Lecture 6: Wrap up character strings
Structs
Hank’s OH

- Monday: great OH!
- Thursday: Viet will cover my OH (in PSC)
NO CLASS TUESDAY

- Instead YouTube lecture
- Posted this weekend
- Will be entirely about how to do Project 2D
- (2C discussed later this lecture)
Automatic vs Dynamic Memory

- In C, there are two types of variables
  - Actually three, but ignoring “static” for now
- Automatic: taken care of for you automatically
- Dynamic: you manage it

- So, of course, everyone wants automatic, right?
What is happening here?

```c
#include <stdio.h>

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

Answer: “automatic” allocation of an integer. This integer has an address in memory. The address is in hexadecimal.
New operator: &

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

```c
#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$ cat t.c
Hanks-iMac:Downloads hank$ gcc t.c
Hanks-iMac:Downloads hank$ ./a.out
Value of A is 4, address of A is 0x7fff56486bc8
```
Arrays

```
#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.
What is an array?

- Block of contiguous memory
- If elements each have size N bytes and there are M elements, them 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...
```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 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**
Pointers

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

• Special built-in function to allocate dynamic memory: `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

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

```c
#include <stdlib.h>
int main()
{
    /* allocates memory */
    int *ptr = malloc(2*sizeof(int));

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

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...

```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...

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

int *foo2()
{
    int A[2];
    A[0] = 3;
    A[1] = 4;
    return A;
}

int main()
{
    int *Y = foo1();
    int *B = foo2();
    printf("Y[0] is %d and Y[1] is %d\n", Y[0], Y[1]);
    printf("B[0] is %d and B[1] is %d\n", B[0], B[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;
    ^

bad_scope.c:17:12: warning: address of stack memory associated with local variable 'A' returned
    [-Wreturn-stack-address]
    return A;
    ^
```

2 warnings generated.
Hanks-iMac:Downloads hank$ ./a.out
```
Y[0] is 3 and Y[1] is 4
B[0] is 0 and B[1] is 0
```
Character Strings
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

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

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

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);
}
```

```bash
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”)
```c
#include <stdio.h>

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

Great Question

• What happens when we set $H[5] = \backslash 0$?
Character strings example

```
128-223-223-72-wireless:330 hank$ cat string.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$ 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

```
128-223-223-72-wireless:330 hank$ cat strcpy.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
gcc: error: strcpy.c:10: error: unknown #directive 'include'
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

```c
#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](https://cplusplus.com)
- **memmove**
  - Move block of memory ([function](https://cplusplus.com)
- **strcpy**
  - Copy string ([function](https://cplusplus.com)
- **strncpy**
  - Copy characters from string ([function](https://cplusplus.com)

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

### Comparison:
- **memcmp**
  - Compare two blocks of memory ([function](https://cplusplus.com)
- **strcmp**
  - Compare two strings ([function](https://cplusplus.com)
- **strcasecmp**
  - Compare two strings using locale ([function](https://cplusplus.com)
- **strncmp**
  - Compare characters of two strings ([function](https://cplusplus.com)
- **strxfrm**
  - Transform string using locale ([function](https://cplusplus.com)

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

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

## Macros
- **NULL**
  - Null pointer ([macro](https://cplusplus.com)

## Types
- **size_t**
  - Unsigned integral type ([type](https://cplusplus.com)

**source:** cplusplus.com
argv & argc

- two arguments to every C program
- argc: how many command line arguments
- argv: an array containing each of the arguments
- "./a.out hank childs"
- $\rightarrow$ argc == 3
```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]);
    }
}
```

```
C02LN00GFD58:212 hank$ gcc argv.c
C02LN00GFD58:212 hank$ ./a.out hank childs loves C and vi
Argument 0 is ./a.out
Argument 1 is hank
Argument 2 is childs
Argument 3 is loves
Argument 4 is C
Argument 5 is and
Argument 6 is vi
```
Project 2C

CIS 212: Project #2C
Assigned: October 11, 2018
Due: October 16, 2018
(which means submitted by 6am on October 17, 2018)
Worth 4% of your grade

Assignment:
1) Write a C program parses three strings.
   a. String #1: an integer. You need to construct this integer by reading
      one character at a time.
   b. String #2: an operation. These will all be single character strings. The
      valid operations are + or -.
   c. String #3: an integer. (Same as String #1)
2) After parsing the 3 strings, you should perform the desired operation (+ or -).
3) Print the output of the operation.

This project will be graded by:
1) Running the Unix script “check_2c”. It is available on the course website.

If the diff program shows any difference, you will get less than half credit.

What should you upload?: Just a single file, which is your C source code.
Enums, structs, typedef, union
Enums

• Enums make your own type
  – Type is “list of key words”

• Enums are useful for code clarity
  – Always possible to do the same thing with integers

• Be careful with enums
  – … you can “contaminate” a bunch of useful words
C keyword “enum” – means enum definition is coming

```c
enum StudentType {
    HighSchool, Freshman, Sophomore, Junior, Senior, GradStudent
};
```

This enum contains 6 different student types

semi-colon!!!
enum example

```c
int AverageAge(enum StudentType st)
{
    if (st == HighSchool)
        return 16;
    if (st == Freshman)
        return 18;
    if (st == Sophomore)
        return 19;
    if (st == Junior)
        return 21;
    if (st == Senior)
        return 23;
    if (st == GradStudent)
        return 26;

    return -1;
}
```
enums translate to integers ... and you can set their range

```c
#include <stdio.h>

define StudentType
{
    HighSchool = 105,
    Freshman,
    Sophomore,
    Junior,
    Senior,
    GradStudent
};

int main()
{
    printf("HighSchool = %d, GradStudent = %d\n", HighSchool, GradStudent);
}
```

```
128-223-223-72-wireless:330 hank$ gcc enum2.c
128-223-223-72-wireless:330 hank$ ./a.out
HighSchool = 105, GradStudent = 110
```
But enums can be easier to maintain than integers

```c
enum StudentType {
    HighSchool,
    Freshman,
    Sophomore,
    Junior,
    Senior,
    PostBacc,
    GradStudent
};
```

```c
int AverageAge(enum StudentType st) {
    if (st == HighSchool)
        return 16;
    if (st == Freshman)
        return 18;
    if (st == Sophomore)
        return 19;
    if (st == Junior)
        return 21;
    if (st == Senior)
        return 23;
    if (st == PostBac)
        return 24;
    if (st == GradStudent)
        return 26;
    return -1;
}
```

If you had used integers, then this is a bigger change and likely to lead to bugs.
Simple Data Types

- float
- double
- int
- char
- unsigned char

All of these are simple data types
Structs: a complex data type

- Structs: mechanism provided by C programming language to define a group of variables
  - Variables must be grouped together in contiguous memory

- Also makes accessing variables easier ... they are all part of the same grouping (the struct)
struct syntax

C keyword “struct” – means struct definition is coming

struct Ray
{
    double origin[3];
    double direction[3];
};

This struct contains 6 doubles, meaning it is 48 bytes

semi-colon!!!

Declaring an instance

int main()
{
    struct Ray r;
    r.origin[0] = 0;
    r.origin[1] = 0;
    r.origin[2] = 0;
    r.direction[0] = 1;
    r.direction[1] = 0;
    r.direction[2] = 0;
}

“.” accesses data members for a struct
Nested structs

```c
struct Origin
{
    double originX;
    double originY;
    double originZ;
};

struct Direction
{
    double directionX;
    double directionY;
    double directionZ;
};

struct Ray
{
    struct Origin ori;
    struct Direction dir;
};
```

```c
int main()
{
    struct Ray r;
    r.ori.originX = 0;
    r.ori.originY = 0;
    r.ori.originZ = 0;
    r.dir.directionX = 0;
    r.dir.directionY = 0;
    r.dir.directionZ = 0;
}
```

- accesses `dir` part of `Ray`
- accesses `directionZ` part of `Direction` (part of `Ray`)
typedef

- typedef: tell compiler you want to define a new type

```c
typedef struct Ray
{
    double origin[3];
    double direction[3];
} Ray;
```

```c
int main()
{
    struct Ray r;
    r.origin[0] = 0;
    r.origin[1] = 0;
    r.origin[2] = 0;
    r.direction[0] = 1;
    r.direction[1] = 0;
    r.direction[2] = 0;
}
```

saves you from having to type “struct” every time you declare a struct.
Other uses for typedef

• Declare a new type for code clarity
  – typedef int MilesPerHour;
    • Makes a new type called MilesPerHour.
    • MilesPerHour works exactly like an int.

• Also used for enums & unions
  – same trick as for structs ... typedef saves you a word
So important: struct data member access is different with pointers

```c
typedef struct
{
    double origin[3];
    double direction[3];
} Ray;

int main()
{
    Ray r;
    r.origin[0] = 0;
    r.origin[1] = 0;
    r.origin[2] = 0;
    r.direction[0] = 1;
    r.direction[1] = 0;
    r.direction[2] = 0;
}
```

```c
typedef struct
{
    double origin[3];
    double direction[3];
} Ray;

int main()
{
    Ray *r = malloc(sizeof(Ray));
    r->origin[0] = 0;
    r->origin[1] = 0;
    r->origin[2] = 0;
    r->direction[0] = 1;
    r->direction[1] = 1;
    r->direction[2] = 1;
}
```

Pointers: use “->”
Instances (i.e., not pointers): use “.”
Unions

• Union: special data type
  – store many different memory types in one memory location

```c
typedef union
{
    float x;
    int y;
    char z[4];
} cis330_union;
```

When dealing with this union, you can treat it as a float, as an int, or as 4 characters.

This data structure has 4 bytes
Unions

Why are unions useful?

```c
#include <stdio.h>

typedef union
{
    float x;
    int y;
    char z[4];
} cis330_union;

int main()
{
    cis330_union u;
    u.x = 3.5; /* u.x is 3.5, u.y and u.z are not meaningful */
    u.y = 3; /* u.y is 3, now u.x and u.z are not meaningful */
    printf("As u.x = \%f, as u.y = \%d\n", u.x, u.y);
}
```

```
128-223-223-72-wireless:330 hank$ gcc union.c
128-223-223-72-wireless:330 hank$ ./a.out
As u.x = 0.000000, as u.y = 3
```
Unions Example

typedef struct
{
    int firstNum;
    char letters[3];
    int endNums[3];
} CA_LICENSE_PLATE;

typedef struct
{
    char letters[3];
    int nums[3];
} OR_LICENSE_PLATE;

typedef struct
{
    int nums[6];
} WY_LICENSE_PLATE;

typedef union
{
    CA_LICENSE_PLATE ca;
    OR_LICENSE_PLATE or;
    WY_LICENSE_PLATE wy;
} LicensePlate;
typedef struct
{
    int firstNum;
    char letters[3];
    int endNums[3];
} CA_LICENSE_PLATE;

typedef struct
{
    char letters[3];
    int nums[3];
} OR_LICENSE_PLATE;

typedef struct
{
    char letters[3];
    int nums[3];
} WY_LICENSE_PLATE;

typedef union
{
    CA_LICENSE_PLATE ca;
    OR_LICENSE_PLATE or;
    WY_LICENSE_PLATE wy;
} LicensePlate;

typedef enum
{
    CA,
    OR,
    WY
} US_State;

typedef struct
{
    char *carMake;
    char *carModel;
    US_State state;
    LicensePlate lp;
} CarInfo;

int main()
{
    CarInfo c;
    c.carMake = "Chevrolet";
    c.carModel = "Camaro";
    c.state = OR;
    c.lp.or.letters[0] = 'X';
    c.lp.or.letters[1] = 'S';
    c.lp.or.letters[2] = 'Z';
    c.lp.or.nums[0] = 0;
    c.lp.or.nums[1] = 7;
    c.lp.or.nums[2] = 5;
}
Bonus Slides
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
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 * 1 year → ~$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?

**CHOOSE YOUR OPTIMIZATION POINT**

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

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

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

<table>
<thead>
<tr>
<th>7210</th>
<th>64</th>
<th>1.3</th>
<th>16GB 6.4 GT/s</th>
<th>Yes</th>
<th>384GB 2133 MHz</th>
<th>215W</th>
<th>$2438</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best Value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

- Intel Xeon Silver
- Intel Xeon Gold
- Intel Xeon Platinum

M - an optional SKU is available with support for up to 1.5TB memory per CPU socket
F - an optional SKU is available with integrated 100Gbps Intel Omni-Path fabric
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 #3

- 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 1990s onward
  - Mid-2000’s: people start "hacking" 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 Reykjanes 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?

Market summary > NVIDIA Corporation
NASDAQ: NVDA - Mar 5, 7:59 PM EST

235.65 USD ↓ 0.89 (0.38%)
After-hours: 236.50 ↑ 0.36%

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