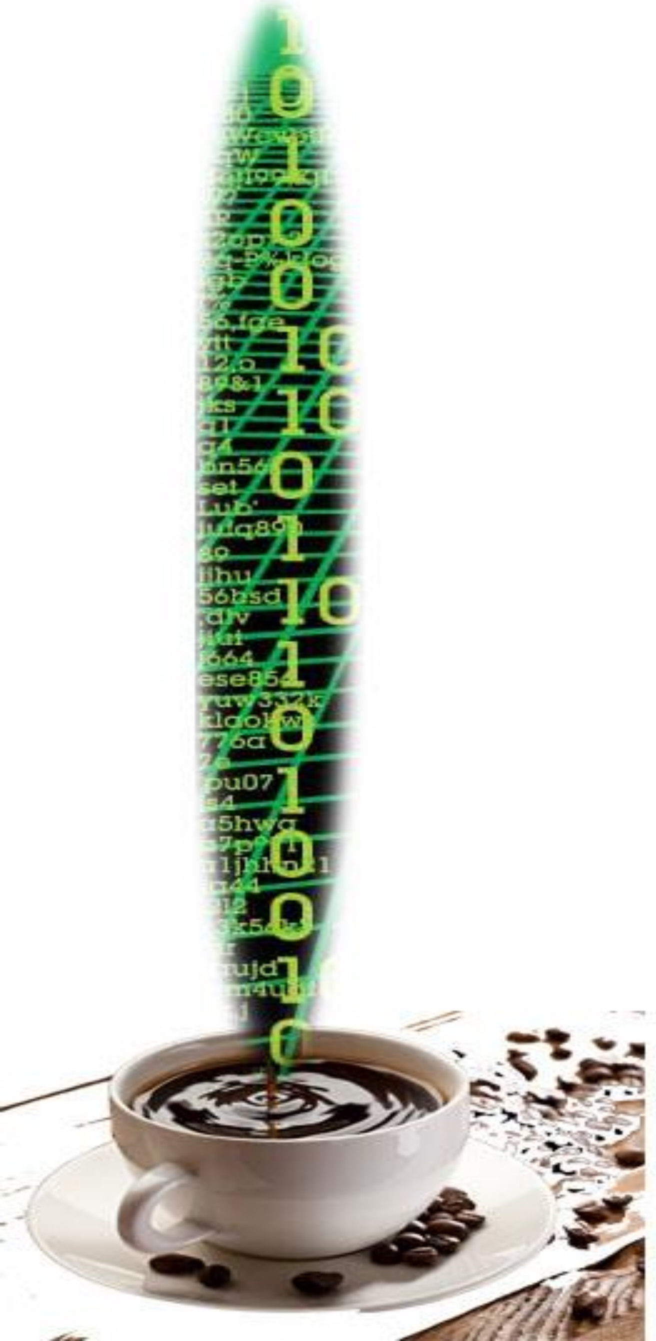
0 1 2 3 4 5 6 7 8 9 A B C D E F



Bits & Bytes

Data – Memory – Pointers

Programming with C++

Dimitrios N. Terzopoulos

terzopod@math.auth.gr

14/10/2015

0 1 2 3 4 5 6 7 8 9 A B C D E F

Contents (1) – Data / Memory

- Processor Registers, Cache & Instructions – Assembly
- Stack / Heap (Function Calls)
- Bit/Byte Endianness (wtf???)
- File Structure – Running Processes – Address Space
- Hacking through Memory

0 1 2 3 4 5 6 7 8 9 A B C D E F

Practice (1) – Pointers in C++

- 0x0C**FE**02AC & (0xFF << 16)) >> 16) = 0x**FE**
- Uninitialized Variables (Garbage)
- Global vs. Local Variables
- Return Value Optimization & Pointer Misassignment
- #define directive, typedef keyword

0 1 2 3 4 5 6 7 8 9 A B C D E F

Practice (2) – Pointers in C++

- Call by Value vs. Call by Reference ("Swap Problem")
- Fixed-size Arrays vs. Variable-size Arrays & Containers
- Pointers to Pointers
- Pointers to Functions
- Casts and Data Reinterpretation

0 1 2 3 4 5 6 7 8 9 A B C D E F

Motivation

- Too much Theory in *Theoretical Information Science*(!)
- Prestige(!)
- Code Elegance & Readability (The FORTRAN curse)
- Development Performance & Productivity
- Code Reusability

0 1 2 3 4 5 6 7 8 9 A B C D E F

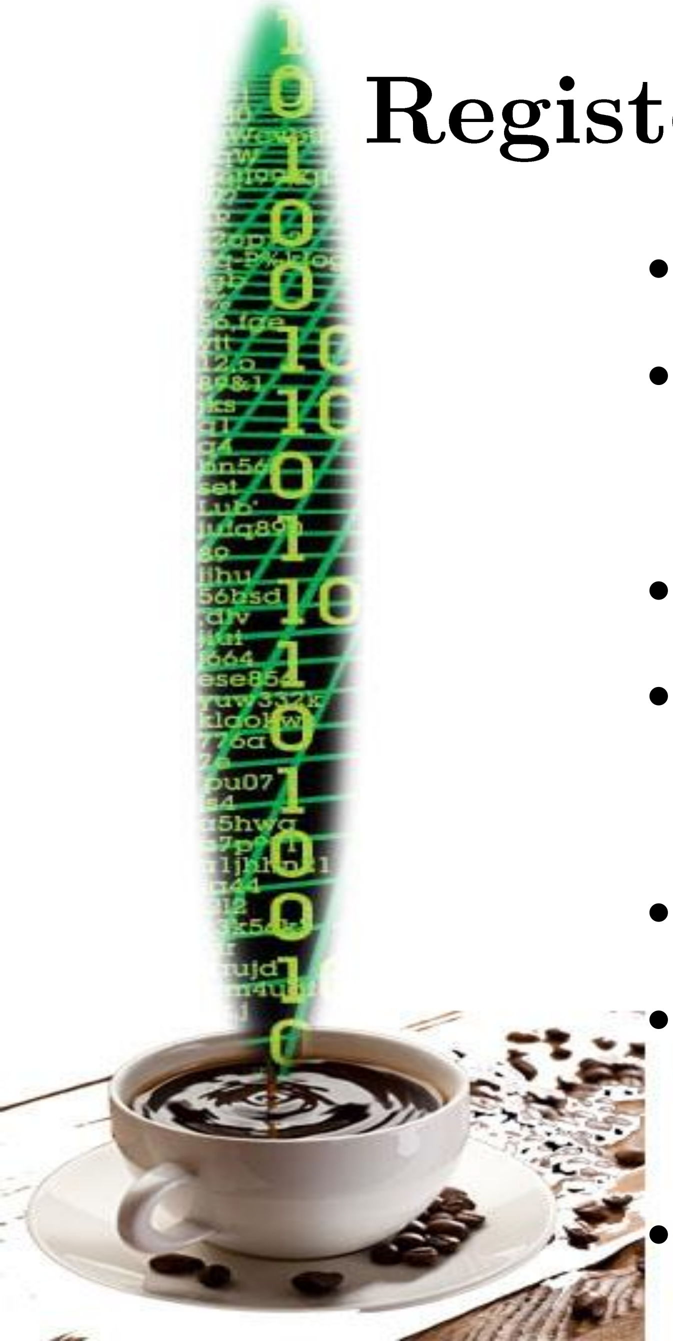
Russian Multiplication

```
int M(int a, int b) {
    int result = 0;
    while (a) { if (a&1) {result+=b;} a>>=1; b<<=1; }
    return result;
}
```

International Obfuscated C Code Contest

0 1 2 3 4 5 6 7 8 9 A B C D E F

Registers – Cache – Instructions – Assembly



- Registers → Like local variables for the processor.
- Registers → Their size depends on Proc. Architecture.

- Cache → Small temporary memory space for duplicates.
- ***Cache hit*** vs. ***Cache miss***

- Instructions → Bits that are an order for the processor.
- Instructions → They make up Machine code (Bytes).

- Assembly → Compacted “baptized” Machine Code.

0 1 2 3 4 5 6 7 8 9 A B C D E F

Motivational Example (Registers)

- 32-bit Architecture ⇔ 32-bit ***register*** variables.
- 32-bit variable → Max Value $= 2^{32} = 4294967295$.
- How much is that??? → $2^{32} = (2^2)\cdot(2^{10})^3 = 4\cdot(1024)^3 \ldots$

  $\ldots = 4294967295$ values.

If each value points to one 1-byte memory cell, that's…

4 GB of ***Addressable*** Memory!

0 1 2 3 4 5 6 7 8 9 A B C D E F

Stack & Heap in Memory

- Stack → „Smaller“ Memory Area for ***local*** operations.
- Stack → Functions place local variables on the Stack.

When stack is full → *Stack Overflow*

- Heap → Large Memory Chunk for ***global*** operations.
- Heap → Hosts ***dynamically assigned*** variables.

If heap memory is not reclaimed → *Memory Leaks*

„Memory“ usually implicitly refers to (*you guessed it*) RAM!

0 1 2 3 4 5 6 7 8 9 A B C D E F

Motivational Example (Stack / Heap)

```
int Func() {
        int d = 20;           // d Is a local variable
        return d*d;           // allocated on the stack.
}                             // It "dies" afterwards.

int Func2() {
        static int w = 50;    // w is a global variable
        return (w/4);         // allocated on the heap
}                             // It never "dies" until we
                              // "kill" it.
```

0 1 2 3 4 5 6 7 8 9 A B C D E F

Bit/Byte Endianness

- 1 Byte = 8 Bits
- 1 Short = 2 Bytes = 1 ***Word*** (Windows API)
- 1 Integer = 4 Bytes = 1 ***Dword***

etc... (but not always – sizes are machine dependent).

- Let intVal = $(\mathbf{A2\ B1\ 12\ 3D})_{16} = (2729513533)_{10}$

The same integer in two different representations.

- Little Endian → 3D 12 B1 A2
- Big Endian → A2 B1 12 3D

- Endianness → The order of Byte Interpretation.

(From *Gulliver's Travels*)

0 1 2 3 4 5 6 7 8 9 A B C D E F

File Structure – Running Processes – Address Space

- Files are made of bytes (honestly?).

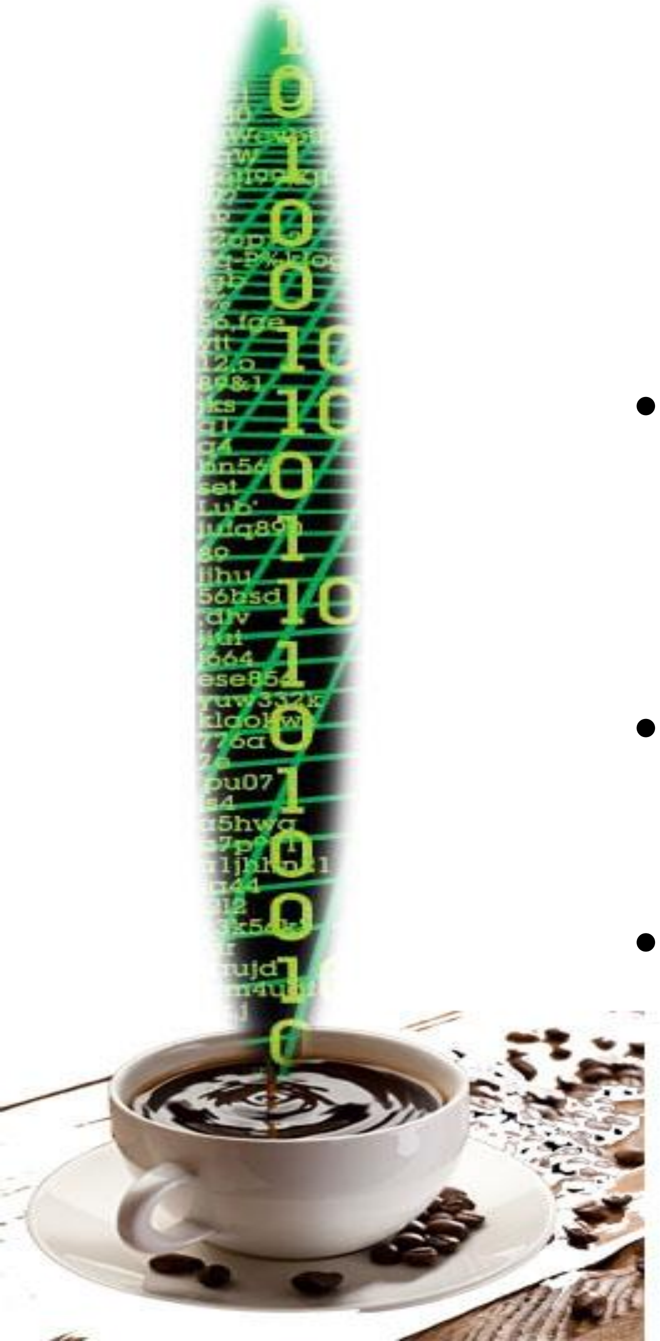
- The meaning of these bytes is ***Contextual***.

- Programs rely on "*Grouped Byte Interpretation*".
  (Usually defined in *Standards* and described in *headers*)

- File contents can be viewed using a ***Hex Editor***.

0 1 2 3 4 5 6 7 8 9 A B C D E F

File Structure – Running Processes – Address Space



- Running Processes are also ... Bytes! Loaded in RAM.
  (It is Machine Code loaded and executed)

- Windows PE (Portable Executable) Format.

- Each Process runs in a *Virtual Address Space.*

Knowing what the Address Space content *looks like* allows for "*tampering*" with Processes.

0 1 2 3 4 5 6 7 8 9 A B C D E F

And some *fun*...

- **Code Injection** (Run code in the address space of a running process)
- **Self-Modifying Code** (Code is mutated at runtime to become malicious).
- **Disassemble / Decompile** (Retrieving the original source code in some form).
- **DLL Proxy** (Build a proxy to incorporate desired behavior in some already existing library)

0 1 2 3 4 5 6 7 8 9 A B C D E F

References

- Patterson, D.A. and Hennessy, J.L. (2014). *Computer Organization and Design* (*5th Edition*): *The Hardware / Software Interface*. The Morgan Kaufmann Series in Computer Architecture and Design, Elsevier, Oxford.

- Eckel, B. (2000). *Thinking in C++* (2 Volumes). Prentice Hall, Pearson Higher Education, New Jersey.

0 1 2 3 4 5 6 7 8 9 A B C D E F

Thank you for your attention!

...on for the Practice