Basics of Optimizing Software
The usual philosophy

- There are two rules for when to optimize:

  1. Don’t do it!
  2. (for experts only) Don’t do it yet!

*Michael A. Jackson, c.1975*
Don’t think about optimization until you have a working system

- Make sure before you begin that your solution will fit on the target application
  - Estimate code size, run time, constraints
  - Engineer the solution before you start coding
- Once you start coding, get a working solution before trimming time/space
- Optimization takes a lot of time and brainwork. Make sure it’s absolutely necessary
- It also sacrifices readability, maintainability, and portability.
Signs you might need to optimize

note: in a given section of code, you can optimize for time or space, not both

• Critical sections of code slow down noticeably during heavy computation
• Erratic performance on input, output, or certain interrupts
• CPU pegged at 100% or overheating
• Insufficient memory, stack overwrites, massive disk swapping (if you have a disk)

• Should you really need a 233 Mhz processor with 64 Meg of RAM for an elevator controller in a 5 story building?
First steps:
use the tools on your desktop

- Locate bottlenecks; problem loops & tasks
  - Indicators (print statements, led drivers)
  - Masking suspicious code (comment out)
  - Debug tools (source level, task level, …)
  - Profiler tools (example on next slide)
  - Oscilloscope
Profiler tool in VC++

"Enable Profiling" under "Build->Settings-> Link"

Results
Easy fixes

- Compiler optimization levels
  - In command-line compilers: -O1, -O2, -O3, -inline, -TRE, -ffast –unroll_loops, etc.

- In VC++:
More Visual C++ optimizations:

- Highly dependent on the processor and the guy who wrote your compiler.
  - Start with the lowest optimization level and move up
- Have a test procedure to see if/how your optimizations are working!
Use a better algorithm

- Linear search:

  worst case: it isn’t here. N searches to fail.

- Binary search:

  worst case: it isn’t here. Log N searches to fail.
Know your machine

• A program on a processor board with an FPU (floating point math chip) may run faster when ints are converted to floats first.
• Pentium with MMX and AMD’s K-6 3DNow! are designed to do matrix operations very quickly
  – If your compiler supports them, that is
• How does your chip do multiplication of ints? Floats?
  – These are things you have to look up in books, newsgroups, or in processor manuals
Relative order of operation speed:
(slowest to fastest)

- **I/O devices:**
  - User input
  - Tape drive
  - Network
  - CD-ROM
  - Hard Drive
  - Main memory
  - Register variables

- **Control Flow**
  - Switch statements
  - Function calls
  - If statements
  - While statements

- **Arithmetic operations:**
  - Transcendental functions
  - Square root
  - Modulo
  - Divide
  - Multiply
  - Add/subtract
  - Multiply by power of 2
  - Divide by power of 2
  - Modulo by power of 2
C-code optimization tricks

- Maximize use of fast operations
- Buffer input/output so you only read or write once
- “Inline” functions
  - Instead of making a function call, write out the code in main()
- Use tables and arrays to store intermediate values instead of re-calculating things
C optimization tricks: register keyword

• Declaring a variable “register” is a suggestion to the compiler to keep that variable in register memory on the CPU. A variable that you will use or change a lot can benefit from this:

```c
int main( void ){
    int A, B, C;
    ...
}
```

```c
int main(void){
    register int A, B;
    int C;
    ...
}
```
C optimization tricks: data types

- Every conversion from float to int, int to char, signed to unsigned, etc. incurs overhead.
  
  ```c
  while( fscanf("%f", &myLittleFloat) )
  {
    i_sum = i_sum + (int) myLittleFloat;
  }
  ```

- Do as much of your calculation as possible in one data type, then convert as a final step.
- Floats are often faster than doubles
- Ints are always faster than characters
  - Int is defined as the native word-length of the processor
  - All shorter or longer data types require extra “massaging”
C optimization tricks: memset

```c
int i, j, arr[50][20];

for (i = 0; i < 50; i++) {
    for (j = 0; j < 20; j++) {
        arr[i][j] = 0;
    }
}
```

or

```c
int arr[50][20];

memset(arr, 0,
       50*20*sizeof(int));
```

or

```c
int arr[50][20] = {0};
```
C optimization tricks: loop unrolling

for (i = 0; i < 100; i++){
    do_stuff(i);
}

or

i = 0;
while (i < 100){
    do_stuff(i);
    i++;
}

i = 0;
while (i < 100){
    do_stuff(i); i++;
    do_stuff(i); i++;
    do_stuff(i); i++;
    do_stuff(i); i++;
    do_stuff(i); i++;
    do_stuff(i); i++;
    do_stuff(i); i++;
    do_stuff(i); i++;
    do_stuff(i); i++;
}
More optimization tricks

- Re-write cascading if/elses with most common case on top

- Optimize AND/OR statements so the first argument is the most likely to succeed

  
  ```
  if( hatch_open && dark_outside ) {...}
  ```

- Upgrade your compiler
Summary

- Optimization is a powerful tool with a steep learning curve
- There are simple things you can do to make your code run a little faster. A good compiler will do a lot of those things for you if you know how to ask, but
  - (1) You don’t always have a good compiler
  - (2) You can usually do better if you have to
- It takes time and effort - only optimize if it’s very necessary or very easy
- Any time you make a change, test _everything_ all over again
End of Week 11

• Next week: Real Time Operating Systems, Multitasking, Communication and Synchronization
Resources for further info

- Optimization of a Computer Program in C
- Randall Hyde’s Assembly page