Computer programming in INF236

Today:

- Introduction to C
  - Some important differences from Java

The C programming language

Developed 1969 – 1973 by Dennis Ritchie
“High level assembly code”

Not object oriented! (Look at C++)

Similar types of constructions as Java
- Declaration of variables (int, float, double, boolean etc)
- Code blocks delimited by {…}
- While and for loops just like Java
- Use of (static) procedures and functions

Some differences:
- No objects or classes
- Compiles to assembly code, no portable byte code
- Gives more low level control, often more efficient code
- Use of pointers
- Allocation of memory
- Input – Output (simpler than in Java)

Programming in C

The simplest and best known among C programs:

```
#include <stdio.h>

int main(int argc, char *argv[]) {

  printf("Hello World!\n");

}
```

Compile with:

```
gcc hello.c -o hello
```

Run with:

```
./hello
```

Will result in:

```
Hello World!
```

Pointers

Memory is viewed as one long consecutive array of bytes.

Consecutive bytes can be interpreted as any data type.

```
int i=5;
int *a;       // The * says that a is a pointer to an int
a = &i;       // & returns the address of i, now a points to i

printf(”The variable i has value %d.\n”, i); // Should print 5
printf(”The pointer a points at an int with value %d.\n”, *a);
         // Should also print 5
```

Observations:

- `&i` means the memory address of variable `i`
- `*a` means the value of the variable (memory cell) to which `a` points

Pointers are useful:

- to build arrays with dynamic memory allocation
- when passing arguments to functions

Dynamic memory allocation

One-dimensional arrays:

```
// Static memory allocation:
int aStatic[100];
aStatic[5] = 73; // Legal

// Dynamic memory allocation:
int n=100;  // Array size
int *a;     // To point at the FIRST array element
a = (int*) malloc(n*sizeof(int)); // malloc allocates
                                  // contiguous memory!
// Now a can be considered as an array:
int i;  // Counters cannot be declared within for statements!
for(i=0;i<n;i++)
  a[i] = rand();  // Fill a with random integers

for(i=0;i<n;i++)
  printf("a[%d]=%d\n",i,a[i]);

free(a);  // Free up the memory that a points to!
```

Dynamic memory allocation

Observations:

- `malloc(m)`
  - allocates `m` bytes of contiguous memory
  - returns the memory address (`void`*) of the first byte
  - must cast returned memory address to wanted type
- `sizeof(datatype)` = number of bytes occupied by `datatype`-variables
- when the memory is no longer needed it must be released (`free`) to avoid memory leaks
- the compiler (might) accept

  ```
  int *a;
  a[5] = 73;
  ```

  but a runtime error (segmentation fault) could occur
- You can write either `int *a` or `int* a`
- You can write either `a[i]` (recommended) or `*(a+i)`

- Note that there is no range checking at runtime

Dynamic memory allocation

Two-dimensional arrays:

```
// Static memory allocation:
int aStatic[100][200];
aStatic[5][8] = 73; // Legal

// Dynamic memory allocation:
int m=100, n=200;  // Array size (rows, columns)
int **a;  // Arrays of int-arrays => pointer to int-pointers
a = (int**) malloc(m*sizeof(int*)); // space for row pointers

// Allocate memory for each row
int i,j;  // Row and column counters
for(i=0;i<m;i++)
  a[i] = (int*) malloc(n*sizeof(int)); // Rows have length n

for(i=0;i<m;i++)
  for(j=0;j<n;j++)
    a[i][j] = rand();  // Fill a with random integers
```

Dynamic memory allocation

Two-dimensional arrays:

```
for(i=0;i<m;i++)
  for(j=0;j<n;j++)
    a[i][j] = rand();  // Fill a with random integers

// … Do something with this array

for(i=0;i<m;i++)
  free(a[i]); // Free memory allocated to row i
free(a);  // Free memory allocated to the row pointers
```

Observations:

- `k`-dimensional `int`-arrays can be declared as `int *...*a` (`k` asterisks)
- require nested loops (`k-1` levels) for calls to `malloc` and `free`

Dynamic memory allocation

Two-dimensional arrays in contiguous memory:

```
// Dynamic memory allocation:
int m=100, n=200;  // Array size (rows, columns)
int **a; // Our array (almost) as before
int *p;  // Auxiliary pointer
p = (int*) malloc(m*n*sizeof(int)); // m*n integers
     // in contiguous memory

// Allocate memory for row pointers:
a = (int**) malloc(m*sizeof(int*));

// Assign value to each row pointer:
int i,j;
for(i=0;i<m;i++)
  a[i] = p+i*n; // Move i rows beyond the start of a

// Go on as before...
for(i=0;i<m;i++)
  for(j=0;j<n;j++)
    a[i][j] = rand();
```

Dynamic memory allocation

Two-dimensional arrays in contiguous memory:

```
// Go on as before...
for(i=0;i<m;i++)
  for(j=0;j<n;j++)
    a[i][j] = rand();

// Releasing memory:
free(a);
free(p);
```

Passing parameters to functions

```
void myFunction(int u, double v); // Function prototype

int main(int argc, char *argv[]) {
  int a=0;
  double b=0.0;
  myFunction(a,b);
  // What values are a and b?

}


void myFunction(int u, double v) {

  u=1;

  v=3.14;

}
```

Passing parameters to functions

```
void myFunction(int u, double v); // Function prototype

int main(int argc, char *argv[]) {
  int a=0;
  double b=0.0;
  myFunction(a,b);  // Call by value (just like in Java)
  // What values are a and b?
  // Answer: Still a=0 and b=0.0
  // Parameter passing is by VALUE
}

void myFunction(int u, double v) {

  u=1;

  v=3.14;

}
```

Passing parameters to functions

```
void myFunction(int *u, double *v); // New function prototype

int main(int argc, char *argv[]) {
  int a=0;
  double b=0.0;
  myFunction(&a,&b);
  // What values are a and b?

}


void myFunction(int *u, double *v) {

  *u=1;

  *v=3.14;

}
```

Passing parameters to functions

```
void myFunction(int *u, double *v); // New function prototype

int main(int argc, char *argv[]) {
  int a=0;
  double b=0.0;
  myFunction(&a,&b);  // Call by reference (not in Java)
  // What values are a and b?
  // Answer: Changed to a=1 and b=3.14
}


void myFunction(int *u, double *v) {

  *u=1;

  *v=3.14;

}
```

Passing parameters to functions

```
void swap(double *u, double *v) {
  double tmp=*u;
  *u=*v;
  *v=tmp;
}
```

Use:

```
  int n=100;
  double *a;
  a = (double*)malloc(n*sizeof(double));
  // … fill a with real numbers
  // … let i and j be integers in 0..99
  swap(a+i, a+j);  // Swap a[i] and a[j]
  swap(&(a[i]), &(a[j])); // Would give the same result
```

Passing arrays to functions

```
void initialize(double *a, int size) {
  int i;
  for(i=0;i<size;i++)
    a[i] = 0.0;
}
```

Use:

```
  int n=100;
  double *a;
  a = (double*) malloc(n*sizeof(double));
  intialize(a,n); // note that a is a pointer to a double
```

Keyboard and file input

Let the user assign values to an int and a `double`:

```
int n;
double f;
printf("Enter an integer and a real number: ");
scanf("%d%lf", &n, &f);
```

Read an `int` and a `double` from the file `myData.txt`

```
int n;
double f;
// Open the file in input (r) mode
FILE *filePtr = fopen("myData.txt", "r");
if (filePtr)
  fscanf(filePtr, "%d%lf", &n, &f);
else
  printf("Could not open myData.txt for reading.\n");
```

File output

Save results in the textfile `myData.txt`

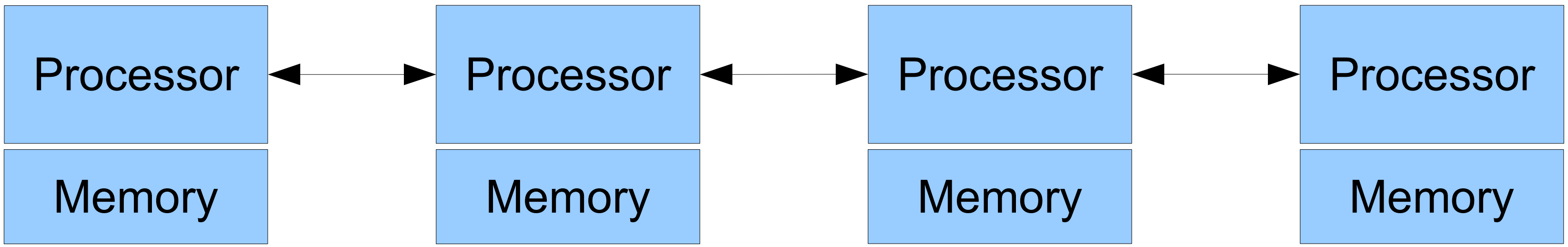
```
int n=100;
double f=3.14;
// Open the file in output (w) mode
FILE *filePtr = fopen("myData.txt", "w");
if (filePtr)
  fprintf(filePtr, "%d %lf", n, f);
else
  printf("Could not open myData.txt for writing.");
```

Getting Help

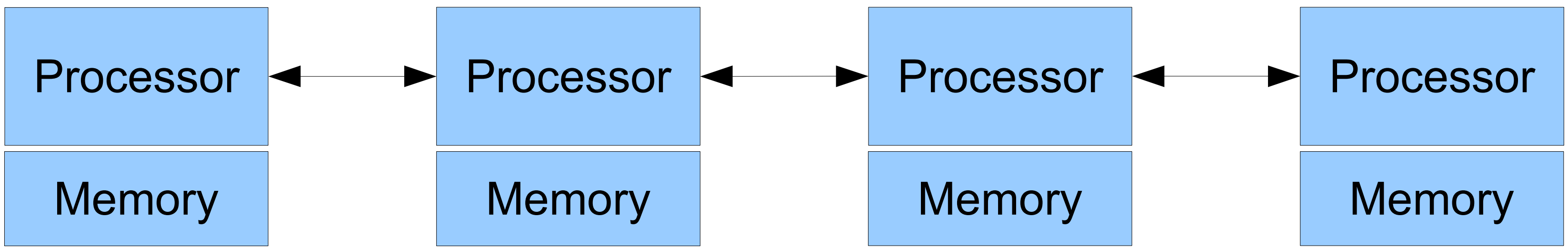
- Number of online C tutorials
  https://en.wikibooks.org/wiki/C_Programming
- Buy a book
- Look up manual pages:

```
>man printf

PRINTF(1)    User Commands          PRINTF(1)

NAME
       printf - format and print data

SYNOPSIS
       printf FORMAT [ARGUMENT]...
       printf OPTION

DESCRIPTION
       Print ARGUMENT(s) according to FORMAT.
```

Message Passing Interface (MPI)



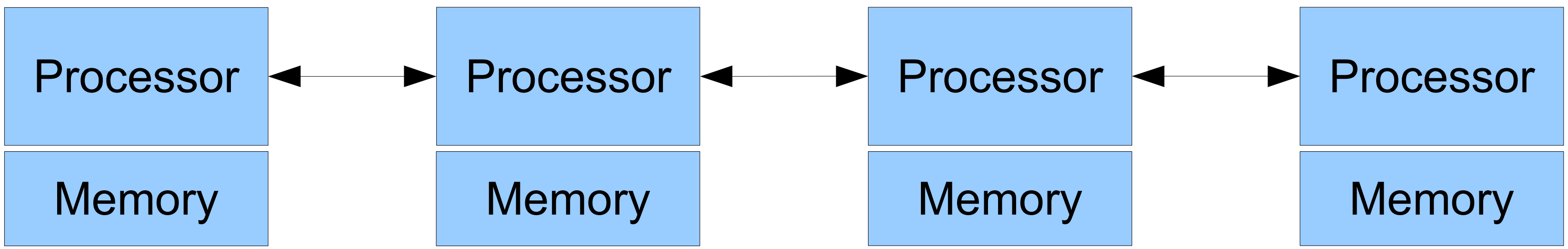
Message Passing Interface (MPI)



Everybody says hello:

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char *argv[]) {
  int rank; // Process ID, also called rank
  MPI_Init (&argc, &argv);                    // Start MPI session
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);  // Get current process id
  printf( "Hello world from process %d!\n", rank);
  MPI_Finalize();                             // End MPI session
}
```

Message Passing Interface (MPI)



Observations:

- To compile the program:
  `mpicc hello.c -o hello`
- To run the program on 4 processors:
  `mpirun -np 4 hello`
- The processes each hold a variable `rank`
  - with distinct values!
  - memory is not shared
- The function call `MPI_Comm_rank (MPI_COMM_WORLD, &rank)`
  - assigns the process ID to variable `rank`
  - IDs are in the range 0..3 (3 = `np-1`)
- The program might need to know `np`:
  - `int np;`
  - `MPI_Comm_size(MPI_COMM_WORLD, &np)`
- `MPI_COMM_WORLD` is the default *communicator*

MPI

Master-slave model:

- Process 0 is the master, processes `1..np-1` are the slaves
- The master reads (and preprocesses) the data
- The master sends one data segment to each slave
- Each slave receives one data segment from the master
- All processors (slaves and master) do their jobs concurrently
  - jobs are (typically) identical
  - data segments are distinct
  - data segments are (ideally) of equal size
- The master collects the results

MPI

Process 0 sends an int to processes 1,...,np-1:

```
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
MPI_Comm_size (MPI_COMM_WORLD, &np);
if (!rank) { // This block is executed only by the master
  printf("Hello world from process %d. I am the master.\n", rank);
  int i;
  for (i=1; i<np; i++) { // Enumerate the slaves
    value = 100 + rand()%np;  // Random number in 100..(100+np-1)
    printf("I send the value %d to process %d.\n", value, i);
    MPI_Send(&value, 1, MPI_INT, i, 1, MPI_COMM_WORLD);
  }
} else { // This block is executed by each slave
  MPI_Status status;  // To tell whether message is well received
  MPI_Recv(&value, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
  printf("Hello world from process %d. I am a slave,
    and I received the value %d from the master.\n", rank, value);
}
```

MPI

We used built-in MPI-functions for sending and receiving:

```
MPI_Send(&value, 1, MPI_INT, i, 1, MPI_COMM_WORLD);
```

`&value`	address of the first byte of the data to be sent (pointer to the data segment)
`1`	number of data elements in the segment to be sent
`MPI_INT`	type of data to be sent
`i`	destination process (receiver)
`1`	message tag, used for recognition in case of multiple messages
`MPI_COMM_WORLD`	communicator

```
MPI_Recv(&value, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
```

`&value`	where the received data are to be stored
`1`	number of data elements to be received
`MPI_INT`	type of data to be received
`0`	source process (sender)
`1`	message tag, must agree with sender's tag
`MPI_COMM_WORLD`	communicator
`status`	record (`struct`) with status information

MPI

Example: Process 0 sends a[4..7] to process 1:

```
int* a;
int tag=1;
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
if (!rank) { // This block is executed only by the master
  a = (int*)malloc(8*sizeof(int));
  int i;
  for (i=0; i<8; i++) a[i]=i*i;
  MPI_Send(a+4, 4, MPI_INT, 1, tag, MPI_COMM_WORLD);
  // Here we might process a[0..3]
} else if (rank==1) { // This block is executed by process 1
  a = (int*)malloc(4*sizeof(int));
  MPI_Status status;
  MPI_Recv(a, 4, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);
  // Here we might process a[0..3]
}
```