Strategy for developing efficient programs:

1. Design the program well
2. Implement the program well**
3. Test the program well
4. Only after you're sure it's working, *measure* performance
5. If (and only if) performance is inadequate, *find* the "hot spots"
6. *Tune* the code to fix these
7. Repeat measure-analyse-tune cycle until performance ok

(** see "Programming Pearls", "Practice of Programming", etc. etc.)
Rapid development of a prototype may be the best way to discover/assess performance issues.
Hence Fred Brooks maxim - "Plan To Throw One Away".

Two C Fucntions which Initialize an array

```c
void test0(int x, int y, int a[x][y]) {
    fprintf(stderr, "writing to array i-j order\n");
    for (int i = 0; i < x; i++)
        for (int j = 0; j < y; j++)
            a[i][j] = i+j;
}
```

```c
void test1(int x, int y, int a[x][y]) {
    fprintf(stderr, "writing to array j-i order\n");
    for (int j = 0; j < y; j++)
        for (int i = 0; i < x; i++)
            a[i][j] = i+j;
}
```

Performance Improvement Example - Caching

Although the loops are almost identical, the first loop runs 20x faster on a large array!

```
$ time ./cachegrind_example 0 32000 32000
allocating a 32000x32000 array = 4096000000 bytes
writing to array i-j order
real    0m0.893s
user    0m0.364s
sys     0m0.524s
$ time ./cachegrind_example 1 32000 32000
allocating a 32000x32000 array = 4096000000 bytes
writing to array j-i order
real    0m15.189s
user    0m14.633s
sys     0m0.528s
```

Performance Improvement Example - Caching

The tool valgrind used to detect accesses to uninitialized variables at runtime also can give memory caching infomation.
The memory subsystem is beyond the scope of this course but you can see valgrind explain the performance difference between these loops.
For the first loop D1 miss rate = 24.8%
For the second loop D1 miss rate = 99.9%
Due to the C array memory layout the first loop produces much better caching performance.
Tuning caching performance is important for some application and valgrind makes this much easier.

Performance Improvement Example - Caching

```
$ valgrind '--tool=cachegrind' ./cachegrind_example 0 10000 10000
allocating a 10000x10000 array = 400000000 bytes
writing to array i-j order
==7025==
==7025== I   refs:      225,642,966
==7025== I1  misses:            882
==7025== LLi misses:            875
==7025== I1  miss rate:        0.00%
==7025== LLi miss rate:        0.00%
==7025==
==7025== D   refs:       25,156,289  (93,484 rd   + 25,062,805 wr)
==7025== D1  misses:      6,262,957  ( 2,406 rd   +  6,260,551 wr)
==7025== LLd misses:      6,252,482  ( 1,982 rd   +  6,250,500 wr)
==7025== D1  miss rate:        24.8% (    2.5%    +       24.9%  )
==7025== LLd miss rate:        24.8% (    2.1%    +       24.9%  )
==7025==
==7025== LL refs:         6,263,839  ( 3,288 rd   +  6,260,551 wr)
==7025== LL misses:       6,253,357  ( 2,857 rd   +  6,250,500 wr)
==7025== LL miss rate:          2.4% (    0.0%    +       24.9%  )
```

Performance Improvement Example - Caching

```
$ valgrind '--tool=cachegrind' ./cachegrind_example 1 10000 10000
allocating a 10000x10000 array = 400000000 bytes
writing to array j-i order
==7006==
==7006== I   refs:      600,262,960
==7006== I1  misses:            876
==7006== LLi misses:            869
==7006== I1  miss rate:        0.00%
==7006== LLi miss rate:        0.00%
==7006==
==7006== D   refs:      100,056,288  (43,483 rd   + 100,012,805 wr)
==7006== D1  misses:    100,002,957  ( 2,405 rd   + 100,000,552 wr)
==7006== LLd misses:      6,262,481  ( 1,982 rd   +   6,260,499 wr)
==7006== D1  miss rate:        99.9% (   5.5%     +        99.9%  )
==7006== LLd miss rate:         6.2% (   4.5%     +         6.2%  )
==7006==
==7006== LL refs:       100,003,833  ( 3,281 rd   + 100,000,552 wr)
==7006== LL misses:       6,263,350  ( 2,851 rd   +   6,260,499 wr)
==7006== LL miss rate:          0.8% (   0.0%     +         6.2%  )
```

Where is execution time being spent?

Typically programs spend most of their execution time in a small part of their code.
This is often quoted as the 90/10 rule (or 80/20 rule or ...):

"90% of the execution time is spent in 10% of the code"

This means that

- most of the code has little impact on overall performance
- small parts of the code account for most execution time

We should clearly concentrate efforts at improving execution spped in the 10% of code which accounts for most of the execution time.

clang -p/gprof

Given the -p flag clang instruments a C program to collect profile information
When the program executes this data is left in the file gmon.out.
The program gprof analyzes this data and produces:

- number of times each function was called
- % of total execution time spent in the function
- average execution time per call to that function
- execution time for this function and its children

Arranged in order from most expensive function down.
It also gives a *call graph*, a list for each function:

- which functions called this function
- which functions were called by this function

Program for producing sorted counts of words

```
[illegible]

// Written by andrewt@cse.unsw.edu.au as an COMP1921 lecture example

/*
 * Written in C for "speed" but slow on large inputs
 *
 * $ gcc -O3 -o word_frequency0  word_frequency0.c
 * $ time word_frequency0 <WarAndPeace.txt >/dev/null
 * real    0m52.726s
 * user    0m52.642s
 * sys     0m0.020s
 */

/*
 * Profiling with gprof reveals get function is problem
 *
 * gcc -p -g word_frequency0.c -o word_frequency0_profile
 * head -10000 WarAndPeace.txt|word_frequency0_profile >/dev/null
 * $ gprof word_frequency0_profile
 *  %   cumulative   self              self     total
 * time   seconds   seconds    calls  ms/call  ms/call  name
 88.90      0.70      0.70    88335     0.01     0.01  get
  7.88      0.86      0.07     7531     0.01     0.01  put
  3.25      0.86      0.02    88805     0.00     0.00  get_word
  1.13      0.80      0.01        1    10.02   823.90  read_words
  0.00      0.80      0.00        2     0.00     0.00  size
  0.00      0.80      0.00        1     0.00     0.00  create_map
  0.00      0.80      0.00        1     0.00     0.00  keys
  0.00      0.80      0.00        1     0.00     0.00  sort_words
 ...
 */

#include <stdlib.h>
#include "time.h"
#include <string.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/*
 * returns the next word from the stream
 * a word is a non-zero length sequence of
 * alphabetic characters
 *
 * NULL is returned if there are no more words to be read
 */
char *
get_word(FILE *stream) {
    int i, c;
    char *w;
    static char *buffer = NULL;
    static int buffer_length = 0;

    if (buffer == NULL) {
        buffer_length = 32;
        buffer = malloc(buffer_length*sizeof (char));
        if (buffer == NULL) {
            fprintf(stderr, "out of memory\n");
            exit(1);
        }
    }

    i = 0;
    while ((c = fgetc(stream)) != EOF) {
        if (!isalpha(c) && i == 0)
            continue;
        if (!isalpha(c))
            break;
        if (i >= buffer_length) {
            buffer_length += 16;
            buffer = realloc(buffer, buffer_length*sizeof (char));
            if (buffer == NULL) {
                fprintf(stderr, "out of memory\n");
                exit(1);
            }
        }
        buffer[i++] = c;
    }

    if (i == 0)
        return NULL;

    buffer[i] = '\0';

    w = malloc(strlen(buffer) + 1);
    if (w == NULL) {
        fprintf(stderr, "out of memory\n");
        exit(1);
    }
    strcpy(w, buffer);
    return w;
}

typedef struct map  map;

struct map {
       int             size;
       struct map_node  *list;
};

struct map_node {
       char            *key;
       void            *value;
       struct map_node  *next;
};


map *
create_map() {
    struct map *m;
    if ((m = malloc(sizeof *m)) == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(1);
    }
    m->size = 0;
    m->list = NULL;
    return m;
}

void *get(map *m, char *key) {
    struct map_node *v;
    for (v = m->list; v != NULL; v = v->next) {
        if (strcmp(key, v->key) == 0) {
            return v->value;
        }
    }
    return NULL;
}

void
put(map *m, char *key, void *value) {
    struct map_node *v;
    for (v = m->list; v != NULL; v = v->next) {
        if (strcmp(key, v->key) == 0) {
            v->value = value;
            return;
        }
    }

    if ((v = malloc(sizeof *v)) == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(1);
    }
    v->key = key;
    v->value = value;
    v->next = m->list;
    m->list = v;
    m->size++;
}

int
size(map *m) {
    return m->size;
}

char **keys(map *m) {
    struct map_node *v;
    int  i, n_keys = size(m);
    char **key_array;

    if ((key_array = malloc(n_keys*sizeof (char *))) == NULL) {
        fprintf(stderr, "Out of memory\n");
        exit(1);
    }
    for (v = m->list, i=0; v != NULL; v = v->next,i++)
        key_array[i] = v->key;
    return key_array;
}

static void
free_map_nodes(struct map_node  *list) {
    if (list == NULL)
        return;
    free_map_nodes(list->next);
    free(list);
}

void free_map(map *m) {
    free_map_nodes(m->list);
    free(m);
}
/*
 * One word_count struct is malloc'ed for each
 * distinct word read
 */
struct word_count {
    int count;
};

/*
 * read the words from a stream
 * associate a word_count struct with
 * each new word
 *
 * increment the count field each time the
 * word is seen
 */
map *
read_words(FILE *stream) {
    char *word;
    struct word_count *w;
    map *m;

    m = create_map();
    while (1) {
        word = get_word(stdin);
        if (word == NULL)
            return m;
        w = get(m, word);
        if (w != NULL) {
            w->count++;
            free(word);
            continue;
        }
        if ((w = malloc(sizeof *w)) == NULL) {
            fprintf(stderr, "Out of memory\n");
            exit(1);
        }
        w->count = 1;
        put(m, word, w);
    }
}

void
sort_words(char **sequence, int length) {
    int i, j;
    char *pivotValue;
    char *temp;

    if (length <= 1)
        return;

    /* start from left and right ends */

    i = 0;
    j = length - 1;

    /* use middle value as pivot */

    pivotValue = sequence[length/2];
    while (i < j) {

        /* Find two out-of-place elements */

        while (strcmp(sequence[i], pivotValue) < 0)
            i++;
        while (strcmp(sequence[j], pivotValue) > 0)
            j--;
        /* and swap them over */

        if (i <= j) {
            temp = sequence[i];
            sequence[i] = sequence[j];
            sequence[j] = temp;
            i++;
            j--;
        }
    }
    sort_words(sequence, j + 1);
    sort_words(sequence+i, length - i);
}
int
main(int argc, char *argv[]) {
    int i, n_unique_words;
    char **key_array;
    map *m;

    m = read_words(stdin);

    key_array = (char **)keys(m);

    n_unique_words = size(m);

    sort_words(key_array, n_unique_words);

    for (i = 0; i < n_unique_words; i++) {
        struct word_count *w;
        w = (struct word_count *)get(m, key_array[i]);
        printf("%5d %s\n", w->count, key_array[i]);
    }

    return 0;
}
```

Performance Improvement Example - Word Count

Program is slow on large inputs e.g.

```
$ clang -O3 word_frequency0.c -o word_frequency0
$ time word_frequency0 <WarAndPeace.txt >/dev/null
real    0m52.726s
user    0m52.643s
sys     0m0.020s
```

Performance Improvement Example - Word Count

We can instrument the program to collect profiling information and examine it with clang

```
$ clang -p -g word_frequency0.c -o word_frequency0_profile
$ head -10000 WarAndPeace.txt|word_frequency0_profile >/dev/null
$ gprof word_frequency0_profile
....
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 88.90      0.79      0.79    88335     0.01     0.01  get
  7.88      0.86      0.07     7531     0.01     0.01  put
  2.25      0.88      0.02    80805     0.00     0.00  get_word
  1.13      0.89      0.01        1    10.02   823.90  read_words
  0.00      0.89      0.00        2     0.00     0.00  size
  0.00      0.89      0.00        1     0.00     0.00  create_map
  0.00      0.89      0.00        1     0.00     0.00  keys
  0.00      0.89      0.00        1     0.00     0.00  sort_words
....
```

Its clear that only the functions *get* and to a much lesser extent *put* are relevant to performance improvement.

Performance Improvement Example - Word Count

Examine *get* and we find it traverses a linked list.
So replace it with a binary tree and the program runs 200x faster on War and Peace.

```
$ clang -O3 word_frequency1.c -o word_frequency1
$ time word_frequency1 <WarAndPeace.txt >/dev/null
real    0m0.277s
user    0m0.268s
sys     0m0.008s
```

Was C the best choice for our count words program?

Performance Improvement Example - Word Count

Shell, Perl and Python are slower - but a lot less code.
So faster to write, less bugs to find, easier to maintain/modify

```
$ time word_frequency1 <WarAndPeace.txt >/dev/null
real    0m0.277s
user    0m0.268s
sys     0m0.008s
$ time word_frequency.sh <WarAndPeace.txt >/dev/null
real    0m0.564s
user    0m0.584s
sys     0m0.036s
$ time word_frequency.pl <WarAndPeace.txt >/dev/null
real    0m0.643s
user    0m0.632s
sys     0m0.012s
$ time word_frequency.py <WarAndPeace.txt >/dev/null
real    0m1.046s
user    0m0.836s
sys     0m0.024s
$ wc word_frequency*.*
 286  759 5912 word_frequency1.c
   8   19   82 word_frequency.sh
  11   38  325 word_frequency.py
  14   43  301 word_frequency.pl
```

Performance Improvement Example - cp - read/write

Here is a cp implementation in C using low-level calls to read/write

```c
while (1) {
    char c[1];
    int bytes_read = read(in_fd, c, 1);
    if (bytes_read < 0) {
            perror("cp: ");
            exit(1);
    }
    if (bytes_read == 0)
        return;
    int bytes_written = write(out_fd, c, bytes_read);
    if (bytes_written <= 0) {
        perror("cp: ");
        exit(1);
    }
}
```

Performance Improvement Example - cp

Its suprisingly slow compared to /bin/cp

```
$ time /bin/cp input_file /dev/null
real    0m0.006s
user    0m0.000s
sys     0m0.004s
$ clang  cp0.c -o cp0
$ time ./cp0 input_file /dev/null
real    0m6.683s
user    0m0.932s
sys     0m5.740s
$ clang -O3 cp0.c -o cp0
$ time ./cp0 input_file /dev/null
real    0m6.688s
user    0m0.900s
sys     0m5.776s
```

Performance Improvement Example - cp

- most execution time is spent executing system calls
- as a consequence -O3 is no help.
- 2 system calls for every byte copied - a huge overhead.
- modify it to buffer its I/O
- make 2 system calls for every 8192 bytes copied - should run much faster.

Performance Improvement Example - cp

```c
while (1) {
    char c[8192];
    int bytes_read = read(in_fd, c, sizeof c);
    if (bytes_read < 0) {
        perror("cp: ");
        exit(1);
    }
    if (bytes_read <= 0)
        return;
    int bytes_written = write(out_fd, c, bytes_read);
    if (bytes_written <= 0) {
        perror("cp: ");
        exit(1);
    }
}
```

```
$ time ./cp1 input_file /dev/null
real    0m0.005s
user    0m0.000s
sys     0m0.008s
```

Performance Improvement Example - cp

```c
while (1) {
    int ch = fgetc(in);
    if (ch == EOF)
        break;
    if (fputc(ch, out) == EOF) {
        perror("");
        exit(1);
    }
}
```

Using portable stdio library a byte-by-byte loop runs quite fast, because stdio buffers the I/O behind the scenes.

```
$ time ./cp2 input_file /dev/null
real    0m0.456s
user    0m0.448s
sys     0m0.008s
```

Performance Improvement Example - cp

```c
while (1) {
    if(fgets(input, sizeof input, in) == NULL) {
        break;
    }
    if (fprintf(out, "%s", input) == EOF) {
        fprintf(stderr, "cp:");
        perror("");
        exit(1);
    }
}
```

And with a little more complex code we get reasonable speed with portability:

```
$ clang -O3 cp3.c -o cp3
$ time ./cp3 input_file /dev/null
real    0m0.095s
user    0m0.084s
sys     0m0.012s
```

Performance Improvement Example - cp

For comparison Perl code which does a copy via an array of lines:

```
die "Usage: cp <src> <dest>\n" if @ARGV != 2;
$in = shift @ARGV;
$out = shift @ARGV;
open IN, '<', $in or
die "Cannot open $in: $!\n";
open OUT, '>', $out or die "Cannot open $out: $!\n";
print OUT <IN>;
```

```
$ time ./cp4.pl input_file /dev/null
real     0m0.248s
user     0m0.168s
sys      0m0.032s
\end{verbatim}
```

Performance Improvement Example - cp

And Perl code which unsets Perl's line terminator variable so a single read returns the whole file:

```
die "Usage: cp <infile> <outfile>\n" if @ARGV != 2;
$infile = shift @ARGV;
$outfile = shift @ARGV;
open IN, '<', $infile or die "Cannot open $infile: $!\n";
open OUT, '>', $outfile or die "Cannot open $outfile: $!\
undef $/;
print OUT <IN>;
```

```
$ time ./cp5.pl input_file /dev/null
real    0m0.029s
user    0m0.008s
sys     0m0.020s
```

Performance Improvement Example - Fibonacci

Here is a simple Perl program to calculate the n-th Fibonacci number:

```perl
sub fib {
    my ($n) = @_;
    return 1 if $n < 3;
    return fib($n-1) + fib($n-2);
}
printf "fib(%d) = %d\n", $_, fib($_) foreach @ARGV;
```

It becomes slow near n=35.

```
$ time fib0.pl 35
fib(35) = 9227465
real    0m10.776s
user    0m10.729s
sys     0m0.016s
```

we can rewrite in C.

Performance Improvement Example - Fibonacci

```c
#include <stdio.h>
int fib(int n) {
    if (n < 3) return 1;
    return fib(n-1) + fib(n-2);
}
int main(int argc, char *argv[]) {
    for (int i = 1; i < argc; i++) {
        int n = atoi(argv[i]);
        printf("fib(%d) = %d\n", n, fib(n));
    }
}
```

Faster but the program's complexity doesn't change:

```
$ clang -O3 -o fib0 fib0.c
$ time fib0 45
fib(45) = 1134903170
real    0m4.994s
user    0m4.976s
sys     0m0.004s
```

Performance Improvement Example - Fibonacci

```perl
#!/usr/bin/perl -w
sub fib {
  my ($n) = @_;
  return 1 if $n < 3;
  $f{$n} = fib($n-1) + fib($n-2) if !defined $f{$n};
  return $f{$n};
}
printf "fib(%d) = %d\n", $_, fib($_) foreach @ARGV;
```

It is very easy to cache already computed results in a Perl hash.
This changes the program's complexity from exponential to linear.

```
$ time fib1.pl 45
fib(45) = 1134903170
real     0m0.004s
user     0m0.004s
sys      0m0.000s
```

Now for Fibonanci we could also easily change the program to an iterative form which would be linear too.
But memoization is a general technique which can be employed in a variety of situations to improve performnce.