Avoid Big If-Else Structures

If you had something like this:

```
if (a == 0) b = 8;
else if (a == 1) b = 4;
else if (a == 2) b = 10;
else b = 7;
```

Then you may want to do this:

```
switch(a) {
    case 0: b = 8; break;
    case 1: b = 4; break;
    case 2: b = 10; break;
    default: b = 7;
}
```

But this would be even better (though would require some checking):

```
int B[] = {8, 4, 10, 7};
b = B[a];
```

What if you had this:

```
int x;
char *msg = ...
if (strcmp(msg, CMD_1) == 0) x = foo();
else if (strcmp(msg, CMD_2) == 0) x = bar();
else if (strcmp(msg, CMD_3) == 0) x = baz();
...
```

Rather than using if/else, use a Map and function pointers:

```
hashtable h = ...
put(h, CMD_1, &foo);
put(h, CMD_2, &bar);
...
int (*my_fun)(void); // pointer to function that takes void and
                      // returns int
char *msg = ...
my_fun = get(h, msg);
x = my_fun(); // need some error checking
```

Does it always make sense to do it this way?
Calling Functions

passing structs: pointers are better because you don't have to copy the entire thing onto the stack, even if you're not changing it

```c
struct hashtable {
    int capacity;
    node *list[1024];
};

struct hashtable h;
put("foo", h);
put("moo", &h);
```

How many things put on the stack in the first case? 1025
How many in the second? 1

Fun With Math!

integer arithmetic – fast
floating point arithmetic in hardware – slower
floating point arithmetic in software – slowest

general consensus is to avoid using floating point values – unless you know you have hardware support
```c
float f = ...
int i = f * 100; // convert to int
// do stuff with i
f = i / 100;     // convert back to float
```

on my machine:
1 billion int adds: 2.8s
1 billion int mults: 2.8s
1 billion int divides: 15s
1 billion int <<=: 2.8s
1 billion register int adds: 2.6s
1 billion float adds: 2.8s
1 billion float mults: 2.8s
1 billion float divides: 4.7s
1 billion sqrt: 10.5s
1 billion sin: 85s
1 billion pow: 220s

Most of the time, no speed gain from using smaller int datatypes

use int instead of float when possible???
multiplying by powers of 2
instead of \( x = y \times 68 \) use \( x = (y \times 64) + (y \times 4) \) … or \( x = y \ll 6 + y \ll 2 \)

faking multiplication: use elementary-school style but use shifting and adding (only useful if you don't have a hardware multiplier)

\[
\begin{array}{c}
43 \times 13 \\
\hline
101011 \\
\times \ 1101 \\
\hline
101011 = 43 \\
10101100 = 43 \ll 2 \\
101011000 = 43 \ll 3 \\
\hline
43 \times 13 = 43 + 43 \ll 2 + 43 \ll 3 \\
\end{array}
\]

\[
43 \times 14 = 43 \times (16 – 2) = (43 \times 16) – (43 \times 2) = 43 \ll 4 – 43 \ll 1
\]

harder to fake division, except when you're dividing by powers of 2

division is a lot slower than multiplication
if \((a / b > c)\) should be if \((a > b \times c)\)
should \(x = y / 2.5\) be \(x = y \times 0.4\) instead?

may be able to rewrite division as multiplication, and then use shift-and-add, e.g. \(a / 1.333 = a \times 0.75 = a \times 0.5 + a \times 0.25 = a \ll 1 + a \ll 2\)

use bit vectors when possible: less memory, faster
bit manipulation
\[
a \|= 0x4; // set bit 2 \\
b \&= \sim 0x4; // clear bit 2 \\
c \&= \sim (1 \ll 3); // clear bit 3 \\
d \^= (1 \ll 5); // toggle bit 5 \\
e \gg= 2; // divide e by 4 \\
get the nth bit of k: k = (k \ll n) & 1;
\]

avoid heap usage:
- allocation/deallocation unpredictably slow
- danger of exhausting memory
- danger of fragmentation

global variables can't be stored in registers – stack/local variables are considered to be faster
however, global variables don't have to be pushed onto the stack when a function is called

keywords:
• volatile: tells the compiler not to attempt to optimize it because it might be used by multiple threads and programmer wants to control it; generally avoided because it means the value can't be stored in a register
• register: store the variable in a register if possible; can't take its address, though
• neither are used that much anymore