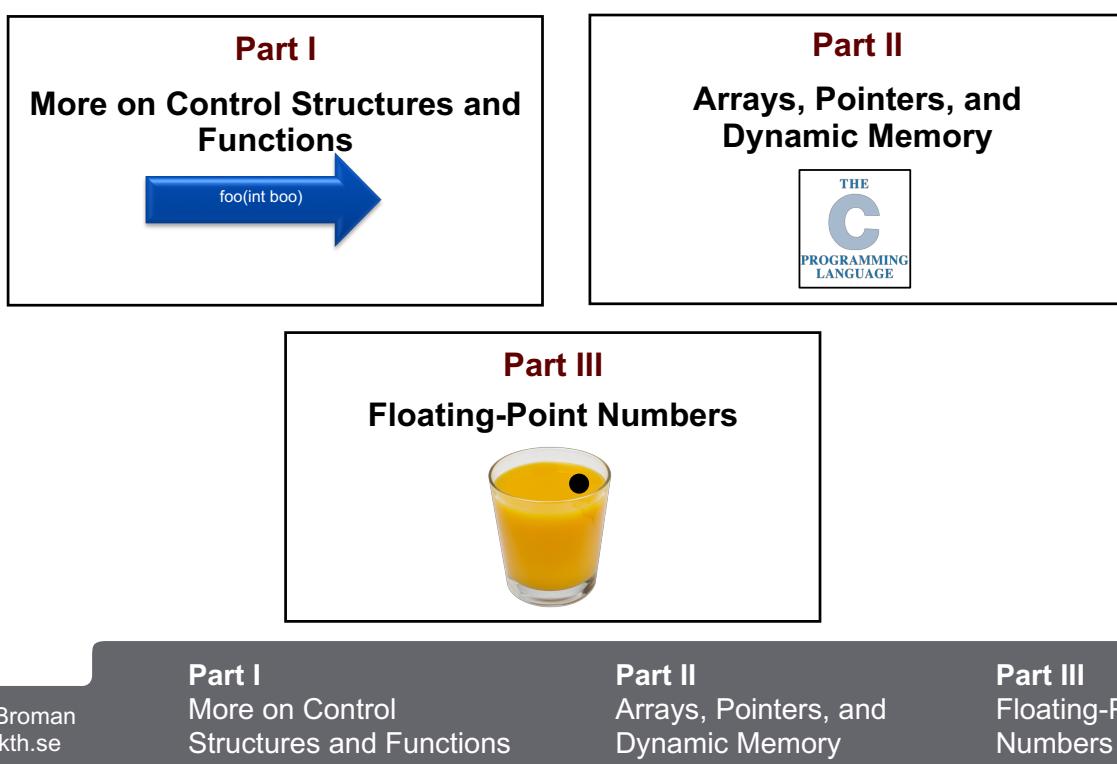




# Abstractions in Computer Systems



# Agenda





## Part I

# More on Control Structures and Function Calls

foo(int boo)

David Broman  
dbro@kth.se



**Part I**  
More on Control  
Structures and Functions

**Part II**  
Arrays, Pointers, and  
Dynamic Memory

**Part III**  
Floating-Point  
Numbers



## Primitive Data Types Integers and Floating Points

`int foo;`

Defines an uninitialized integer. What does it mean?  
We can use expression `sizeof(foo)` to find out the size.

Type	Size(bits)	Min	Max
<code>char</code>	8	$-2^7 = -128$	$2^7-1 = 127$
<code>unsigned char</code>	8	0	$2^8-1 = 255$
<code>short</code>	16	$-2^{15} = -32768$	$2^{15}-1 = 32767$
<code>unsigned short</code>	16	0	$2^{16}-1 = 65535$
<code>long</code>	32 or 64		
<code>int</code>	machine dependent (signed)		
<code>unsigned int</code>	machine dependent (unsigned)		
<code>float</code>	32		
<code>double</code>	64		

Floating-point numbers approximate the result.

David Broman  
dbro@kth.se



**Part I**  
More on Control  
Structures and Functions

**Part II**  
Arrays, Pointers, and  
Dynamic Memory

**Part III**  
Floating-Point  
Numbers

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

```
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 if  
**int x = 10;**

while loop: no prints

do/while loop: prints out number 11

David Broman  
dbro@kth.se



**Part I**  
More on Control  
Structures and Functions

**Part II**  
Arrays, Pointers, and  
Dynamic Memory

**Part III**  
Floating-Point  
Numbers

```
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 is not part of any case.

David Broman  
dbro@kth.se



**Part I**  
More on Control  
Structures and Functions

**Part II**  
Arrays, Pointers, and  
Dynamic Memory

**Part III**  
Floating-Point  
Numbers

# 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 == '\n')
            in_space = 1;
        else
            words += in_space;
            in_space = 0;
    }

    printf("%d %d %d\n", lines, words, chars);
    return 0;
}
```

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

This is a text. We have right now ten words.

For a text file **wctest.txt** (above), we should get:

\$ cat wctest.txt | wc  
 2 10 45

↑      ↑      ↑  
 lines    words    chars

**Exercise:**  
 Find 4 errors in the code!

David Broman  
 dbro@kth.se



**Part I**  
 More on Control  
 Structures and Functions

**Part II**

Arrays, Pointers, and  
 Dynamic Memory

**Part III**

Floating-Point  
 Numbers

# 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("%d %d %d\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.

The compiler can help!  
 gcc warns about 1 and 2 when run with **-Wall**

David Broman  
 dbro@kth.se



**Part I**  
 More on Control  
 Structures and Functions

**Part II**

Arrays, Pointers, and  
 Dynamic Memory

**Part III**

Floating-Point  
 Numbers

# Functions (1/3)

## Parameters and Arguments

Return type

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

Two parameters  
and types

Return  
value

Two arguments

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

Expression return 105 (obviously)

### Exercise:

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

Multiplication  
and assignment  
operator.

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

David Broman  
dbro@kth.se



**Part I**  
More on Control  
Structures and Functions

**Part II**  
Arrays, Pointers, and  
Dynamic Memory

**Part III**  
Floating-Point  
Numbers

# Functions (2/3)

## Local and Global Variables

### Global variable.

Should in general be avoided. Violates the principle of **modularity**.

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

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

### Local variable.

Can only be used inside a function.

The **function** has side effects.

```
expo_glob(2);  
expo_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...

David Broman  
dbro@kth.se



**Part I**  
More on Control  
Structures and Functions

**Part II**  
Arrays, Pointers, and  
Dynamic Memory

**Part III**  
Floating-Point  
Numbers

**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.

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

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

**Functional, Recursive****Imperative**

David Broman  
dbro@kth.se



**Part I**  
More on Control  
Structures and Functions

**Part II**  
Arrays, Pointers, and  
Dynamic Memory

**Part III**  
Floating-Point  
Numbers

## Part II

# Arrays, Pointers, and Dynamic Memory



**Part I**  
More on Control  
Structures and Functions

**Part II**  
Arrays, Pointers, and  
Dynamic Memory

**Part III**  
Floating-Point  
Numbers



# Arrays (1/3)

## Declaration

15

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

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.

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

An array can be initialized directly.

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

The length can be inferred.

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

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

**Casting** to integer (int) from unsigned long.

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

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

David Broman  
dbro@kth.se

**Part I**  
More on Control  
Structures and Functions

**Part II**  
Arrays, Pointers, and  
Dynamic Memory

**Part III**  
Floating-Point  
Numbers



# Arrays (2/3)

## Accessing Elements

16  
E

```
int a[] = {1,3,2,4,3};
```

```
int k0 = a[0];
int k5 = a[5];
```

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];           (returns 2.0)
    return sum / len;
}
```

```
double mean3(int d[], int len){
    int i, sum = 0;
    for(i=0; i<len; i++)
        sum += d[i];           (returns 2.0)
    return (double) (sum / len);
}
```

```
double mean2(int d[], int len){
    int i;
    double sum = 0;
    for(i=0; i<len; i++) (returns 2.6)
        sum += d[i];
    return sum / len;
}
```

```
double mean4(int d[], int len){
    int i,sum = 0;
    for(i=0; i<len; i++) (returns 2.6)
        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

Needs to be a double before dividing.

David Broman  
dbro@kth.se

**Part I**  
More on Control  
Structures and Functions

**Part II**  
Arrays, Pointers, and  
Dynamic Memory

**Part III**  
Floating-Point  
Numbers

```
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");
    }
}
```

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

Rows      Columns      Can declare two dimensional arrays.

```
#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;
    }
}
```

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.

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

David Broman  
dbro@kth.se

**Part I**  
More on Control  
Structures and Functions

**Part II**  
Arrays, Pointers, and  
Dynamic Memory

**Part III**  
Floating-Point  
Numbers



# Pointers (1/3)

Swap values of two variables.

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

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

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

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.

David Broman  
dbro@kth.se

**Part I**  
More on Control  
Structures and Functions

**Part II**  
Arrays, Pointers, and  
Dynamic Memory

**Part III**  
Floating-Point  
Numbers

## Solution: Use pointers

A pointer is defined with the \* symbol before the variable name in a variable definition. NOTE: we can also write

```
int* p;
```

```
int a = 2;
int *p;
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

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

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

0x0010 4008  
0x0010 4004  
0x0010 4000

0x0000 0002	a
0x0010 4008	p

Memory content after executing the 3 first lines of code.

David Broman  
dbro@kth.se

**Part I**  
More on Control  
Structures and Functions

**Part II**  
Arrays, Pointers, and  
Dynamic Memory

**Part III**  
Floating-Point  
Numbers



Back to the swap example...

We define pointer parameters.

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

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.

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

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.

David Broman  
dbro@kth.se

**Part I**  
More on Control  
Structures and Functions

**Part II**  
Arrays, Pointers, and  
Dynamic Memory

**Part III**  
Floating-Point  
Numbers

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.

```
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()**

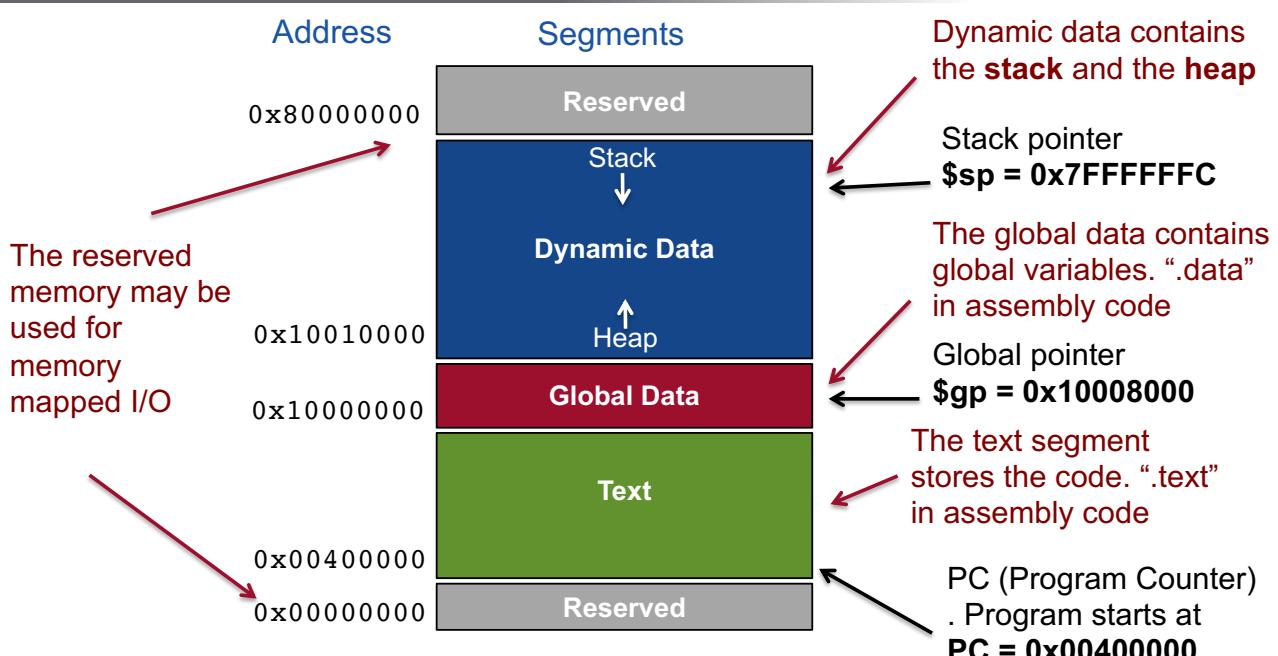
David Broman  
dbro@kth.se

**Part I**  
More on Control  
Structures and Functions

**Part II**  
Arrays, Pointers, and  
Dynamic Memory

**Part III**  
Floating-Point  
Numbers

## Layout of Memory – the Memory Map for MIPS



David Broman  
dbro@kth.se

**Part I**  
More on Control  
Structures and Functions

**Part II**  
Arrays, Pointers, and  
Dynamic Memory

**Part III**  
Floating-Point  
Numbers

```

.data
.align 2
numbers: .space 40

.align 2
.text
    la      $t1, numbers
    addi   $s0, $0, 10
loop:
    sw      $s0, 0($t1)
    addi   $t1, $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.

```

#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)

David Broman  
dbro@kth.se

**Part I**  
More on Control  
Structures and Functions

**Part II**  
Arrays, Pointers, and  
Dynamic Memory

**Part III**  
Floating-Point  
Numbers



## Part III

### Floating-Point Numbers



David Broman  
dbro@kth.se

**Part I**  
More on Control  
Structures and Functions

**Part II**  
Arrays, Pointers, and  
Dynamic Memory

**Part III**  
Floating-Point  
Numbers

# 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.

$$3.7 \times 10^{-3} = 0.0037$$

mantissa      base      exponent

C-code. Type float represents a 32-bit floating-point number.

```
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)

David Broman  
dbro@kth.se

**Part I**  
More on Control  
Structures and Functions

**Part II**  
Arrays, Pointers, and  
Dynamic Memory

**Part III**  
Floating-Point  
Numbers

# Floating-Point Numbers (2/2)

## Rounding



Surprising fact about floating point numbers:  
 $0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 \neq 1.0$

```
double z = 0.1;
double r = 0.0;
double k = 1.0;
for(int i = 0; i < 10; i++)
    r += z;
printf("%f,%f,%d\n",r,k,r==k);
```

**Exercise:** What is the output of the program

**Solution:**  
1.000000,1.000000,0

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!**

David Broman  
dbro@kth.se

**Part I**  
More on Control  
Structures and Functions

**Part II**  
Arrays, Pointers, and  
Dynamic Memory

**Part III**  
Floating-Point  
Numbers

**Yes, there will be coffee in just second...**  
**(please do not fumble with the bags)**



David Broman  
dbro@kth.se

**Part I**  
More on Control  
Structures and Functions

**Part II**  
Arrays, Pointers, and  
Dynamic Memory

**Part III**  
Floating-Point  
Numbers

## 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.

David Broman  
dbro@kth.se

**Part I**  
More on Control  
Structures and Functions

**Part II**  
Arrays, Pointers, and  
Dynamic Memory

**Part III**  
Floating-Point  
Numbers



## 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!

David Broman  
dbro@kth.se

**Part I**  
More on Control  
Structures and Functions

**Part II**  
Arrays, Pointers, and  
Dynamic Memory

**Part III**  
Floating-Point  
Numbers