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 → PC 12
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!"
```

Labels

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) { }
  ```

Structured program

The code is much easier to understand because each block has
- **only one entry point at the beginning**
- **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!"
```
A way to create a software module or component is to wrap a sequence of statements in a **procedure**.

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

A procedure implements a reusable functionality (behaviour) **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

```
C version
#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);
    }
    printf("that’s all folks!\n");
}
```

```
MATLAB version
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**.
Memory and variables

- A computer's memory is just a large array of words (strings of bits).

Addresses

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>...</td>
</tr>
<tr>
<td>001F034C</td>
<td>Memory (4GB)</td>
</tr>
</tbody>
</table>

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:

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

Structured version: using a variable

```c
milesToGo = 1718773092;
```

A variable is just a label to a memory address (and a data type).

Types

- 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 supports 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, ...

Dynamic typing

Consider the following MATLAB fragment:

```matlab
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>

Consider the following MATLAB fragment:

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

Now the variable x refers to
- a new memory block and
- a different data type

In MATLAB, the data type associated to a variable can be determined only at run-time, i.e. when the program is executed

This is called **dynamic typing**

### 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

```c
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
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

MATLAB is an interpreted language
A MATLAB program is executed by an interpreter
the interpreted emulates a CPU capable of understanding MATLAB instructions
significant overhead at run-time

C and C++ are compiled languages
A C/C++ program must be translated by a compiler into an executable format before it can be executed
no overhead at run-time
the compiler can spot programming error violating assumptions expressed in the code
for example, in a statically typed language the compiler can check that operations on variables involve data of the correct types

Example
Compiling the following fragment generates an error because the multiplication of an integer and a pointer (see later) is not defined:

```c
int *aPointerToInt = 0;
int anInt = 10;
int anotherInt = anInt * aPointerToInt;
```

```plaintext
error-pointer-by-integer.c:7:
error: invalid operands to binary *(have 'int*' and 'int')
```

MATLAB: program organisation

- MATLAB procedures are called functions

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

```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
    fprintf('that\'s all folks!\n');
end
```

```matlab
function thats_all()
    fprintf('that\'s all folks!\n');
end
```

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

```
#include "usefulstuff.h"

int get_an_awesome_number() {
    return 42;
}
```

```
#include <stdio.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");
}
```

```
#include "usefulstuff.h"

int main(int argc, char** argv) {
    print_n_squared_numbers(10);
}
```

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

```
file: usefulstuff.h

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

```
file: usefulstuff.c

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");
}
```

```
file: myprogram.c

#include "usefulstuff.h"
#include <stdio.h>

int get_an_awesome_number() {
    return 42;
}
```

```
file: myprogram.o
```

---

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

```
file: morestuff.h

int get_an_awesome_number() {
    return 42;
}
```

```
file: morestuff.c

int get_an_awesome_number() {
    return 42;
}
```

```
file: myprogram.o
```

```
file: myprogram.o
```

```
file: myprogram.o
```

---

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

```
file: usefulstuff.o
```

```
file: myprogram.o
```

```
file: morestuff.o
```

```
file: myprogram.o
```

```
file: myprogram.o
```
More on declaring, defining, and calling functions

Declaring a function

- defines its **prototype** = \{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.

---

**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);
```

---

In C functions have a single output value, assigned by the **return** statement.

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

**Invocation of the function**

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

---

In MATLAB functions have an arbitrary number of output values.

- They get their value from homonymous variables.

```matlab
function [a,b,c] = get_many_numbers()
    a = 42;
    b = 3.14;
    c = +inf;
    return;
end
```

**Invocation of the function**

```matlab
[x,y,z] = get_many_numbers();
```

---

% get eigenvectors and eigenvalues

```matlab
[V, D] = eig(A);
```

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)](http://en.wikipedia.org/wiki/Obfuscation_(software))):

```c
```

Can you figure out what this does?