Why C?

- See [http://www.tiobe.com/index.htm](http://www.tiobe.com/index.htm)

<table>
<thead>
<tr>
<th>02/07 rank</th>
<th>02/06 rank</th>
<th>movement</th>
<th>Language</th>
<th>share</th>
<th>Δ in last year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>=</td>
<td>Java</td>
<td>18.978%</td>
<td>-3.45%</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>=</td>
<td>C</td>
<td>16.104%</td>
<td>-2.23%</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>=</td>
<td>C++</td>
<td>10.768%</td>
<td>-0.53%</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>↑</td>
<td>PHP</td>
<td>8.847%</td>
<td>-0.07%</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>↓</td>
<td>(Visual) Basic</td>
<td>8.369%</td>
<td>-1.03%</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>=</td>
<td>Perl</td>
<td>6.073%</td>
<td>-0.63%</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>↑</td>
<td>Python</td>
<td>3.566%</td>
<td>+0.90%</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>↓</td>
<td>C#</td>
<td>3.189%</td>
<td>-0.78%</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>↑</td>
<td>JavaScript</td>
<td>2.982%</td>
<td>+1.47%</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>↑</td>
<td>Ruby</td>
<td>2.528%</td>
<td>+2.12%</td>
</tr>
</tbody>
</table>

What we will and won’t do

- We will learn:
  - to write simple programs
  - basic interface
  - control flow, math, printing
  - data types
  - enough to be dangerous
- We won’t learn:
  - advanced pointer operations
  - large projects (linking separate programs)
  - distinctions between public, private, external variables
  - enough to be really dangerous

C How it Stacks Up

- As U can C, the C language (and its extensions/derivatives) dominates the software community
  - Java also a strong showing
  - Python worth a peek
- Advantages of C:
  - compiled code runs FAST
  - allows low-level device control
  - a foundation of the programming world
- Disadvantages of C:
  - strings are a pain in the @$$
  - awkward conventions (pointers can be difficult to learn)
  - requires a compiler
C File Types

- **Source Code**
  - The stuff you type in; has .c extension
- **Compiled “Executable”**
  - The ready-to-run product: usually no extension in Unix, .exe
    in DOS
- **Header Files**
  - Contain definitions of useful functions, constants: .h
    extension
- **Object Files**
  - A pre-linked compiled tidbit: .o in Unix, .obj in DOS
  - Only if you’re building in pieces and linking later

A typical (short) program

```c
#include <stdio.h>

int main(void)
{
    int i=53;
    printf("The illustrious variable, i, is %d\n",i);
    return 0;
}
```

- **Notes:**
  - First include is so we have access to printf (standard i/O)
  - Define the main program (must be called main) to take no
    arguments (thus void) and return an integer
  - Braces surround the program
  - Print value of integer, i, in formatted line
  - Return zero (common return value for successful program)

More on program

```c
#include <stdio.h>

int main(void)
{
    int i; i=53;
    printf("\nThe illustrious variable, i, is %d\n"),i);
    return 0;
}
```

- Semicolons delimit separate statements, but this program,
  while compact, is harder on the eyes
- This time, we defined and assigned the variable in separate
  steps (more commonly done)
- We shortened the print statement fluff
- The format is now 4 characters wide, forcing leading zeros
  - Output will be: i = 0053
- Could compactify even more, if sadistic

Alternate form

```c
#include <stdio.h>

int main(void)
{
    int i; i=53;
    printf("\nThe illustrious variable, i, is %04d\n"),i);
    return 0;
}
```

- Semicolons end each line within program
- Spacing is not required, but makes for easier reading
- All variables must be declared before they are used
- Could have simply said: int i; then declared later that
  i=53;
- The \n is a newline; the %d formats as decimal integer
#include <stdio.h>

int main(void)
{
    char c;                       // single byte
    int i;                        // typical integer
    long j;                       // long integer
    float x;                      // floating point (single precision)
    double y;                     // double precision

    c = 'A';
    i = 356;
    j = 230948935;
    x = 3.14159265358979;
    y = 3.14159265358979;

    printf("c = %d = 0x%02x, i = %d, j = %ld, x = %f, y = %f\n",c,c,i,j,x,y);

    c = i;
    i = 9259852835;

    printf("c = %d, i = %d, x = %.14f, y = %.14lf\n",c,i,x,y);

    return 0;
}

---

Variable types

- `char`: 1 byte
- `int`: typical integer
- `long`: long integer
- `float`: single precision floating point
- `double`: double precision floating point

---

Output of previous program

- Output looks like:
  - `c` = 65 = 0x41, `i` = 356, `j` = 230948935, `x` = 3.141593, `y` = 3.141593
  - c = 100, i = 669918243, x = 3.14159274101257, y = 3.14159265358979

- Notes:
  - `c` "wrapped" around 256 when assigned to be 356
  - i couldn't handle the large value, and also wrapped
    - int is actually the same as long on this machine
  - The `float` can't handle the full precision set out
  - broke printf line: spacing irrelevant: semicolons do the work
  - The d, x, ld, f, and if format codes correspond to decimal, hex, long decimal, float, and long float, respectively

---

Feeding data to the program

- Command line arguments allow the same program to be run repeatedly with different inputs (very handy)
- How to do it:
  - `main()` now takes arguments: traditionally `argc` and `argv[]`
  - `argc` is the number of command line arguments
    - minimum is one: the command itself
  - `argv[]` is an array of strings (words)
    - one for each of the space-separated blocks of text following the command on the command line
  - C arrays are numbered starting at zero
  - The command line entry: `one-ray -10.0 1.0 0.0` has:
    - `argc` = 4
    - `argv[0] = one-ray, argv[1] = -10.0, etc.`

---

```c
#include <stdio.h> // for printf(), sscanf()
#include <stdlib.h> // for exit()

int main(int argc, char* argv[]) {
    int int_val;
    double dbl_val;

    if (argc > 2) {
        scanf(argv[1],"%f",&dbl_val);
        scanf(argv[2],"%d",&int_val);
    }
    else {
        printf("usage: %s double_val int_val\n",argv[0]);
        exit(-1);
    }
    printf("Got double_val = %f; int_val = %d\n",double_val,int_val);
    return 0;
}
```
### C-Programming, Part 1

#### Lecture 12

---

**Result**

- If I run simply `prog_name`, without arguments, I get:
  - Usage: `prog_name double_val int_val`
  - normally, these would be given more descriptive names, like `initial_x_position` and `number_of_trials`

- If I run `prog_name 3.14 8`, I get:
  - Got `double_val = 3.140000; int_val = 8`

**Note that:**
- we needed a new header file for `exit()`
- we are using `ascanf()` to scan a value into a variable
- the `#` symbol before the variable name points to that variable’s memory address so `ascanf` knows where to put the value
- `printf` (and `asprintf`, `fprintf`, etc.) is forgiving about `if` vs `flf`, etc., but not so with scan functions (`scanf`, `ascanf`, `fscanf`, etc.)

---

**Math**

```c
#include <math.h>
...
double x,y,z,pi,ampl=3.0,sigma=1.2;
pi = 3.14159265358979;
x = sin(60.0*pi/180.0);
y = sqrt(fabs(2*x + pi));
z = ampl*exp(-0.5*pow(x/sigma,2.0))
```

**Notes:**
- Must include `math.h`
  - if compiling on Linux/Unix, use `-lm` flag to link math
  - note mixed assignment in variable declarations
  - `fabs` is "floating absolute value", and here keeps `sqrt` from getting a negative argument
  - otherwise result could generate `NaN` (Not A Number)
  - `pow(x,y)` raises `x` to the `y` power (`x^y`)

---

**For Loops**

```c
int k,count;
count = 0;
for (k=0; k < 10; k++)
{
    count += 1;
    count %= 4;
    printf("count = %d\n",count);
}
```

**Notes:**
- declared more than one integer on same line (common practice)
- `k` starts at zero, remains less than 10 (will stop at 9), increments by one each time through loop
- `x++` adds one to variable: same as `k += 1`; same as `x = x + 1`;
- `add1 one to count each time (see rule above)`
- "mods" count by 4 (remainder of count/4)
- output is: 1, 2, 3, 0, 1, 2, 3, 0, 1, 2
- could (and often do) use `k` as `int value within loop`
- for `;;` is a way to get an indefinite loop (Ctrl-C to quit)

---

**Math Warnings**

**Number one mistake by C newbies: disrespecting variable type**

```c
int i,j=2,k=3;
double x,y,z;
i = 2/3;
s = 2/3;
y = 2/3.0;
z = 2.0/3;
printf("i = %d; x = %f; y = %f; z = %f; other = %f\n",i,x,y,z,j/k);
i = 0; x = 0.000000; y = 0.666667; z = 0.666667; other = 0.000000
```

- `i` is an integer, so 2/3 truncates to zero
- even though `x` is a double, 2/3 performs integer math, then converts to double
  - "other" value in `printf` shows same is true if `j/k` used
- as long as one value is a float, it does floating-point math

---

Winter 2012
C-Programming, Part 1

Casting

- when necessary, one may "cast" a value into a type of your choice:
  - `(double) j → 2.0`
  - `(double) j/k → 0.666667`
  - `j/(double) k → 0.666667`
  - `(double) (j/k) → 0.000000 (integer math already done)`
  - `(int) 6.824786 → 6`
- lesson is to take care when mixing variable types
- also, get into habit of putting .0 on floating point math numbers, even if strictly unnecessary

Sample code (stripped down to fit on slide)

```c
#include <stdio.h>
#include <conio.h>
#include <windows.h>

#define PPORT_BASE 0xD010  // usu. 0x378 if built-in

typedef void (_stdcall *oupfuncPtr)(short portaddr, short datum);

oupfuncPtr oup32fp;

void Out32(short portaddr, short datum){
  (oup32fp)(portaddr,datum);
}

int main(void)
{
  HANDLE hLib;
  short x=0xAA;

  // value to write (expr. in hex)
  hLib = LoadLibrary("inpout32.dll");
  oup32fp = (oupfuncPtr) GetProcAddress(hLib, "Out32");
  Out32(PPORT_BASE,x); // the actual output command

  FreeLibrary(hLib);
  return 0;
}
```

Talking to the Parallel Port in Windows

- We will use the input32.dll package
  - parallel port access in linux/unix is very straightforward
  - Windows 98 and before was also easy
  - new Hardware Abstraction Layer (HAL) gets in the way
  - this input32 package bridges the HAL
  - see www.linux4u.net to get the package (already installed on MHA-5574 machines)
  - http://www.hytherion.com/beatltd/pib니port.htm for test programs
- Can also access via LPT file handle
  - discussed at end of lecture
  - runs 25 times slower than the input32 version
    - because you have to open/close the port all the time

Looping to make a waveform

```c
short outval = 0;
for (;;) {      // way to make infinite loop: ^C kills
  outval += 1;
  outval %= 256;
  Out32(PPORT_BASE,outval);
}
```

- The code above makes a ramp of output values, then cuts down to zero and starts again
- repeat until Ctrl-C kills it
- Each time:
  - the outval is increased by 1
    - statement equivalent to `outval = outval + 1`
  - then mod by 256 (256→0, and start over)
    - statement is equivalent to `outval = outval % 256`
#include <stdio.h>
#include <unistd.h>     // needed for ioperm()
#include <asm/io.h>     // for outb() and inb()
#define DATA 0x378      // parallel port memory address

int main()
{
    int x = 0xAA;
    if (!ioperm(DATA,3,1))
    {
        printf("You must be root to run this program\n");
        exit(1);
    }
    outb(x,DATA);     // sends 1010 1010 to the Data Port
    return 0;
}

outb() performs direct write to hardware/memory address