Lecture 5: More on Memory
University of Oregon Cybersecurity Club

Meetings every Thursday at 4:30 PM in PSC Visualization Lab

For more information you can contact: uocyberinfo@gmail.com
Instagram: @uocyb
Facebook: @UO Cybersecurity Club
This week’s lab...

• If you “do memory wrong,” then your program will crash
• How to fix?
• Special program: gdb (GNU debugger)
  – gdb can give you lots of hints about why your program crashed!
Lecture 7 (Oct 22) will be online

- 4 videos on memory topics
- So do not come to class on Tuesday October 22
- There will be (should be) Hank OH on Oct 22

- There IS class on Thursday October 17 & Thursday October 24
  – (don’t get confused)
What should you be doing?

• Due today: 2B
• 2C may be posted tonight if we make it far enough in lecture
• Reading the textbook, if you have not done it already
  – Chapter 2.1, 2.2, 2.3, 2.4., 2.5, and 2.6.1
  – Chapter 4.1 (but not 4.1.2), 4.2-4.4
  – \(\rightarrow\) will be an upcoming quiz on details in the book (things I have not lectured on will be fair game)
Review / 2B
Requesting automatic memory

- This is done “automatically”...

```c
#include <stdio.h>

int main()
{
    int A;
    A = 4;
    printf("Value of A is %d\n", A);
}
```
#include <stdio.h>

int main()
{
    int A[21];
    int i;
    for (i = 0 ; i < 21 ; i++)
        A[i] = 3;
    printf("Value of A[10] is %d\n", A[10]);
}

This is just 84 (4*21) bytes. We interpret these 84 bytes as 21 integers, following known conventions.
CIS 212: Project #2B
Assigned: October 10, 2019
Due: October 16, 2019
(which means submitted by 6am on October 17, 2019)
Worth 4% of your grade

Assignment:
1) Write a C program that sorts 100 numbers in an array. The name of the C program should be “project2B.c”
2) You can sort however you want.
   a. https://en.wikipedia.org/wiki/Bubble_sort if you need ideas. Also see the Python code below.
   b. You should not use any subroutines from the C library. (Don’t use qsort, for example)
3) Your program should have the exact same output as mine.
   a. 10 numbers per row, 10 rows
   b. Note I used “tab” to do whitespaces. That makes it pretty. You will need to use tabs too.
   c. You can also see the correct output as “proj2B_correct_output”. Make sure to download this by right clicking and “save link as.” Otherwise, tabs can get converted to spaces.
4) You can confirm this:
   a. Download “proj2B_checker”.
   b. Do a chmod: “chmod 755 proj2B_checker”
   c. Run your program as “./proj2B_checker”

This project will be graded by:
1) Running proj2B_checker on your code and confirming it is correct
2) Inspection of your code
Correct output

<p>| | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>40</td>
<td>45</td>
<td>66</td>
<td>70</td>
<td>77</td>
<td>95</td>
<td>133</td>
<td>134</td>
<td>145</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>158</td>
<td>159</td>
<td>166</td>
<td>166</td>
<td>181</td>
<td>190</td>
<td>202</td>
<td>214</td>
<td>216</td>
<td>230</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>237</td>
<td>243</td>
<td>252</td>
<td>267</td>
<td>268</td>
<td>285</td>
<td>314</td>
<td>325</td>
<td>325</td>
<td>369</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>385</td>
<td>388</td>
<td>391</td>
<td>395</td>
<td>402</td>
<td>414</td>
<td>428</td>
<td>434</td>
<td>437</td>
<td>443</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>446</td>
<td>466</td>
<td>474</td>
<td>477</td>
<td>508</td>
<td>517</td>
<td>552</td>
<td>577</td>
<td>577</td>
<td>599</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>600</td>
<td>603</td>
<td>625</td>
<td>634</td>
<td>635</td>
<td>637</td>
<td>647</td>
<td>649</td>
<td>657</td>
<td>673</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>673</td>
<td>679</td>
<td>682</td>
<td>682</td>
<td>687</td>
<td>697</td>
<td>705</td>
<td>712</td>
<td>717</td>
<td>729</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>737</td>
<td>752</td>
<td>755</td>
<td>760</td>
<td>765</td>
<td>765</td>
<td>780</td>
<td>794</td>
<td>801</td>
<td>804</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>805</td>
<td>821</td>
<td>855</td>
<td>858</td>
<td>876</td>
<td>899</td>
<td>902</td>
<td>904</td>
<td>916</td>
<td>920</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>927</td>
<td>941</td>
<td>949</td>
<td>950</td>
<td>953</td>
<td>954</td>
<td>963</td>
<td>964</td>
<td>968</td>
<td>987</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Used tab ("\t") to make the columns so organized
Starter code

#include <stdio.h>

int main()
{
};
    /* YOUR CODE GOES HERE */

    /* SORT IT */

    /* PRINT IT */

    /* HINT: WRITE THE PRINT FUNCTION FIRST */
}
Important Memory Concepts in C: Automatic Vs Dynamic

• You can allocate variables that only live for the invocation of your function
  – Called automatic/stack variables (will talk more about this later)

• You can allocate variables that live for the whole program (or until you delete them)
  – Called dynamic/heap variables (will talk more about this later as well)
#include <stdio.h>

void function1()
{
    int X = 4;
    printf("X is %d\n", X);
    /* X goes out of scope and the end of this function and thus
     * *automatically* disappears and is no longer accessible */
}

int main()
{
    int Z = 6;
    function1();

    /* NEW SCOPE */
    {
        int Y = 5;
    }
    /* Y no longer exists. Automatically allocated and de-allocated */

    /* only valid variable here is Z ... not X, not Y. */
    printf("Z is %d\n", Z);

    /* Now Z goes away. */
}
Dynamic memory works differently

• You allocate it, it stays around until you de-allocate it or the program ends

• Important: you need a way to keep track of memory
  — If not, the memory will be “leaked.”

• So we need a way of managing dynamic memory.

• The concept for doing this in C is **POINTERS**
Important Memory Concepts in C: Hexadecimal address

• Memory addresses are in hexadecimal
  – Hexadecimal: 16 options for each digit
    – 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
• House address: 123 A Street
• Memory address: 0x7FFFFFFF3AB
Pointers

• Pointer: points to memory location
  – Denoted with ‘*’
  – Example: “int *p”
    • pointer to an integer
  – You need pointers to get to heap memory

• Address of: gets the address of memory
  – Operator: ‘&’
  – Example:
    ```c
    int x;
    int *y = &x;  // this example is pointing to an automatic variable, not a dynamic variable
    ```
New Material
New operator: &

- & == address of
- This will tell you the address of a variable.

```
#include <stdio.h>

int main()
{
    int A;
    A=4;
    printf("Value of A is %d, address of A is %p\n", A, &A);
    return 0;
}
```

```
Hanks-iMac:Downloads hank$ gcc t.c
Hanks-iMac:Downloads hank$ ./a.out
Value of A is 4, address of A is 0x7fff56486bc8
```
Let’s pause and do a basic example with * and &
What is an array?

• Block of contiguous memory
• If elements each have size N bytes and there are M elements, then N*M contiguous bytes
• Let A be address of the beginning of the array
• Then A[0] is at “A”
• And A[1] is at “A+N”
• and so on...
Let’s pause and do some more examples with arrays (and film it too)
C02LN00GFD58:212 hank$ cat A.c
#include <stdio.h>

int main()
{
    int A[5];
    int i;

    printf("A is at \%p\n", A); /* Note I can pass in A as a pointer */
    for (i = 0 ; i < 5 ; i++)
    {
        printf("A[%d] is at \%p\n", i, &(A[i]));
    }
}

C02LN00GFD58:212 hank$ gcc A.c
C02LN00GFD58:212 hank$ ./a.out
A is at 0x7fff5e8b5bd0
A[0] is at 0x7fff5e8b5bd0
A[1] is at 0x7fff5e8b5bd4
A[2] is at 0x7fff5e8b5bd8
A[3] is at 0x7fff5e8b5bdc
A[4] is at 0x7fff5e8b5be0
Dynamic Memory Allocation

• Special built-in function to allocate memory from heap: `malloc`
  – Interacts with Operating System
  – Argument for malloc is how many bytes you want

• Also built-in function to deallocate memory: `free`
free/malloc example

```c
#include <stdlib.h>

int main()
{
    /* allocates memory */
    int *ptr = malloc(2*sizeof(int));

    /* deallocates memory */
    free(ptr);
}
```

Enables compiler to see functions that aren’t in this file. More on this next week.

sizeof is a built in function in C. It returns the number of bytes for a type (4 bytes for int).

don’t have to say how many bytes to free ... the OS knows
Automatic vs Dynamic

• Automatic memory lives only for its current scope
• Dynamic memory lives until you free it, or until the program ends
This is just fine...

Hanks-iMac:Downloads hank$ cat scope.c
#include <stdio.h>
#include <stdlib.h>

int *foo1()
{
    int *X = malloc(sizeof(int)*2);
    X[0] = 1;
    X[1] = 2;
    return X;
}

int main()
{
    int *Y = foo1();
    printf("Y[0] is %d and Y[1] is %d\n", Y[0], Y[1]);
}

Hanks-iMac:Downloads hank$ gcc scope.c
Hanks-iMac:Downloads hank$ ./a.out
Y[0] is 1 and Y[1] is 2
This is not fine...

Hanks-iMac:Downloads hank$ cat bad_scope.c
#include <stdio.h>
#include <stdlib.h>

int *foo1()
{
    int X[2];
    X[0] = 1;
    X[1] = 2;
    return X;
}

int main()
{
    int *Y = foo1();
    printf("Y[0] is %d and Y[1] is %d\n", Y[0], Y[1]);
}

Hanks-iMac:Downloads hank$ gcc bad_scope.c
bad_scope.c:9:12: warning: address of stack memory associated with local variable 'X' returned
        [-Wreturn-stack-address]
    return X;

1 warning generated.
Hanks-iMac:Downloads hank$ ./a.out
Y[0] is 1 and Y[1] is 2
And here it goes wrong...
Memory Segments

• Your program is divided into 4 segments:
  – Code segment: where the machine code instructions are
  – Data segment: where the global variables live ... and other things
  – “Stack”: where automatic memory lives
  – “Heap”: where dynamic memory lives
  – Note: stack & heap are data structures and we will learn more about them later.

If you access memory addresses outside your segments, you get a “segmentation fault” ... which causes a crash
And in some cases it doesn’t have to hit the heap. Just take more memory than intended.
Character Strings
There have been various extensions to ASCII ...
now more than 128 characters
Many special characters are handled outside this convention

ASCII Character Set

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NUL</td>
<td>SOH</td>
<td>STX</td>
<td>ETX</td>
<td>EOT</td>
<td>ENQ</td>
</tr>
<tr>
<td>1</td>
<td>ACK</td>
<td>BEL</td>
<td>BS</td>
<td>HT</td>
<td>LF</td>
<td>VT</td>
</tr>
<tr>
<td>2</td>
<td>FF</td>
<td>CR</td>
<td>SO</td>
<td>SI</td>
<td>DLE</td>
<td>DC1</td>
</tr>
<tr>
<td>3</td>
<td>DC2</td>
<td>DC3</td>
<td>DC4</td>
<td>NAK</td>
<td>SYN</td>
<td>ETB</td>
</tr>
<tr>
<td>4</td>
<td>CAN</td>
<td>EM</td>
<td>SUB</td>
<td>ESC</td>
<td>FS</td>
<td>GS</td>
</tr>
<tr>
<td>5</td>
<td>RS</td>
<td>US</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

image source: granneman.com
printf

First, printf is a function. You call printf and it prints stuff; that is probably obvious.

The next thing is the syntax.

printf(STRING_TO_PRINT, arg1, arg2, arg3, ..., argN);

The trick is that STRING_TO_PRINT will contain directions about how to incorporate arg1, arg2, arg3, etc.

For example,
printf("This is 4: \%d\n", 4);

will print "this is 4: 4"

The \n is a new line -- it makes the print statement go to the next line.
A % tells printf to bring in the next argument. It would start with arg1, then arg2, etc.
The character that follows the % the type of the argument. So far, we only know 2 types:
\%d: the argument is an integer
\%p: the argument is a pointer
Some Special Characters

```c
#include <stdio.h>

int main()
{
    char A = 'A';
    char B = 'B';
    char NL = '\n';
    char tab = '\t';
    char backslash = '\\';
    char NULL_character = '\0';
    printf("A = %d, B = %d, NL = %d, tab =%d, backslash = %d, NULL char = %d\n", A, B, NL, tab, backslash, NULL_character);
    printf("A = %c, B = %c, NL = %c, tab =%c, backslash = %c, NULL char = %c\n", A, B, NL, tab, backslash, NULL_character);
}
```

New: `%c` prints character
character strings

• A character “string” is:
  – an array of type “char”
  – that is terminated by the NULL character

• The C library has multiple functions for handling strings
building a character string

C02LN00GFD58:212 hank$ cat hello_world.c
#include <stdio.h>

int main()
{
    char H[12];
    H[5] = ' ';
    H[11] = '\0';
    printf("String is %s\n", H);
}

C02LN00GFD58:212 hank$ gcc hello_world.c
C02LN00GFD58:212 hank$ ./a.out
String is Hello World

New: %s prints “strings” (arrays of chars terminated by \0)
building a character string

```c
#include <stdio.h>

int main()
{
    char H[12] = "hello world";
    printf("H is %s\n", H);
}
```

Compiler automatically adds a \0 for you!

```sh
C02LN00GFD58:212 hank$ cat hello_world2.c
C02LN00GFD58:212 hank$ gcc hello_world2.c
C02LN00GFD58:212 hank$ ./a.out
H is hello world
```
```c
#include <stdio.h>

int main()
{
    char H[12] = "hello world";
    printf("H is %s\n", H);
    H[5] = '\0';
    printf("H is %s\n", H);
}
```

```
C02LN00GFD58:212 hank$ gcc hello_world2.c
C02LN00GFD58:212 hank$ ./a.out
H is hello world
H is hello
```
Did you notice

• Characters in C are single quotes: ‘A’
• Strings in C are double quotes: “hello world”
• A string of a single character is still double quotes: “A”
  – (And this is actually “A\0”)
C02LN00GFD58:212 hank$ gcc hello_world2.c
C02LN00GFD58:212 hank$ ./a.out
H is hello world!

C02LN00GFD58:212 hank$ cat hello_world2.c
#include <stdio.h>

int main()
{
    char H[12] = "hello world";
    H[11] = '!'s
    printf("H is %s\n", H);
}

C02LN00GFD58:212 hank$ gcc hello_world2.c
C02LN00GFD58:212 hank$ ./a.out
H is hello world!d?I?
Character strings example

```c
#include <stdio.h>

int main()
{
    char str[12] = "hello world";
    char *str2 = str+6;

    printf("str is \"%%s\" and str2 is \"%%s\\n", str, str2);

    str[5] = '\0';

    printf("Now str is \"%%s\" and str2 is \"%%s\\n", str, str2);
}
```

```
128-223-223-72-wireless:330 hank$ cat string.c
128-223-223-72-wireless:330 hank$ gcc string.c
128-223-223-72-wireless:330 hank$ ./a.out
str is "hello world" and str2 is "world"
Now str is "hello" and str2 is "world"
```
Useful C library string functions

- `strcpy`: string copy
- `strncpy`: string copy, but just first N characters
- `strlen`: length of a string

```c
#include <string.h>
#include <stdio.h>

int main()
{
    char str[12] = "hello world";
    char str2[6], str3[7];
    strcpy(str2, str+strlen("hello "));
    strncpy(str3, str, strlen("hello "));
    printf("%s,%s\n", str2, str3);
}
```

```
128-223-223-72-wireless:330 hank$ gcc strcpy.c
128-223-223-72-wireless:330 hank$ ./a.out
world,hello
```
Useful C library string functions

• `strcpy`: string copy
• `strncpy`: string copy, but just first N characters
• `strlen`: length of a string

```
#include <string.h>
#include <stdio.h>

int main()
{
    char str[12] = "hello world";
    char str2[7], str3[6];
    strcpy(str2, str + strlen("hello "));
    strncpy(str3, str, strlen("hello "));
    printf("%s,%s\n", str2, str3);
}
```

What happened here?
## More useful C library string functions

### Functions

**Copying:**
- `memcpy`: Copy block of memory ([function](source:cplusplus.com))
- `memmove`: Move block of memory ([function](source:cplusplus.com))
- `strcpy`: Copy string ([function](source:cplusplus.com))
- `strncpy`: Copy characters from string ([function](source:cplusplus.com))

**Concatenation:**
- `strcat`: Concatenate strings ([function](source:cplusplus.com))
- `strncat`: Append characters from string ([function](source:cplusplus.com))

**Comparison:**
- `memcmp`: Compare two blocks of memory ([function](source:cplusplus.com))
- `strcmp`: Compare two strings ([function](source:cplusplus.com))
- `strcoll`: Compare two strings using locale ([function](source:cplusplus.com))
- `strncmp`: Compare characters of two strings ([function](source:cplusplus.com))
- `strxfrm`: Transform string using locale ([function](source:cplusplus.com))

**Searching:**
- `memchr`: Locate character in block of memory ([function](source:cplusplus.com))
- `strchr`: Locate first occurrence of character in string ([function](source:cplusplus.com))
- `strcspn`: Get span until character in string ([function](source:cplusplus.com))
- `strpbrk`: Locate characters in string ([function](source:cplusplus.com))
- `strrchr`: Locate last occurrence of character in string ([function](source:cplusplus.com))
- `strspn`: Get span of character set in string ([function](source:cplusplus.com))
- `strstr`: Locate substring ([function](source:cplusplus.com))
- `strtok`: Split string into tokens ([function](source:cplusplus.com))

**Other:**
- `memset`: Fill block of memory ([function](source:cplusplus.com))
- `strerror`: Get pointer to error message string ([function](source:cplusplus.com))
- `strlen`: Get string length ([function](source:cplusplus.com))

### Macros

- `NULL`: Null pointer ([macro](source:cplusplus.com))

### Types

- `size_t`: Unsigned integral type ([type](source:cplusplus.com))

[source: cplusplus.com]
argc & argv

- two arguments to every C program
- argc: how many command line arguments
- argv: an array containing each of the arguments
- ".a.out hank childs"
- \( \text{argc} = 3 \)
- \( \text{argv}[0] = \text{"a.out"}, \text{argv}[1] = \text{"hank"}, \text{argv}[2] = \text{"childs"} \)
```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i;
    for (i = 0; i < argc; i++) {
        printf("Argument %d is %s\n", i, argv[i]);
    }
}
```
Bonus Slides
Important note on debugging: buffering and printf

• Important: printf is buffered
• So:
  – printf puts string in buffer
  – other things happen
  – buffer is eventually printed
• But what about a crash?
  – printf puts string in buffer
  – other things happen ... including a crash
  – buffer is never printed!

One solution: fflush(stdout)
Let’s pause and do a video on fflush
mv: Unix command for renaming a file

```bash
mkdir tmp
cd tmp
ls

# Create file 'a'
touch a
ls

# Rename file 'a' to 'b'
mv a b
ls
```
How a Computer is Not Like Monopoly

- **Actions during cycle/turn:**
  - Monopoly: roll, buy, build, trade
  - Computer: other

- **Who’s turn?**
  - Monopoly: passes between players
  - Computer: always the computer’s turn

- **Time spent per cycle/turn:**
  - Monopoly: variable
  - Computer: fixed

- **Duration of turn:**
  - Monopoly: sometimes over a minute
  - Computer: much faster!!!
“First” computer: ENIAC

- Year: 1946
- Location: Pennsylvania
- Purpose: military
- Cost: $487K
  - ($6.9M today)
- Technology:
  - very different than today
  - ... but still the same
Vacuum Tubes

• Vacuum tubes:
  – Glass tubes with no gas
  – Used to control electron flow in early computers

• Occasionally, a bug would get stuck in the tube and cause the program to malfunction

• We no longer have vacuum tubes, but the term bug has remained with us...
An ENIAC Computation

• Used for military calculations:
  – A-bomb design
  – Missile delivery

• ENIAC could do ~5000 calculations in one minute

• In one case:
  – ENIAC did a calculation in 30 seconds
  – Human being took 20 hours
  – 2400x increase in speed

source: wikipedia
Hertz (Hz) = unit of measurement for how fast you do something

- 1 Hertz = do something once per second
- KHz = 1024 Hz
- MHz = 1024 KHz
- GHz = 1024 MHz

- The ENIAC machine ran at 5000 Hertz, or about 5 KHz.
  - Vocab term: “clock speed” → the number of cycles per second
    - the clock speed of the ENIAC was 5 KHz
Today’s Desktop Computers Are Fast!

- Most computers run at ~1-3 GHz
- i.e., operates billions of instructions each second

- This is about one million times faster than the ENIAC
  - ... and the ENIAC was 2400X faster than humans
  - (at least at tasks computers are good at)
What does a million-fold increase mean?

Distance: a 2” map of Oregon is 1:1,000,000 scale

Time: 1 second to 277 hours is 1:1,000,000 scale

Time: 1 minute to 694 days is 1:1,000,000 scale

Time: 1 hour to 114 years is 1:1,000,000 scale

Time: 1 day to 2738 years is 1:1,000,000 scale
1 million-fold increase!
How does this happen?

• Moore’s Law (old timer’s version)
  – Clock speed doubles every 18 months

• Moore’s Law (newer version but still for old timers)
  – Clock speed doubles every 24 months
Moore’s Law

• Moore’s Law (actual version)
  – Number of transistors doubles every 24 months
  – And clock speed is a reflection of number of transistors

• So what is a transistor?
  – Semiconductor device for amplifying or switching electronic signals/power
  – Fundamental building block of modern electronics
  – Replacement for vacuum tube
Moore's Law – The number of transistors on integrated circuit chips (1971-2016)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.

The data visualization is available at OurWorldInData.org. There you find more visualizations and research on this topic. Licensed under CC-BY-SA by the author Max Roser.
But actually…

Source: maximumpc.com
The reason is power

- Desktop computer takes ~200W
  - There are multiple components that consume the power:
    - CPU
    - Monitor
    - Disk
    - Memory
- $200W \times 1 \text{ year} \rightarrow \sim$70
Relationship Between Power and Clock Speed

• Clock goes twice as fast $\rightarrow$ Power goes up by factor of 8
  – (Increase of X in clock speed $\rightarrow$ Increase of $X^3$ in power)

• Clock speeds haven’t changed in 12 years

• What if they had doubled every 2 years?

• Then 64X faster
  – $\rightarrow$ 262144X more power (for the CPU)
  – $\rightarrow$ power bill now $\$18M$
New vocab term: "core" lightweight version of a CPU

What Changed?
• We are getting double the transistors every two years
• ... but clock speed is the same
• ... so what is changing?

<table>
<thead>
<tr>
<th>Core</th>
<th>GHZ</th>
<th>Memory</th>
<th>Fabric</th>
<th>DDR4</th>
<th>Power</th>
<th>Pricing</th>
</tr>
</thead>
<tbody>
<tr>
<td>72901</td>
<td>72</td>
<td>16GB</td>
<td>Yes</td>
<td>384GB</td>
<td>245W</td>
<td>$6254</td>
</tr>
<tr>
<td></td>
<td>1.5</td>
<td>7.2 GT/s</td>
<td></td>
<td>2400 MHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Best Performance/Node</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Core</th>
<th>GHZ</th>
<th>Memory</th>
<th>Fabric</th>
<th>DDR4</th>
<th>Power</th>
<th>Pricing</th>
</tr>
</thead>
<tbody>
<tr>
<td>7250</td>
<td>68</td>
<td>16GB</td>
<td>Yes</td>
<td>384GB</td>
<td>215W</td>
<td>$4876</td>
</tr>
<tr>
<td></td>
<td>1.4</td>
<td>7.2 GT/s</td>
<td></td>
<td>2400 MHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Best Performance/Watt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Core</th>
<th>GHZ</th>
<th>Memory</th>
<th>Fabric</th>
<th>DDR4</th>
<th>Power</th>
<th>Pricing</th>
</tr>
</thead>
<tbody>
<tr>
<td>7230</td>
<td>64</td>
<td>16GB</td>
<td>Yes</td>
<td>384GB</td>
<td>215W</td>
<td>$3710</td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td>7.2 GT/s</td>
<td></td>
<td>2400 MHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Best Memory Bandwidth/Core</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Core</th>
<th>GHZ</th>
<th>Memory</th>
<th>Fabric</th>
<th>DDR4</th>
<th>Power</th>
<th>Pricing</th>
</tr>
</thead>
<tbody>
<tr>
<td>7210</td>
<td>64</td>
<td>16GB</td>
<td>Yes</td>
<td>384GB</td>
<td>215W</td>
<td>$2438</td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td>6.4 GT/s</td>
<td></td>
<td>2133 MHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Best Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1 Available beginning in September  
2 Plus 15W for integrated fabric  
3 Pricing shown is for parts without integrated fabric. Add additional $278 for integrated fabric versions of these parts. Integrated fabric parts available in October.
How To Use Multiple Cores?

• Answer: parallel programming
  – Write computer programs that use all the cores
  – Ideally the coordination between the cores is minimal
Parallel Programming Concepts

• Usual goal:
  – if program takes $N$ seconds to run with one core
  – then take $N/2$ seconds to run with two cores
  – and $N/M$ seconds to run with $M$ cores

Let’s consider an example outside of computers
Example: paint a house

• One person: 6 days (1 day = 10 hours)
• Two people: 3 days
• Three people: 2 days
• Six people: 1 day

• Sixty people: 1 hour?
• Six hundred people: 6 minutes?
Example: paint a house, plan #2

- One person: paint one house in 6 days
- Two people: paint two houses in 6 days
- Three people: paint three houses in 6 days
- One thousand people: paint 1000 houses in 6 days?

Parallel programming is hard, and smart people spend their whole careers figuring out how to make parallel programs be efficient.
GPUs: Graphical Processing Units (graphics cards)

**Historical:**
- Introduced to accelerate graphics (gaming)!
- Boom with desktop PCs in late 90s onward
- Mid-2000’s: people start hacking the interface to program a GPU to make it do things besides graphics
- Late 2000’s: GPU makers jump on board and start encouraging folks to program GPUs directly

**GPGPU:** General-purpose GPU programming
- Mid-2010’s: GPUs used for *lots* of computing problems.
- Machine learning workhorse!

**Emergent Tech**

**Bitcoin heist with a twist: This time it's servers that were stolen**

Icelandic cops cuff 11 on suspicion of data centre robberies

By Simon Sharwood, APAC Editor 5 Mar 2018 at 04:57

Icelandic police have cuffed 11 people in connection with four raids on data centres that targeted cryptocurrency mining equipment.

The raids started in December 2017 when three data centres were cracked in December. Another raid took place in January. 600 servers went missing in the heists.

Icelandic police kept the raids secret while they pursued their investigations. Those efforts culminated in 11 arrests and an appearance before the Reykjavik District Court last Friday. Two of the 11 were detained and the matter held over for another day.

The 600 servers are all still missing, the Associated Press reports. Which is no surprise: x86 kit is pretty generic. The real prize inside a bitcoin-mining rig is either GPUs, RAM or nicely fast solid-state disks. Those components are all tiny compared to servers and could probably have been posted out of Iceland piecemeal without much hassle.

Iceland has become something of a hub for demanding workloads like cryptocurrency mining because cheap energy and low ambient temperatures make it a low-cost destination to run data centres and the kit inside them. The nation also has a low crime rate. 😎
Why Are GPUs So Good?

- NVIDIA: company that makes GPUs
- NVIDIA Volta: latest type of NVIDIA GPU

Volta facts:
- 5120 cores
- 1200MHz clock speed
- Can do 2000X more operations than my laptop
- Suggested MSRP: $2,999.00

This level of increase in computation is not just a quantitative change, it is a qualitative one too.