Lecture 2 outline

- **Control flow**
  - Imperative languages
  - Goto (considered harmful)
  - Blocks, conditionals, loops

- **State**
  - Variable
  - Data types
  - Static vs dynamic typing

- **Compiled vs interpreted language**
  - MATLAB functions, subfunctions, toolboxes
  - C/C++ declaration, definition, objective, and executable files

- **Practical notes**
  - Clean vs obfuscated code
  - Avoid cut & paste

---

Control flow

An imperative program is a list of statements (instructions).

Statements are executed sequentially

- incrementing the program counter

- 10 sleep eight hours
- 11 wake up
- 12 have breakfast

Branching statements

- allow for non-sequential execution, conditionally on the state
- reset the program counter

- 13 if today is Saturday goto 10
- 14 leave home
Control flow with goto

**BASIC** example

```
10 i ← 0
20 i ← i + 1
30 print i, “ squared is “, i * i
40 if i >= 10 then goto 60
50 goto 20
60 print “that’s all folks!”
```

To add a line of code use an intermediate line number (!)

```
10 i ← 0
20 i ← i + 1
30 print i, “ squared is “, i * i
35 print i, “ cubed is “, i * i * i
40 if i >= 10 then goto 60
50 goto 20
60 print “that’s all folks!”
```

Use labels to abstract line numbers and simplify the use of goto

A label is just a name given to a statement in the sequence

```
i ← 0
more: i ← i + 1
print i, “ squared is “, i * i
if i >= 10 then goto end
goto more
end: print “that’s all folks!”
```


**Structured Control Flow**

Give up goto altogether. Any program can be expressed in term of three simple control structures [Böhm-Jacopini 66]:

- **Blocks** (sequences of executable statements)
  ```
  { do_this ; do_that ; do_something_else ; }
  ```

- **Conditionals** (execute a block if condition is true)
  ```
  if (condition) { }
  ```

- **Loops** (keep executing a block until condition is true)
  ```
  while (condition) { }
  ```

**Spaghetti monster**

```
i ← 0
more: i ← i + 1
print i, “ squared is “, i * i
if i >= 10 then goto end
goto more
end: print “that’s all folks!”
```

**Structured program**

```
i ← 0
while (i < 10)
{
  i ← i + 1
  print i, “ squared is “, i * i;
}
print “that’s all folks!”
```

**Obvious flow**

The code is much easier to understand because each block has only one entry point at the beginning and only one exit point at the end.

**Example**

**Spaghetti monster**

```
i ← 0
more: i ← i + 1
print i, “ squared is “, i * i
if i >= 10 then goto end
goto more
end: print “that’s all folks!”
```

**Structured program**

```
i ← 0
while (i < 10)
{
  i ← i + 1
  print i, “ squared is “, i * i;
}
print “that’s all folks!”
```
Structured Control Flow: Procedures

A way to create a software module or component is to wrap a sequence of statements into a **procedure**.

A procedure implements a reusable functionality (behavior) hiding the internal implementation details.

**Examples**
- \( y = \tan(x) \) // compute the tangent of a number
- `printf("a string")` // display a string on the screen
- `window = createWindow()` // create a new window on the display
- `destroyWindow(window)` // destroy it

A procedure implements a reusable functionality (behavior) hiding the internal implementation details.

**Lecture 2 outline**
- Control flow
  - Imperative languages
    - Goto (considered harmful)
    - Blocks, conditionals, loops
  - State
    - Variable
    - Data types
    - Static vs dynamic typing
- Compiled vs interpreted language
  - MATLAB functions, subfunctions, toolboxes
  - C/C++ declaration, definition, objective, and executable files
- Practical notes
  - Clean vs obfuscated code
  - Avoid cut & paste

**C version**
```c
#include <stdio.h>

void print_n_squared_numbers(int n)
{
    int i = 0;
    while (i < n)
    {
        i += 1;
        printf("%d squared is %d\n", i, i * i);
    }
}
```

**MATLAB version**
```matlab
function print_n_squared_numbers(n)
    i = 0;
    while i < n
        i = i + 1;
        fprintf('%%d squared is %%d\n',i,i*i);
    end
    fprintf('that's all folks!\n');
end
```

Both MATLAB and C are **imperative** and **procedural**.

MATLAB is **interpreted**, C/C++ is **compiled**

MATLAB is **dynamically typed**, C/C++ is **statically typed**.

**Statements and the state**
- **State** = program counter + content of memory
- **Executing a statement changes the state**
  - updates the program counter
  - almost always modifies the content of the memory too
- **Example**
  ```
  50 * x * x + y ; // result is not remembered, no effect
  z = 50 * x * x + y ; // write the result to the variable z
  ```
- If a statement does not alter the content of the memory, it has essentially no effect
- **Exceptions**:
  - wasting time
  - in MATLAB, displaying a value on the screen
  - ...
A computer’s memory is just a large array of words (strings of bit).

Addresses

00000000
00000001 ...

milesToGo: 0x001F034C

MEMORY (4GB)

001F034C: [66 72 65 64]

The meaning depends on how it is interpreted:

32 bit integer  1718773092
4 characters “fred”
floating point  1.68302e+022

Write a 32-bit integer to memory in C

*((int*)0x001F034C) = 1718773092;

Structured version: using a variable

milesToGo = 1718773092;

A variable is just a label to a memory address.

Dynamic typing

Consider the following MATLAB fragment

% x, y, and z are stored as 64-bit float
x = 5;
y = 10;
z = x * y;

Each variable stores both:
- the address of the data in memory and
- the type of the data.

Use MATLAB command who to get a list of variables and their types:

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Bytes</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>1x1</td>
<td>8</td>
<td>double</td>
</tr>
<tr>
<td>y</td>
<td>1x1</td>
<td>8</td>
<td>double</td>
</tr>
<tr>
<td>z</td>
<td>1x1</td>
<td>8</td>
<td>double</td>
</tr>
</tbody>
</table>

Dynamic typing

Consider the following MATLAB fragment

% x, y, and z are stored as 64-bit float
x = 5;
y = 10;
z = x * y;

A (data) type specifies
- a set of possible values
e.g. integers in the range ~2,147,483,648 to 2,147,483,647
- what one can do with them
e.g. create, assign, sum, multiply, divide, print, convert to float, ...

A data type representation specifies how values are actually stored in memory
e.g. integer is usually represented as a string of 32 bits, or four consecutive bytes, in binary notation

This is another example of abstraction
You never have to think how MATLAB represents numbers to use them!

Every language support an array of primitive data types:

- MATLAB: numeric arrays (characters, integer, single and double precision),
  logical arrays, cell arrays, ....
- C: various integer types, character types, floating point types, arrays, strings, ...

Memory and variables

Types
Consider the following MATLAB fragment

```
% x, y, and z are stored as 64-bit float
x = 5;
y = 10;
z = x * y;
```

Now reassign x and z

```
x = 'Oxford U';
z = x * y;
```

What is z?

### Overhead in dynamic typing

In dynamic typing each a variable is associated to both the **actual data record** as well as **metadata** describing its type.

While usually this is not a problem, in some cases the overhead may be significant.

Example: MATLAB uses about 80 bytes to store the data type descriptor.

- Storing one array of 1 million numbers uses 80 + 8 * 1e6 bytes (~ 7.6 MB, efficiency ~ 100%)
- Storing 1 million arrays of 1 number each uses (80 + 8) * 1e6 bytes (~ 83 MB, efficiency ~ 9%)

### Static typing and variable declaration

In C variables must be declared before they can be used.

A declaration assigns **statically a data type** to a variable

**Examples**

```
int anInteger; /* usually 32 bits length, but implementation dependent */
unsigned int anUnsignedInteger;
char aCharacter;
double aFloat;
int32_t a32BitInteger; /* C99 and C++ */
int16_t a16BitInteger;
```

**Statically-typed variables**

- have a well defined type before the program is run
- incorporate constraints on how a variable can be used

Static typing allows for

- smaller run-time overhead in handling variables
- better error checking before the program is run
MATLAB: program organisation

MATLAB procedures are called functions

A MATLAB procedure is a function.

A MATLAB function is stored in a homonymous file with a .m extension

File: print_ten_squared_numbers.m

```matlab
function print_ten_squared_numbers(n)
i = 0;
while i < n
    i = i + 1;
    fprintf('%d squared is %d\n', i, i*i);
end
end
```

File: my_script.m

```matlab
% demonstrates the use of a function
print_ten_squared_numbers();
```

A .m file can also contain a script.

A script does not define a function. It is more similar to cutting & pasting code into the MATLAB prompt.

MATLAB: program organisation

MATLAB procedures are called functions

A MATLAB function is stored in a homonymous file with a .m extension

File: print_ten_squared_numbers.m

```matlab
function print_ten_squared_numbers(n)
i = 0;
while i < n
    i = i + 1;
    fprintf('%d squared is %d\n', i, i*i);
end
end
```

File: my_script.m

```matlab
% demonstrates the use of a function
print_ten_squared_numbers();
```

A .m file defines a function that can be accessed by functions and scripts in other files.

A .m file can contain also any number of local functions.

Local functions are only visible from the file where they are defined.¹

¹Advanced techniques allow to pass references to local functions, so that they can be called from other files.
MATLAB: grouping related functions

- Put related functions into a given directory

  **Directory**: drawing/
  - drawAnArc.m
  - drawAnArrow.m
  - drawACircle.m

  **Directory**: math/
  - tan.m
  - atan.m
  - sqrt.m

  **Directory**: pde/
  - euler.m

- Use MATLAB `addpath()` function to include directories in MATLAB search path and make these functions visible.

- **MATLAB Toolboxes** are just collections of functions organised in directories.

---

C/C++: program organisation

- C/C++ explicitly support the notion of **modules**.

  A module has two parts:
  - the **declaration** (.h), defining the interface of the functions i.e. the function names and the types of the input and output arguments
  - the **definition** (.c), containing the actual implementation of the functions

  **file**: usefulstuff.h
  ```
  #include "usefulstuff.h"
  
  void print_n_squared_numbers(int n) {
    int i = 0;
    while (i < n) {
      i = i + 1;
      printf("%d squared is %d\n", i, i*i);
    }
    printf("that's all folks!\n");
  }
  
  int get_an_awesome_number() {
    return 42;
  }
  ```

  **file**: usefulstuff.c

  **file**: myprogram.c
  ```
  #include "usefulstuff.h"
  
  int main(int argc, char** argv) {
    print_n_squared_numbers(10);
    get_an_awesome_number();
    printf("myprogram\n");
    return 0;
  }
  ```

---

C/C++: compiling a program

- Run the compiler **cc**
  - Each .c file is **compiled** into an object file .o
  - This is the binary translation of a module

  - Run the linker, usually also implemented in **cc**
  - The .o files are merged to produce an executable file

  **declaration file**: usefulstuff.h
  **definition file**: usefulstuff.c
  **object file**: usefulstuff.o
  **executable file**: myprogram

---

C/C++: compiling a program

- Run the compiler **cc**
  - Each .c file is **compiled** into an object file .o
  - This is the binary translation of a module

  - Run the linker, usually also implemented in **cc**
  - The .o files are merged to produce an executable file

  **declaration file**: morestuff.h
  **definition file**: morestuff.c
  **object file**: morestuff.o
  **executable file**: myprogram
More on declaring, defining, and calling functions

Declaration of the function prototype

```c
void print_n_squared_numbers(int n);
```

Definition of the function implementation

```c
void print_n_squared_numbers(int n)
{
    /* do something */
}
```

Invocation of the function

```c
print_n_squared_numbers(10);
```

Declaring a function

- defines its prototype \( = \{\text{name of the functions, type of input/output parameters}\}\)
- specifies the interface
given the prototype, the compiler knows how to call the function

Defining a function specifies its implementation. The parameters are said to be formal as their value is not determined until the function is called.

Calling a function starts executing the function body. The parameters are said to be actual because they are assigned a value.

Return value(s)

- In C functions have a single output value, assigned by the `return` statement.
- In MATLAB functions have an arbitrary number of output values.
  - They get their value from homonymous variables.

```c
int get_awesome_number()
{
    return 42;
}
```

```c
int x;
```

```c
x = get_awesome_number();
/* x value is now 42 */
```

Some practical notes

- The look is important
  - Use meaningful variable names
  - Use comments to supplement the meaning
  - Indent code for each block/loop

- Avoid to cut and paste code
  - Use functions to encapsulate logic that can be reused
  - Cutting and pasting code leads to guaranteed disasters
  - because when you need to change the code, you need to change all the copies!

- Top-down vs bottom-up
  - Design top-down
  - Code bottom-up or top-down, or a combination

Control flow

- Imperative languages
- Goto (considered harmful)
- Blocks, conditionals, loops

State

- Variable
- Data types
- Static vs dynamic typing

Compiled vs interpreted language

- MATLAB functions, subfunctions, toolboxes
- C/C++ declaration, definition, objective, and executable files

Practical notes

- Clean vs obfuscated code
- Avoid cut & paste
Here is a valid C program (http://en.wikipedia.org/wiki/Obfuscation_(software))

```c
char M[3], A, Z, E = 40, J[40], T[40];
main(C) {
    for (*J = A = scanf("%d", &C);
        --E;
        printf("._");
    for (; (A -= Z = !Z) ||
        printf("\n")
        , A = 39, C --
        )
        , Z ||
        printf(M[Z] = Z - (E = A[J - Z]) && C
        & A == T[A]
        | 6 << rand()
    ];
}
```

Can you figure out what this does?