**Best Practice for Optimizing C Code**

## 1. Struct design

Don't put a large array into struct. Using pointer instead.

Don't put so many function pointer into struct. The struct data is only used to keep track the data state , so The function pointer should move to a function outside the struct. This will help to reduce memory footprint of struct. Example:

bad design:

```
typedef struct {
    void *elts;
    char  name[1024]; // 1024 element
    size_t size;
    ngx_uint_t nalloc;
   ngx_pool_t *pool;
   (void*)  (*ngx_array_push)(ngx_array* self);
   (void*) (*ngx_array_push_n)(ngx_array* self, ngx_uint_t  n);
} ngx_array_t;
```

Good design:

```
typedef struct {
    void *elts;
    char* name;
    size_t size;
    ngx_uint_t nalloc;
   ngx_pool_t *pool;
} ngx_array_t;

ngx_array_t *ngx_array_create(ngx_pool_t *p, ngx_uint_t n, size_t size);
void ngx_array_destroy(ngx_array_t *a);
void *ngx_array_push(ngx_array_t *a);
void *ngx_array_push_n(ngx_array_t *a, ngx_uint_t n);
```

Using union in case we have various type of struct,but We only need to access one type at once.

Example:

```
struct TYPEA {
  char data[30]; // or whatever
};

struct TYPEB {
  double x, y; // or whatever
};

struct some_info {
  int type; // set accordingly
  union {
    struct TYPEA a;
    struct TYPEB b;
  } data; // access with some_info_object.data.a or some_info_object.data.b
};
```

**2. Heap Memory allocation**

- Using malloc to allocate memory. Avoid unnecessary data initialization. If you must initialize a large chunk of memory, consider using memset() or calloc to allocate memory

- Use realloc to extend memory allocation if it's allocated before, in order to minimize memory fragmentation.

- The size of the heap is limited to the amount of RAM left over after all of the global data and stack space has been allocated. If the heap is too small, your program will not be able to allocate memory when it is needed. So always be sure to compare the result of malloc/realloc with NULL before dereferencing it

Example:

```
int * ptr = (int*) malloc(10 * sizeof(int));

if(ptr == NULL)

// Unable to allocate memory. Take Action.

else

// Memory allocation successful. can use ptr
```

**3. Avoid dynamic memory allocation during computation**

- Dynamic memory is great for storing the scene and other data that does not change during computation.

- However, on many (most) systems dynamic memory allocation requires the use of locks to control a access to the allocator. For multi-threaded applications that use dynamic memory, you may actually get a slowdown by adding additional processors, due to the wait to allocate and free memory!

- Even for single threaded applications, allocating memory on the heap is more expensive than adding it on the stack. The operating system needs to perform some computation to find a memory block of the requisite size

**4. Jumps/branches are expensive. Minimize their use whenever possible**.

- Function calls require two jumps, in addition to stack memory manipulation.

- Prefer iteration over recursion. If you must perform recursion, we should take advatage of tail recursion elimiation stype to improve performent.

**5. Reduce Padding**

On machines with alignment restrictions you can sometimes save a tiny amount of space by arranging similarly-typed fields together in a structure with the most restrictively aligned types first. There may still be padding at the end, but eliminating that can destroy the performance gains the alignment was meant to provide.

old code:

```
/* sizeof = 64 bytes */
 struct foo {
   float   a;
   double  b;
   float   c;
   double  d;
   short   e;
   long    f;
   short   g;
   long    h;
   char    i;
   int     j;
   char    k;
   int     l;
 };
```

new code:

```
/* sizeof = 48 bytes */
 struct foo {
   double  b;
```

```
  double  d;
  long    f;
  long    h;
  float   a;
  float   c;
  int     j;
  int     l;
  short   e;
  short   g;
  char    i;
  char    k;
};
```

A typical use of char or short variables is to hold a flag or mode bit. You can combine several of these flags into one byte using bit-fields at the cost of data portability. You might instead use bitmasks and the & operator to extract the values; this is more portable but also more tedious.

**6. INCREASE PADDING**

Paradoxically, increasing the size and alignment of a data structure to match (or to be an integer fraction or multiple of) the cache line size may increase performance. The rationale is that if the data structure is an odd size relative to the cache line size, it may overlap two cache lines thus doubling the time needed to read it in from main memory.

The size of a data structure can be increased by adding a dummy field onto the end, usually a character array. The alignment is harder to control, but usually one of these techniques will work:

- Use `malloc` instead of a static array. Some `malloc`s automatically allocate storage suitably aligned for cache lines. (And some don't.)
- Allocate a block twice as large as you need, then point wherever in it that satisfies the alignment you need.
- Use an alternate allocator (e.g. `memalign`) which guarantees minimal alignment.
- Use the linker to assign specific addresses or alignment requirements to symbols.
- Wedge the data into a known position inside another block which is already aligned.
- Use word-size variables if you can, as the machine can work with these better (instead of `char`, `short`, `double`, bit fields etc.).

**7. Integers**

We should use unsigned int instead of int if we know the value will never be negative. Some processors can handle unsigned integer arithmetic considerably faster than signed (this is also good practice, and helps make for self-documenting code).

So, the best declaration for an int variable in a tight loop would be:

register unsigned int variable_name;

although, it is not guaranteed that the compiler will take any notice of register, and unsigned may make no difference to the processor. But it may not be applicable for all compilers.

Remember, integer arithmetic is much faster than floating-point arithmetic, as it can usually be done directly by the processor, rather than relying on external FPUs or floating point math libraries.

We need to be accurate to two decimal places (e.g. in a simple accounting package), scale everything up by 100, and convert it back to floating point as late as possible.

If your compiler support C99. Consider to use uinit{8,16,32,64}_t defined int stdint.h

**8. Division and Remainder**

In standard processors, depending on the numerator and denominator, a 32 bit division takes 20-140 cycles to execute. The division function takes a constant time plus a time for each bit to divide.

Time (numerator / denominator) = C0 + C1* log2 (numerator / denominator)
= C0 + C1 * (log2 (numerator) - log2 (denominator)).

The current version takes about 20 + 4.3N cycles for an ARM processor. As an expensive operation, it is desirable to avoid it where possible. Sometimes, such expressions can be rewritten by replacing the division by a multiplication. For example, (a / b) > c can be rewritten as a > (c * b) if it is known that b is positive and b *c fits in an integer. It will

be better to use unsigned division by ensuring that one of the operands is unsigned, as this is faster than signed division.

**9. Combining division and remainder**

Both dividend (x / y) and remainder (x % y) are needed in some cases. In such cases, the compiler can combine both by calling the division function once because as it always returns both dividend and remainder. If both are needed, we can write them together like this example:

```
int func_div_and_mod (int a, int b) {

    return (a / b) + (a % b)

}
```

**10. Division and remainder by powers of two**

We can make a division more optimized if the divisor in a division operation is a power of two. The compiler uses a shift to perform the division. Therefore, we should always arrange, where possible, for scaling factors to be powers of two (for example, 64 rather than 66). And if it is unsigned, then it will be more faster than the signed division.

```
typedef unsigned int uint;
```

```
uint div32u (uint a) {
 return a / 32;
}
int div32s (int a){
 return a / 32;
}
```

Both divisions will avoid calling the division function and the unsigned division will take fewer instructions than the signed division. The signed division will take more time to execute because it rounds towards zero, while a shift rounds towards minus infinity.

**11. An alternative for modulo arithmetic**

We use remainder operator to provide modulo arithmetic. But it is sometimes possible to rewrite the code using if statement checks.

Consider the following two examples:

```
uint modulo_func1 (uint count)
{
  return (++count % 60);
}

uint modulo_func2 (uint count)
{
  if (++count >= 60)
 count = 0;
 return (count);
}
```

The use of the if statement, rather than the remainder operator, is preferable, as it produces much faster code. Note that the new version only works if it is known that the range of count on input is 0-59.

**12. Using array indices**

If you wished to set a variable to a particular character, depending upon the value of something, you might do this:

```
switch ( queue ) {
case 0 :   letter = 'W';
  break;
case 1 :   letter = 'S';
  break;
case 2 :   letter = 'U';
  break;
}
```

Or maybe:

```
if ( queue == 0 )
  letter = 'W';
else if ( queue == 1 )
  letter = 'S';
else
  letter = 'U';
```

A neater (and quicker) method is to simply use the value as an index into a character array, e.g.:

```
static char *classes="WSU";

letter = classes[queue];
```

**13. Global variables**

Global variables are never allocated to registers. Global variables can be changed by assigning them indirectly using a pointer, or by a function call. Hence, the compiler cannot cache the value of a global variable in a register, resulting in extra (often unnecessary) loads and stores when globals are used. We should therefore not use global variables inside critical loops.

If a function uses global variables heavily, it is beneficial to copy those global variables into local variables so that they can be assigned to registers. This is possible only if those global variables are not used by any of the functions which are called.

For example:

```
int f(void);
int g(void);
int errs;
void test1(void)
{
  errs += f();
  errs += g();
}

void test2(void)
{
  int localerrs = errs;
  localerrs += f();
  localerrs += g();
  errs = localerrs;
}
```

Note that test1 must load and store the global errs value each time it is incremented, whereas test2 stores localerrs in a register and needs only a single instruction.

**14. Using Aliases**

Consider the following example -

```
void func1( int *data )
{
   int i;

   for(i=0; i<10; i++)
   {
       anyfunc( *data, i);
   }
}
```

Even though *data may never change, the compiler does not know that anyfunc () did not alter it, and so the program must read it from memory each time it is used - it may be an alias for some other variable that is altered elsewhere. If we know it won't be altered, we could code it like this instead:

```
void func1( int *data )
{
   int i;
   int localdata;

   localdata = *data;
   for(i=0; i<10; i++)
   {
       anyfunc ( localdata, i);
   }
}
```

This gives the compiler better opportunity for optimization.

**15. Live variables and spilling**

As any processor has a fixed set of registers, there is a limit to the number of variables that can be kept in registers at any one point in the program.

Some compilers support live-range splitting, where a variable can be allocated to different registers as well as to memory in different parts of the function. The live-range of a variable is defined as all statements between the last assignment to the variable, and the last usage of the variable before the next assignment. In this range, the value of the variable is valid, thus it is alive. In between live ranges, the value of a variable is not needed: it is dead, so its register can be used for other variables, allowing the compiler to allocate more variables to registers.

The number of registers needed for register-allocatable variables is at least the number of overlapping live-ranges at each point in a function. If this exceeds the number of registers available, some variables must be stored to memory temporarily. This process is called spilling.

The compiler spills the least frequently used variables first, so as to minimize the cost of spilling. Spilling of variables can be avoided by:

- Limiting the maximum number of live variables: this is typically achieved by keeping expressions simple and small, and not using too many variables in a function. Subdividing large functions into smaller, simpler ones might also help.
- Using register for frequently-used variables: this tells the compiler that the register variable is going to be frequently used, so it should be allocated to a register with a very high priority. However, such a variable may still be spilled in some circumstances.

**16. Local variables**

Where possible, it is best to avoid using char and short as local variables. For the types char and short, the compiler needs to reduce the size of the local variable to 8 or 16 bits after each assignment. This is called sign-extending for signed variables and zero extending for unsigned variables. It is implemented by shifting the register left by 24 or 16 bits, followed by a signed or unsigned shift right by the same amount, taking two instructions (zero-extension of an unsigned char takes one instruction).

These shifts can be avoided by using int and unsigned int for local variables. This is particularly important for calculations which first load data into local variables and then process the data inside the local variables. Even if data is input and output as 8- or 16-bit quantities, it is worth considering processing them as 32-bit quantities.

Consider the following three example functions:

```
int wordinc (int a)
{
  return a + 1;
}
short shortinc (short a)
{
   return a + 1;
}
char charinc (char a)
{
   return a + 1;
}
```

The results will be identical, but the first code segment will run faster than others.

**17. Pointers**

If possible, we should pass structures by reference, that is pass a pointer to the structure, otherwise the whole thing will be copied onto the stack and passed, which will slow things down. I've seen programs that pass structures several Kilo Bytes in size by value, when a simple pointer will do the same thing.

Functions receiving pointers to structures as arguments should declare them as pointer to constant if the function is not going to alter the contents of the structure. As an example:

```
void print_data_of_a_structure ( const Thestruct  *data_pointer)
```

```
{
   ...printf contents of the structure...
}
```

This example informs the compiler that the function does not alter the contents (as it is using a pointer to constant structure) of the external structure, and does not need to keep re-reading the contents each time they are accessed. It also ensures that the compiler will trap any accidental attempts by your code to write to the read-only structure and give an additional protection to the content of the structure.

**18. Pointer chains**

Pointer chains are frequently used to access information in structures. For example, a common code sequence is:

```
typedef struct { int x, y, z; } Point3;
typedef struct { Point3 *pos, *direction; } Object;

void InitPos1(Object *p)
{
  p->pos->x = 0;
  p->pos->y = 0;
  p->pos->z = 0;
}
```

However, this code must reload p->pos for each assignment, because the compiler does not know that p->pos->x is not an alias for p->pos. A better version would cache p->pos in a local variable:

```
void InitPos2(Object *p)
{
  Point3 *pos = p->pos;
  pos->x = 0;
  pos->y = 0;
  pos->z = 0;
}
```

Another possibility is to include the Point3 structure in the Object structure, thereby avoiding pointers completely.

**19. Lazy Evaluation Exploitation**

In a if(a>10 && b=4) type of thing, make sure that the first part of the AND expression is the most likely to give a false answer (or the easiest/quickest to calculate), therefore the second part will be less likely to be executed.

**20. switch() instead of if...else...**

For large decisions involving if...else...else..., like this:

```
if( val == 1)
   dostuff1();
else if (val == 2)
   dostuff2();
else if (val == 3)
   dostuff3();
```

It may be faster to use a switch:

```
switch( val )
{
   case 1: dostuff1(); break;

   case 2: dostuff2(); break;

   case 3: dostuff3(); break;
}
```

In the if() statement, if the last case is required, all the previous ones will be tested first. The switch lets us cut out this extra work. If you have to use a big if..else.. statement, test the most likely cases first.

**21. TAIL RECURSION ELIMINATION**

First, let me define tail recursion elimination (TRE). When a recursive function calls itself, an optimizer can, under some conditions, replace the call with an assembly level equivalent of a "goto" back to the top of the function. The saves the effort of growing the stack, saving and restoring registers, and any other function call overhead. For very small recursive functions that make millions of recursive calls, TRE can result in a substantial speedup. With proper design, the TRE can take a recursive function and turn it into whatever is the fastest form of loop for the machine.

Tail recursion elimination (TRE for short) has been around for a long time. It originated with "functional" languages like LISP which do so much recursion that TRE is a necessity. C, C++, and the "pascal-like" languages fall into the "imperative" language category and extremely efficient programs, recursive or not, can be written and compiled without the benefit of TRE and still perform on par with (and often better than) a similar LISP program. What I'm leading up to is that TRE is not automatically present in every modern optimizer, though many certainly do have it.

Back to the conditions I mentioned earlier. In order for TRE to be a safe optimization the function must return the value of the recursive call without any further computation. Example:

```
int isemptystr(char * str)

{ if (*str == '\0') return 1;

     else if (! isspace(*str)) return 0;
```

```
    else return isemptystr(++str);
}
```

The above can have TRE applied to the final return statement because the returned value from this invocation of isemptystr will be exactly that of the n+1th invocation, with no further computation.

And now a another example:

```
int factorial(int num)
  {
    if (num == 0) return 1;
    else return num * factorial(num - 1);
  }
```

The above cannot have TRE applied because the returned value is not used directly: it is multiplied by num after the call, so the state of that invocation must be maintained until after the return. Even a compiler that supports TRE cannot use it here.

And now a another example, a rewrite of the factorial program to allow TRE optimization.

```
int factorial(int num, int factor)
  {
    if (num == 0) return factor;
    else return factorial(num - 1, factor * num);
  }
```

The optimizers for imperative languages don't bother to perform this sort of rewriting. I have seen it done with Scheme compilers, so it is possible. Programmers who write code this way (other than for example purposes!) should be beaten about the head and shoulders with a blunt dandruff-removing object.

Even if your compiler implements TRE optimization, you should not assume that it automatically has done so for you. You may have to rewrite even the simplest recursive function before TRE can be applied. If doing so reduces the readability of the function, the effort is highly questionable.

There is a large subset of recursive algorithms that have simple iterative counterparts. C compilers are very, very good at optimizing loops and can do so without the sometimes-onerous conditions that TRE requires, so if you must optimize at the source level, consider using plain iteration before TRE-friendly rewriting.

Finally, if the recursive function contains loops or large amounts of code, TRE will be only marginally helpful, since TRE only optimizes the recursion, not anything about the algorithm itself. Function calls are very fast, and optimizing out something that is already very fast will quickly run into the law of diminishing returns.

**23. TABLE LOOKUP**

Consider using lookup tables especially if a computation is iterative or recursive, e.g. convergent series or factorial. (Calculations that take constant time can often be recomputed faster than they can be retrieved from memory and so do not always benefit from table lookup.)

old Code:

```
 long factorial(int i)
  {
     if (i == 0)
        return 1;
     else
        return i * factorial(i - 1);
  }
```

new code:

```
 static long factorial_table[] =
     {1, 1, 2, 6, 24, 120, 720 /* etc */};
  long factorial(int i)
 {
    return factorial_table[i];
  }
```

If the table is too large to type, you can have some initialization code compute all the values on startup or have a second program generate the table and printf it out in a form suitable for #include-ing into a source file. At some point, the size of the table will have a noticeable affect on paging space. You could have the table cover the first N cases then augment this with a function to compute the rest. As long as a significant number of the values requested are in the table there will be a net win.

**24. STRENGTH REDUCTION**

Strength reduction is the replacement of an expression by a different expression that yields the same value but is cheaper to compute. Many compilers will do this for you automatically. The classic examples:

old code:

```
 x = w % 8;

  y = pow(x, 2.0);

  z = y * 33;

  for (i = 0; i < MAX; i++)

  {

     h = 14 * i;

     printf("%d", h);

  }
```

new code:

```
x = w & 7;            /* bit-and cheaper than remainder */

  y = x * x;          /* mult is cheaper than power-of */

  z = (y << 5) + y;      /* shift & add cheaper than mult */

  for (i = h = 0; i < MAX; i++)

  {

     printf("%d", h);

     h += 14;         /* addition cheaper than mult */

  }
```

Also note that array indexing in C is basically a multiply and an add. The multiply part can be subjected to strength reduction under some circumstances, notably when looping through an array.

**25. STACK USAGE**

A typical cause of stack-related problems is having large arrays as local variables. In that case the solution is to rewrite the code so it can use a static or global array, or perhaps allocate it from the heap. A similar solution applies to functions which have large structs as locals or

parameters. (On machines with small stacks, a stack bound program may not just run slow, it might stop entirely when it runs out of stack space.)

Recursive functions, even ones which have few and small local variables and parameters, can still affect performance. On some ancient machines there is a substantial amount of overhead associated with function calls, and turning a recursive algorithm into an iterative one can save some time.

A related issue is last-call optimization. Many LISP and Scheme compilers do this automatically, but few C compilers support it. Consider this example:

```
int func1()

  {

     int a, b, c, etc;



     do_stuff(a, b, c)

     if (some_condition)

        return func2();

     else

        return 1;

  }
```

Since func1()'s last statement is to call func2() and func1() has no need of its variables after that point, it can remove its stack frame. When func2() is done it returns directly to func1()'s caller. This (a) reduces the maximum depth of the stack and (b) saves the execution of some return-from-subroutine code as it will get executed only once instead of twice (or more, depending on how deeply the function results are passed along.)

If func1() and func2() are the same function (recursion) the compiler can do something else: the stack can be left in place and the call can be replaced by a goto back to the top of the function. This is called tail-recursion elimination.

## 26. SHARED LIBRARY OVERHEAD

Dynamically linked shared libraries are a really nifty thing. In many cases though, calling a dynamically linked function is slightly slower than it would be to call it statically. The principal part of this extra cost is a one-time thing: the first time a dynamically called function is called there is a bit of searching going on, but thereafter the overhead should be a very tiny number of instructions.

For applications with thousands of functions, there can be a noticeable lag at startup. Linking statically will reduce this, but defeats (to some extent) the benefits of code sharing that dynamic libraries can bring. Often, you can selectively link some libraries as dynamic and others as static. For example, the X11, C and math libraries you'd link dynamically (since other processes will be using these also and the program can use to the copy already in memory) but still link your own application-specific libraries statically.

## 27. MACHINE-SPECIFIC OPTIMIZATION

As with other machine-specific code, you can use #ifdef to set off sections of code which are optimized for a particular machine. Compilers don't predefine RISC or SLOW_DISK_IO or HAS_VM or VECTORIZING so you'll have to come up with your own and encode them into a makefile or header file.

## 28. VARIABLES

Avoid referring to global or static variables inside the tightest loops. Don't use the volatile qualifier unless you really mean it. Most compilers take it to mean roughly the opposite of register, and will deliberately not optimize expressions involving the volatile variable.

Avoid passing addresses of your variables to other functions. The optimizer has to assume that the called function is capable of stashing a pointer to this variable somewhere and so the variable could get modified as a side effect of calling what seems like a totally unrelated function. At less intense levels of optimization, the optimizer may even assume that a signal handler could modify the variable at any time. These cases interfere with placing variables in registers, which is very important to opimization. Example:

*a = b(); c(&d);*

Because d has had its address passed to another function, the compiler can no longer leave it in a register across function calls. It can however leave the variable a in a register. The register keyword can be used to track down problems like this; if d had been declared register the compiler would have to warn that its address had been taken.

## 29. FUNCTION CALLS

While functions and modularity are a good thing, a function call inside an oft-executed loop is a possible bottleneck. There are several facets to the problem, some of which I've alluded to above. Aside from the expense of executing the instructions in the other function:

- Function calls interrupt an optimizer's train of thought in a drastic way. Any references through pointers or to global variables are now "dirty" and need to be saved/restored across the function call. Local variables which have had their address taken and passed outside the function are also now dirty, as noted above.
- There is some overhead to the function call itself as the stack must be manipulated and the program counter altered by whatever mechanism the CPU uses.
- If the function being called happens to be paged out, there will be a very long delay before it gets read back in. For functions called in a loop it's unusual for the called function to be paged out until the loop is finished, but if virtual memory is scarce, calls to other functions in the same loop may demand the space and force the other

function out, leading to thrashing. Most linkers respect the order in which you list object files, so you can try to get functions near each other in hopes that they'll land on the same page.

**30. Binary Breakdown**

Break things down in a binary fashion, e.g. do not have a list of:

```
if(a==1) {
} else if(a==2) {
} else if(a==3) {
} else if(a==4) {
} else if(a==5) {
} else if(a==6) {
} else if(a==7) {
} else if(a==8)

{
}
```

Have instead:

```
if(a<=4) {

    if(a==1)     {
    }  else if(a==2)  {
    }  else if(a==3)  {
    }  else if(a==4)   {

    }
}
else
{
    if(a==5)  {
    } else if(a==6)   {
    } else if(a==7)  {
    } else if(a==8)  {
    }
}
```

Or even:

```
if(a<=4)

{
    if(a<=2)
    {
        if(a==1)
        {
            /* a is 1 */
        }
        else
        {
            /* a must be 2 */
        }
    }
    else
```

```
    {
        if(a==3)
        {
            /* a is 3 */
        }
        else
        {
            /* a must be 4 */
        }
    }
}
else
{
    if(a<=6)
    {
        if(a==5)
        {
            /* a is 5 */
        }
        else
        {
            /* a must be 6 */
        }
    }
    else
    {
        if(a==7)
        {
            /* a is 7 */
        }
        else
        {
            /* a must be 8 */
        }
    }
}
```

### 31. Loop termination

The loop termination condition can cause significant overhead if written without caution. We should always write count-down-to-zero loops and use simple termination conditions. The execution will take less time if the termination conditions are simple. Take the following two sample routines, which calculate n!. The first implementation uses an incrementing loop, the second a decrementing loop.

```
int fact1_func (int n)
{
    int i, fact = 1;
    for (i = 1; i <= n; i++)
      fact *= i;
    return (fact);
}

int fact2_func(int n)
{
    int i, fact = 1;
    for (i = n; i != 0; i--)
       fact *= i;
    return (fact);
```

```
}
```

As a result, the second one `fact2_func`" will be more faster than the first one.

### 32. Faster for() loops

It is a simple concept but effective. Ordinarily, we used to code a simple `for()` loop like this:

```
for( i=0;  i<10;  i++){ ... }
```

[ `i` loops through the values 0,1,2,3,4,5,6,7,8,9 ]

If we needn't care about the order of the loop counter, we can do this instead:

```
for( i=10; i--; ) { ... }
```

Using this code, `i` loops through the values 9,8,7,6,5,4,3,2,1,0, and the loop should be faster.

This works because it is quicker to process `i--` as the test condition, which says "Is `i` non-zero? If so, decrement it and continue". For the original code, the processor has to calculate "Subtract `i` from 10. Is the result non-zero? If so, increment `i` and continue.". In tight loops, this makes a considerable difference.

The syntax is a little strange, put is perfectly legal. The third statement in the loop is optional (an infinite loop would be written as `for( ; ; )`). The same effect could also be gained by coding:

```
for(i=10; i; i--){}
```

or (to expand it further):

```
for(i=10; i!=0; i--){}
```

The only things we have to be careful of are remembering that the loop stops at 0 (so if it is needed to loop from 50-80, this wouldn't work), and the loop counter goes backwards. It's easy to get caught out if your code relies on an ascending loop counter.

We can also use register allocation, which leads to more efficient code elsewhere in the function. This technique of initializing the loop counter to the number of iterations required and then decrementing down to zero, also applies to `while` and `do` statements.

### 33. Function Looping

Functions always have a certain performance overhead when they are called. Not only does the program pointer have to change, but in-use variables have to be pushed onto a stack, and new variables allocated. There is much that can be done then to the structure of a program's functions in order to improve a program's performance. Care must be taken though to maintain the readability of the program whilst keeping the size of the program manageable.

If a function is often called from within a loop, it may be possible to put that loop inside the function to cut down the overhead of calling the function repeatedly, e.g.:

```
for(i=0 ; i<100 ; i++)
{
    func(t,i);
}
-
-
-
void func(int w,d)
{
    lots of stuff.
}
```

Could become....

```
func(t);
-
-
-
void func(w)
{
    for(i=0 ; i<100 ; i++)
    {
        //lots of stuff.
    }
}
```

### 34. Loop unrolling

Small loops can be unrolled for higher performance, with the disadvantage of increased code size. When a loop is unrolled, a loop counter needs to be updated less often and fewer branches are executed. If the loop iterates only a few times, it can be fully unrolled, so that the loop overhead completely disappears.

This can make a **big** difference. It is well known that unrolling loops can produce considerable savings, e.g.:

```
for(i=0; i<3; i++){
    something(i);        something(0);
}                        something(1);
                         something(2);
//is less efficient than
```

because the code has to check and increment the value of `i` each time round the loop. Compilers will often unroll simple loops like this, where a fixed number of iterations is involved, but something like:

```
for(i=0;i< limit;i++) { ... }
```

is unlikely to be unrolled, as we don't know how many iterations there will be. It is, however, possible to unroll this sort of loop and take advantage of the speed savings that can be gained.

The following code (Example 1) is obviously much larger than a simple loop, but is much more efficient. The block-size of 8 was chosen just for demo purposes, as any suitable size will do - we just have to repeat the "loop-contents" the same amount. In this example, the loop-condition is tested once every 8 iterations, instead of on each one. If we know that we will be working with arrays of a certain size, you could make the block size the same size as (or divisible into the size of) the array. But, this block size depends on the size of the machine's cache.

```
//Example 1

#include<STDIO.H>

#define BLOCKSIZE (8)

void main(void)
{
int i = 0;
int limit = 33;  /* could be anything */
int blocklimit;

/* The limit may not be divisible by BLOCKSIZE,
 * go as near as we can first, then tidy up.
 */
blocklimit = (limit / BLOCKSIZE) * BLOCKSIZE;

/* unroll the loop in blocks of 8 */
while( i < blocklimit )
{
    printf("process(%d)\n", i);
    printf("process(%d)\n", i+1);
    printf("process(%d)\n", i+2);
    printf("process(%d)\n", i+3);
    printf("process(%d)\n", i+4);
    printf("process(%d)\n", i+5);
    printf("process(%d)\n", i+6);
    printf("process(%d)\n", i+7);

    /* update the counter */
    i += 8;

}

/*
 * There may be some left to do.
 * This could be done as a simple for() loop,
 * but a switch is faster (and more interesting)
 */

if( i < limit )
{
    /* Jump into the case at the place that will allow
     * us to finish off the appropriate number of items.
     */

    switch( limit - i )
    {
        case 7 : printf("process(%d)\n", i); i++;
        case 6 : printf("process(%d)\n", i); i++;
        case 5 : printf("process(%d)\n", i); i++;
```

```
        case 4 : printf("process(%d)\n", i); i++;
        case 3 : printf("process(%d)\n", i); i++;
        case 2 : printf("process(%d)\n", i); i++;
        case 1 : printf("process(%d)\n", i);
    }
}

}
```

### 35. Population count - counting the number of bits set

This example 1 efficiently tests a single bit by extracting the lowest bit and counting it, after which the bit is shifted out. The example 2 was first unrolled four times, after which an optimization could be applied by combining the four shifts of n into one. Unrolling frequently provides new opportunities for optimization.

```
//Example - 1

int countbit1(uint n)
{
  int bits = 0;
  while (n != 0)
  {
    if (n & 1) bits++;
    n >>= 1;
   }
  return bits;
}


//Example - 2

int countbit2(uint n)
{
   int bits = 0;
   while (n != 0)
   {
      if (n & 1) bits++;
      if (n & 2) bits++;
      if (n & 4) bits++;
      if (n & 8) bits++;
      n >>= 4;
   }
   return bits;
}
```

### 36. Early loop breaking

It is often not necessary to process the entirety of a loop. For example, if we are searching an array for a particular item, break out of the loop as soon as we have got what we need. Example: this loop searches a list of 10000 numbers to see if there is a -99 in it.

```
found = FALSE;
for(i=0;i<10000;i++)
{
    if( list[i] == -99 )
    {
```

```
        found = TRUE;
    }
}

if( found ) printf("Yes, there is a -99. Hooray!\n");
```

This works well, but will process the entire array, no matter where the search item occurs in it. A better way is to abort the search as soon as we've found the desired entry.

```
found = FALSE;
for(i=0; i<10000; i++)
{
    if( list[i] == -99 )
    {
        found = TRUE;
        break;
    }
}
if( found ) printf("Yes, there is a -99. Hooray!\n");
```

If the item is at, say position 23, the loop will stop there and then, and skip the remaining 9977 iterations.

**37. Function call overhead**

Function call overhead on the processor is small, and is often small in proportion to the work performed by the called function. There are some limitations up to which words of arguments can be passed to a function in registers. These arguments can be integer-compatible (`char`, `short`s, `int`s and `float`s all take one word), or structures of up to four words (including the 2-word `double`s and long `long`s). If the argument limitation is 4, then the fifth and subsequent words are passed on the stack. This increases the cost of storing these words in the calling function and reloading them in the called function.

In the following sample code:

```
int f1(int a, int b, int c, int d) {
   return a + b + c + d;
}

int g1(void) {
   return f1(1, 2, 3, 4);
}


int f2(int a, int b, int c, int d, int e, int f) {
  return a + b + c + d + e + f;
}

ing g2(void) {
 return f2(1, 2, 3, 4, 5, 6);
}
```

the fifth and sixth parameters are stored on the stack in `g2`, and reloaded in `f2`, costing two memory accesses per parameter.

## 38. Inline functions

Function inlining is disabled for all debugging options. Functions with the keyword `__inline` results in each call to an inline function being substituted by its body, instead of a normal call. This results in faster code, but it adversely affects code size, particularly if the inline function is large and used often.

```
__inline int square(int x) {
   return x * x;
}

#include <MATH.H>

double length(int x, int y){
    return sqrt(square(x) + square(y));
}
```

There are several advantages to using inline functions:

- No function call overhead.

  As the code is substituted directly, there is no overhead, like saving and restoring registers.

- Lower argument evaluation overhead.

  The overhead of parameter passing is generally lower, since it is not necessary to copy variables. If some of the parameters are constants, the compiler can optimize the resulting code even further.

The big disadvantage of inline functions is that the code sizes increase if the function is used in many places. This can vary significantly depending on the size of the function, and the number of places where it is used.

It is wise to only inline a few critical functions. Note that when done wisely, inlining may decrease the size of the code: a call takes usually a few instructions, but the optimized version of the inlined code might translate to even less instructions.

The use of macros will do the same thing, and, since they arc resolved at compile time, they require absolutely no run-time overhead!
An example of inlining via macros:
old code:

```
int foo(a, b)
  {
    a = a - b;
    b++;
    a = a * b;
    return a;
  }
```

new code:

```
#define foo(a, b) (((a)-(b)) * ((b)+1))
```

The extra parentheses are necessary to preserve grouping in case foo is used in an expression with higher precedence than * or in case a and/or b contain subexpressions with lower precedence than + or -.

**Comma expressions and do { ... } while(0) can be used for more complicated functions, with some restrictions. The do-while macros let you create local variables for use in the macro, but you can't return a value to the expression the macro is used in. The opposite is true for macros using comma expressions.**

## 39. SEQUENTIAL ACCESS

Buffered I/O is usually (but not always) faster than unbuffered. Some gain can be wrought from using a larger than normal sized buffer; try out `setvbuf()`. If you aren't worried about portability you can try using lower level routines like `read()` and `write()` with large buffers and compare their performance to `fread()` and `fwrite()`. Using `read()` or `write()` in a single-character-at-a-time mode is especially slow on Unix machines because of the system call overhead.

Consider using `mmap()` if you have it. This can save effort in several ways. The data doesn't have to go through stdio which saves a buffer copy. Depending on the sophistication of the paging hardware, the data need not even be copied into user space; the program can just access an existing copy. `mmap()` also lends itself to read-ahead; theoretically the entire file could be read into memory before you even need it. Lastly, the file is can be paged directly off the source disk and doesn't have to use up virtual memory.

## 40. RANDOM ACCESS

Again, consider `mmap()` if you have it. If there's a trade off between I/O bound and memory bound in your program, consider a lazy-free of records: when memory gets tight, free unmodified records and write out modified records (to a temporary file if need be) and read them in later. Though if you take the disk space you'd use to write the records out and just add it to the paging space instead you'll save yourself a lot of hassle.

## 41. ASYNCHRONOUS I/O

You can set up a file descriptor to be non-blocking (see ioctl(2) or fcntl(2) man pages) and arrange to have it send your process a signal when the I/O is done; in the meantime your process can get something else done, including perhaps sending off other I/O requests to other devices. Significant parallelism may result at the cost of program complexity.

Multithreading packages can aid in the construction of programs which utilize asynchronous I/O.

## 42. TERMINALS

If your program spews out a lot of data to the screen, it's going to run slow on a 1200 baud line. Waiting for the screen to catch up stops your program. This doesn't add to the CPU or disk time as reported for accounting purposes, but it sure seems slow to the user. Bitmap displays are subject to a similar problem: xterms with jump scroll mode off can be quite slow at times.

A general solution is to provide a way for the user to squelch out irrelevant data. Screen handling utilities like curses can speed things up by reducing wasted screen updates.

## 42. SOCKETS

These have some weird properties. You may find that sending short, infrequent messages is extremely slow. This is because small messages may just sit in a buffer for a while waiting for the rest of the buffer to get filled up. There's some socket options you can set to get around this for TCP; or switch to an non-streaming protocol such as UDP. But the usual limitation is the network itself and how busy it is.

For the most part there's no free lunch and sending more data simply takes more time. However, a local network with a switched hub, bandwidth should have a clear theoretical upper limit. If you aren't getting anywhere near it, you might be able to blame the IP implementation being used with your OS. Try switching to a different "IP stack" and see if that helps a bit.

On Unix machines there are typically two socket libraries available, BSD sockets, and TLI. One is usually implemented in terms of the other and may suffer an extra buffer copy or other overhead. Try them both and see which is faster.