You will have seen...

- Courses:
  - Programming & software engineering
  - Parallel programming
  - Data structures and algorithms
- Languages: Eiffel, Java, C#, ...
- Constructs and techniques:
  - Object-orientation
  - Design-by-contract
  - Locks, atomicity, OpenMP, etc.

This course covers in depth...

- Programming in C
  - Still the systems programming language of choice
- Programming in Assembly Language
  - What the machine understands
- What really happens at the bit-level
  - Machine instructions
  - Memory systems
  - I/O devices
- Basic elements of processor design
- What makes things go fast (and slow)

Hold on tight!

Course Components

- Lectures
  - Higher level concepts and ideas
- Recitations
  - Applied concepts, important tools and skills for labs, clarification of lectures, exam coverage, C tutorial
- Lab exercises
  - The heart of the course
  - 1 week each (sometimes 2 weeks)
  - Provide in-depth understanding of aspects of systems
  - Programming and measurement
- Exam (100% of grade)
  - Test your understanding of concepts

Language

- I’ll teach in English (and C...)
  - If I speak too fast, or say something unclear, raise your hand!
  - Please ask questions!
- Assistants’ groups are 5 x German, 2 x English
  - So far...
- Examination:
  - Paper will be in English
  - Answers may be in German or English

Logistics

- Lectures here in CAB G.61
  - Tuesday, Wednesday 10:00 – 12:00
- Recitations – very important!
  - Thursday 13:00 – 15:00, IFW various rooms
  - Learn C, simulator, tools
  - Briefings for Lab exercises
  - Knowledge needed for exams, but not in the lectures!
- There will be a session this Thursday
  - Sign up sheets at the front of the room today
  - Check the course web page on Wednesday
More logistics

- Web site: [http://www.systems.ethz.ch/courses/fall2012/SPCA](http://www.systems.ethz.ch/courses/fall2012/SPCA)
  - Lecture notes should appear in advance on web site
  - The notes are not intended to be understood without lectures...
- Procedure for answering additional questions:
  1. Ask your friends
  2. Check the web
  3. Ask your teaching assistant
  4. Ask another teaching assistant
  5. Email me (troscoe@inf.ethz.ch)

Acknowledgements

- Course based on CS 15-213 at Carnegie Mellon University
  - Lots of material gratefully borrowed from CMU
- New material: multicore, devices, etc.
  - All my fault 😊

Textbooks

- Randal E. Bryant and David R. O'Hallaron,
  - [http://csapp.cs.cmu.edu](http://csapp.cs.cmu.edu)
- Brian Kernighan and Dennis Ritchie,

Motivation

- Most CS courses emphasize abstraction
  - Abstract data types (objects, contracts, etc.)
  - Asymptotic analysis (worst-case, complexity)
- These abstractions have limitations
  - Often don’t survive contact with reality
  - Especially in the presence of bugs
  - Need to understand details of underlying implementations

Goals

- Become more effective programmers
  - Find and eliminate bugs efficiently
  - Understand and tune for program performance
- Prepare for later systems classes at ETHZ
  - Compilers, Operating Systems, Networks, Computer Architecture, Embedded Systems

“Systems” as a field

- Encompasses:
  - Operating systems
  - Database systems
  - Networking protocols and routing
  - Compiler design and implementation
  - Distributed systems
  - Cloud computing & online services
- On and above hardware/software boundary
Motivation: 5 realities

**Reality #1:**

- **int's** are not integers.
- **float's** are not real numbers.
- **ints** are not integers, **floats** are not reals
  - Is $x^2 \geq 0$?
    - floats: Yes!
    - ints:
      - $40000 \times 40000 \rightarrow 1600000000$
      - $50000 \times 50000 \rightarrow ??$
  - Is $(x + y) + z = x + (y + z)$?
    - unsigned & signed ints: Yes!
    - floats:
      - $(1e20 + -1e20) + 3.14 \rightarrow 3.14$
      - $1e20 + (-1e20 + 3.14) \rightarrow ??$

Code security example

```c
/* Define KSIZE to 1024 */
#define KSIZE 1024

char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

- Similar to code found in a version of FreeBSD's implementation of `getpeername`
- There are legions of smart people trying to find vulnerabilities in programs
Typical usage

```c
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```c
#define MSIZE 528
void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, MSIZE);
    printf("%s\n", mybuf);
}
```

Malicious usage

```c
/* Kernel memory region holding user-accessible data */
#define KSIZE 1024
char kbuf[KSIZE];

/* Copy at most maxlen bytes from kernel region to user buffer */
int copy_from_kernel(void *user_dest, int maxlen) {
    /* Byte count len is minimum of buffer size and maxlen */
    int len = KSIZE < maxlen ? KSIZE : maxlen;
    memcpy(user_dest, kbuf, len);
    return len;
}
```

```c
#define MSIZE 528
void getstuff() {
    char mybuf[MSIZE];
    copy_from_kernel(mybuf, -MSIZE);
    ...
}
```

Computer arithmetic

- Does not generate random values
  - Arithmetic operations have important mathematical properties
- Cannot assume all “usual” mathematical properties
  - Due to finiteness of representations
  - Integer operations satisfy “ring” properties
    - Commutativity, associativity, distributivity
  - Floating point operations satisfy “ordering” properties
    - Monotonicity, values of signs
- Observation
  - Need to understand which abstractions apply in which contexts
  - Important issues for compiler writers and serious application programmers

Reality #2:
You’ve Got to Know Assembly

- You’ve got to know assembly
  - Chances are, you’ll never write program in assembly
    - Compilers are much better & more patient than you are
  - But: understanding assembly is key to machine-level execution model
    - Behavior of programs in presence of bugs
      - High-level language model breaks down
    - Tuning program performance
      - Understand optimizations done/not done by the compiler
      - Understanding sources of program inefficiency
  - Implementing system software
    - Compiler has machine code as target
    - Operating systems must manage process state
  - Creating / fighting malware
    - x86 assembly is the language of choice!

Assembly code example

```c
double t;
start_counter();
P();
t = get_counter();
printf("P required %f clock cycles\n", t);
```
Code to read counter

- Write small amount of assembly code using GCC's asm facility
- Inserts assembly code into machine code generated by compiler

```c
static unsigned cyc_hi = 0;
static unsigned cyc_lo = 0;

/* Set *hi and *lo to the high and low order bits of the cycle counter. */
void access_counter(unsigned *hi, unsigned *lo)
{
    asm(
        "rdtsc; movl %%edx,%0; movl %%eax,%1
        : "=r" (*hi), "=r" (*lo)
        : "%edx", "%eax" );
}
```

Reality #3:
Memory matters.
RAM is an unrealistic abstraction.

Memory matters

- Memory is **not unbounded**
  - It must be allocated and managed
  - Many applications are memory-dominated
- Memory referencing bugs especially pernicious
  - Effects are distant in both time and space
- Memory performance is **not uniform**
  - Cache and virtual memory effects can greatly affect program performance
  - Adapting program to characteristics of memory system can lead to major speed improvements

Memory referencing bug

```c
double fun(int i)
{
    volatile double d[1] = {3.14};
    volatile long int a[2];
    a[i] = 1073741824; /* Possibly out of bounds */
    return d[0];
}
```

fun(0) → 3.14
fun(1) → 3.14
fun(2) → 3.1399998664856
fun(3) → 2.00000061035156
fun(4) → 3.14, then segmentation fault

Explanation:
```
Saved State
  d7 … d4
  d3 … d0
a[1]
a[0]
```

Location accessed by fun(i)

Memory referencing errors

- C and C++ do not provide any memory protection
  - Out of bounds array references
  - Invalid pointer values
  - Abuses of malloc/free
- Can lead to nasty bugs
  - Whether or not bug has any effect depends on system and compiler
  - Action at a distance
    - Corrupted object logically unrelated to one being accessed
    - Effect of bug may be first observed long after it is generated
- How can I deal with this?
  - Program in C#, or ML, or Scala, or Haskell, or...
  - Understand what possible interactions may occur
  - Use or develop tools to detect referencing errors

Memory system performance

```c
void copyji(int src[2048][2048], int dst[2048][2048])
{
    int i, j;
    for (j = 0; j < 2048; j++)
        for (i = 0; i < 2048; i++)
            dst[i][j] = src[i][j];
}
```

21 times slower (Pentium 4)

- Hierarchical memory organization
- Performance depends on access patterns
  - Including how step through multi-dimensional array
Reality #4:
There’s much more to performance than asymptotic complexity

• Constant factors matter too!
• Even exact op count does not predict performance
  – Easily see 10:1 performance range depending on how code written
  – Must optimize at multiple levels: algorithm, data representations, procedures, and loops
• Must understand system to optimize performance
  – How programs compiled and executed
  – How to measure program performance and identify bottlenecks
  – How to improve performance without destroying code modularity and generality

Example: matrix multiplication

Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz (double precision) Gflop/s

• Standard desktop computer, vendor compiler, using optimization flags
• Both implementations have exactly the same operations count (2n^3)
• What is going on?

MMM plot: analysis

Matrix-Matrix Multiplication (MMM) on 2 x Core 2 Duo 3 GHz Gflop/s

• Reason for 20x: Blocking or tiling, loop unrolling, array scalarization, instruction scheduling, search to find best choice
• **Effect:** less register spills, less L1/L2 cache misses, less TLB misses

Reality #5:
Computers don’t just execute programs

This processor is a little old, but it’s the same on new ones!
Computers don’t just run programs

- They need to get data in and out
  - I/O system critical to program reliability and performance
- They communicate with each other over networks
  - Many system-level issues arise in presence of network
    - Concurrent operations by autonomous processes
    - Coping with unreliable media
    - Cross platform compatibility
    - Complex performance issues

To start:
Bits, Bytes and Integers

Binary representations

- Base 2 number representation
  - Represent 15213\textsubscript{10} as 1110110110111\textsubscript{2}
  - Represent 1.20\textsubscript{10} as 1.0011001101100\textsubscript{1111\ldots}
  - Represent 1.5213 \times 10^4 as 1.1101101101101\times 2^{13}
- Electronic implementation
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires

Encoding byte values

- Byte = 8 bits
  - Binary 00000000\textsubscript{2} to 11111111\textsubscript{2}
  - Decimal: 0\textsubscript{10} to 255\textsubscript{10}
    - First digit must not be 0 in C
  - Hexadecimal 00\textsubscript{16} to FF\textsubscript{16}
    - Base 16 number representation
    - Use characters '0' to '9' and 'A' to 'F'
    - Write FA1D37B\textsubscript{16} in C as 0xFA1D37B
      - Or 0xfa1d37b

Byte-oriented memory organization

- Programs refer to virtual addresses
  - Conceptually, a very large array of bytes
  - Secretly, a hierarchy of different memory types
  - System provides address space private to particular "process"
    - Program being executed
    - Program can clobber its own data, but not that of others
- Compiler + runtime system control allocation
  - Where different program objects should be stored
  - All allocation within single virtual address space
Machine words

- Machines have a “word size”
  - Nominal size of integer-valued data
    - Including addresses
  - Many current machines use 32-bit (4 byte) words
    - Limits addresses to 4GB
    - Becoming too small for memory-intensive applications
  - Modern machines often use 64-bit (8 byte) words
    - Potential address space ~ 1.8 X 10^19 bytes
    - x86-64 machines support 48-bit addresses: 256 Terabytes
- Machines support multiple data formats
  - Fractions or multiples of word size
  - Always integral number of bytes

Word-oriented memory

- Addresses specify byte locations
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)

<table>
<thead>
<tr>
<th>32-bit Words</th>
<th>64-bit Words</th>
<th>Bytes</th>
<th>Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addr = 0000</td>
<td>Addr = 0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addr = 0004</td>
<td>Addr = 0008</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addr = 0012</td>
<td>Addr = 0012</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data representations

- Sizes of C objects (in bytes)

<table>
<thead>
<tr>
<th>C data type</th>
<th>Typical 32-bit</th>
<th>ia32</th>
<th>Intel x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>long long</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>8</td>
<td>10/12</td>
<td>10/16</td>
</tr>
<tr>
<td>char *</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

Byte ordering

- How should bytes within multi-byte word be ordered in memory?
  - Big Endian: Sun, PPC, Internet
    - Least significant byte has highest address
  - Little Endian: x86
    - Least significant byte has lowest address

- Origin: “Gullivers Reisen” (Gulliver’s Travels)
- Which end to crack a soft-boiled egg?

Byte ordering example

- Big Endian
  - Least significant byte has highest address
- Little Endian
  - Least significant byte has lowest address
- Example
  - Variable x has 4-byte representation 0x01234567
  - Address given by &x is 0x100

**Big Endian**

<table>
<thead>
<tr>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
</table>

**Little Endian**

<table>
<thead>
<tr>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
</table>

Byte ordering example

- Big Endian
  - Least significant byte has highest address
- Little Endian
  - Least significant byte has lowest address
- Example
  - Variable x has 4-byte representation 0x01234567
  - Address given by &x is 0x100

<table>
<thead>
<tr>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
</table>

**Big Endian**

| 01 | 23 | 45 | 67 |

**Little Endian**

<table>
<thead>
<tr>
<th>0x100</th>
<th>0x101</th>
<th>0x102</th>
<th>0x103</th>
</tr>
</thead>
</table>
Byte ordering example

- **Big Endian**
  - Least significant byte has highest address
- **Little Endian**
  - Least significant byte has lowest address
- **Example**
  - Variable `x` has 4-byte representation `0x01234567`
  - Address given by `&x` is `0x100`

![Big Endian diagram](image)

Little Endian diagram

Example

Variable `x` has 4-byte representation `0x01234567`
Address given by `&x` is `0x100`

Big Endian diagram

Reading byte-reversed listings

- **Disassembly**
  - Text representation of binary machine code
  - Generated by program that reads the machine code
- **Example fragment**:

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Code</th>
<th>Assembly Rendition</th>
</tr>
</thead>
<tbody>
<tr>
<td>8048365:</td>
<td>5b</td>
<td>pop %ebx</td>
</tr>
<tr>
<td>8048366:</td>
<td>81 c3 ab 12 00 00</td>
<td>add $0x12ab,%ebx</td>
</tr>
<tr>
<td>804836c:</td>
<td>83 bb 28 00 00 00 00</td>
<td>cmp $0x0,0x28(%ebx)</td>
</tr>
</tbody>
</table>

- **Deciphering numbers**:
  - Value: `0x12ab`
  - Pad to 4 bytes: `0x000012ab`
  - Split into bytes: `00 00 12 ab`
  - Reverse (endian): `ab 12 00 00`

Examining data representations

- **Code to print byte representation of data**
  - Casting pointer to `unsigned char *` creates byte array

```c
typedef unsigned char *pointer;

void show_bytes(pointer start, int len)
{
    int i;
    for (i = 0; i < len; i++) {
        printf("0x%p	0x%.2x\n", start+i, start[i]);
    }
}
```

**printf directives:**
- `%p` print pointer
- `%x` print hexadecimal

Representing integers

```
int A = 15213;
int B = -15213;
long int C = 15213;
```

```
Decimal:  15213
Binary:   0011 1011 0110 1101
Hex:      3 B 6 D
```

```
A on ia32, x86-64 | A on SPARC
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6D 6D 00 00 00 00</td>
<td>00 00 00 6D 6D</td>
</tr>
<tr>
<td>3B 3B 3B 3B 3B 3B</td>
<td>00 00 00 00 00</td>
</tr>
<tr>
<td>00 00 00 00 00 00</td>
<td>00 00 00 00 00</td>
</tr>
<tr>
<td>93 93 FF FF FF FF</td>
<td>00 00 00 00 00</td>
</tr>
</tbody>
</table>

C on ia32 | C on x86-64 | C on SPARC
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6D 6D 00 00 00 00</td>
<td>00 00 00 6D 6D</td>
<td>00 00 00 00 00</td>
</tr>
<tr>
<td>3B 3B 3B 3B 3B 3B</td>
<td>00 00 00 00 00</td>
<td>00 00 00 00 00</td>
</tr>
<tr>
<td>00 00 00 00 00 00</td>
<td>00 00 00 00 00</td>
<td>00 00 00 00 00</td>
</tr>
<tr>
<td>FF FF FF FF FF FF</td>
<td>00 00 00 00 00</td>
<td>00 00 00 00 00</td>
</tr>
</tbody>
</table>
```

**Two's complement representation**
(covered later)

Representing pointers

```
int B = -15213;
int *P = &B;
```

**SPARC P** | **IA32 P** | **x86-64 P**
---|---|---
EF | D4 | OC
FF | F8 | 89
FB | FF | EC
2C | FF | FF
BF | 7F | FF
```

Different compilers & machines assign different locations to objects
Representing strings

- Strings in C
  - Represented by array of characters
  - Each character encoded in ASCII format
    - Standard 7-bit encoding of character set
    - Character "0" has code 0x30
      - Digit i has code 0x30+i
  - String should be null-terminated
    - Final character = 0
- Compatibility
  - Byte ordering not an issue

Boolean Algebra

- Developed by George Boole in 19th Century
  - Algebraic representation of logic
    - Encode "True" as 1 and "False" as 0
  - AND: A&B = 1 when A=1 and B=1
  - (Inclusive) OR: A|B = 1 when A=1 or B=1
  - NOT: ~A = 1 when A=0
  - Exclusive-OR (XOR): A^B = 1 when A=1 or B=1 but not both

Application of boolean algebra

- Applied to digital systems by Claude Shannon
  - 1937 MIT Master’s Thesis
  - Reason about networks of relay switches
    - Encode closed switch as 1, open switch as 0

General boolean algebras

- Operate on bit vectors
  - Operations applied bitwise

Representing & manipulating sets

- Width w bit vector represents subsets of {0, ..., w-1}
- \( a_j = 1 \) if \( j \in A \):
  - 01101001 \{ 0, 3, 5, 6 \}
  - 01010101 \{ 0, 2, 4, 6 \}
- Operations:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Result</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Intersection</td>
<td>01000001</td>
<td>{ 0, 6 }</td>
</tr>
<tr>
<td></td>
<td>Union</td>
<td>01111101</td>
<td>{ 2, 3, 4, 5, 6 }</td>
</tr>
<tr>
<td>^</td>
<td>Symmetric difference</td>
<td>00111100</td>
<td>{ 2, 3, 4, 5 }</td>
</tr>
<tr>
<td>~</td>
<td>Complement</td>
<td>10101010</td>
<td>{ 1, 3, 5, 7 }</td>
</tr>
</tbody>
</table>

Bit-level operations in C

- Operations &, |, ^, ~ available in C
  - Apply to any "integral" data type
    - long, int, short, char, unsigned
  - View arguments as bit vectors
  - Arguments applied bit-wise

Examples (char data type):
- 0x41 \rightarrow 0xBE
- ~0x00 \rightarrow 0xFF
- 0x69 \& 0x55 \rightarrow 0x41
- 0x69 | 0x55 \rightarrow 0x7D
- 0x69 ^ 0x55 \rightarrow 0x7D
- 0x110101 \& 0x10101 \rightarrow 0x000001
- 0x110101 | 0x10101 \rightarrow 0x111110
- 0x110101 ^ 0x10101 \rightarrow 0x111110

Contrast: Logic operations in C

- &&, ||, !
  - View 0 as "False"
  - Anything nonzero as "True"
  - Always return 0 or 1
  - Early termination

- Examples (char data type)
  - !0x41 → 0x00
  - !0x00 → 0x01
  - ! !0x41 → 0x01
  - 0x69 && 0x55 → 0x01
  - 0x69 || 0x55 → 0x01
  - p && *p (avoids null pointer access)

Shift operations

- Left shift: x << y
  - Shift bit-vector x left y positions
  - Throw away extra bits on left
    - Fill with 0's on right
- Right shift: x >> y
  - Shift bit-vector x right y positions
  - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on right
- Undefined behavior
  - Shift amount < 0 or ≥ word size

Java writes this ">>>".

Integer C puzzles

- Assume 32-bit word size, two's complement integers
- For each of the following C expressions, either:
  - Argue that is true for all argument values
  - Give example where not true

Initialization

```c
int x = foo();
int y = bar();
unsigned ux = x;
unsigned uy = y;
```

Next time: Integers

- Representation: unsigned and signed
- Conversion, casting
- Expanding, truncating
- Addition, negation, multiplication, shifting