http://www.codeproject.com/Articles/6154/Writing-Efficient-C-and-C-Code-Optimization

- C code optimization, without looking at compiler options and architecture of the machine
- Concentrates on optimization of code speed and minimization of memory usage
- Often speeding up the program causes the source code size to increase,
  which in turn effects the code readability and simplicity

Start from **profiling** the code and localizing the methods (functions, routines, parts of the code) which takes **most time**.

Optimization of these places will have big impact on the code execution time.

How to do the profiling --> labs

Usually it is part of **inner** or **nested loop,** or **call** to some **external library methods**

Inner or nested loop --> we can optimize by recoding the program

external library method --> we can optimize by linking to more optimized library

1. Integers

Use

```
unsigned int variable name;
```

instead of

```
int variable name;
```

(when you know that the value is not negative)

```
register unsigned int variable name;
```

however it depends on the processor (unsigned int is smaller so less memory transfers are required)

`register` tells the compiler to store the variable at CPU register (if possible) for faster access

2. It is better to use integers instead of floating point numbers

- Integer operations can be done directly by processor
- floating point operations usually requires some external FPU (floating point unit) or floating point math library
- if you need accuracy of two decimal places 12.34, scale everyting *100,
  12.34*100=1234
  perform integer operations, and convert back to floats 1234*0.01 as late as possible

3. Divisions

What basic operation is most expensive?

In "standard processors" time of performing divisions for 32 bit floating point numbers takes 20-140 cycles to execute.

In particular the division 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)).
```

```
C0  = constant time
log2 (numerator / denominator) = log2 (numerator) - log2 (denominator)
plus C1 * (log2 (numerator) - log2 (denominator))
```

Avoid it at any cost!

Sometimes it can be rewritten as multiplication

```
if((a/b) > c)
-->
if(a>(b*c))
```

possible when b is positive a/b>c /*b --> a> b*c

b*c fits in an integer --> even better (why ? :)

4. Combining Division and Reminder

To get both dividend and reminder it is necessary to perform an expensive division operation

```
x/y <-- dividend
x%y <-- reminder
```

If both dividend and reminder are needed, combine them into one line so compiler calls the division only once

```
int func_div_and_mod (int a, int b) {
        return (a / b) + (a % b);
    }
```

5. Division and reminder by powers of two

Divisions and reminders by powers of two are faster, since compiler uses shifts to compute them.

Make them explicit --> `/ 32;`

```
typedef unsigned int uint;

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

Additionally, it is even faster if we know that the number is unsigned integer

Both division presented here will be execued by shifts, and unsigned integer version will be little faster

6. Module arithmetic

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

The above functon (routine) is expensive since to get the reminder we need to call an expensive division operation

We can rewrite it using if statement and avoiding the division

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

7. Using arrays instead of switch / if/else statements

Consider the switch pattern

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

or if / else pattern

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

It is much faster to do this (one access to an array instead of multi level if/else)

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

letter = classes[queue];
```

8. Global variables

- global variables are never allocated to registers --> slow access
  why? because they can be changed by assigning a pointer indirectly to them or by a function call

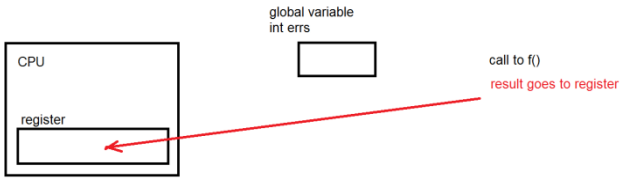
Each access needs one load from memory to register, one modification, and one store of the result

- global variables should not be used inside a heavy loop
- if a loop uses a global variable, it is beneficial to make a local copy (before the loop) so the local copy can be assigned to a register.
  This is only possible if the global variable is not used by any other methods / functions / routines called from the loop

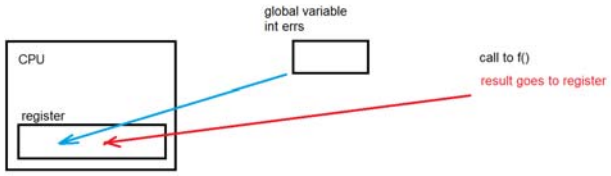
Example:

```
int f(void); // headers of the two functions called
int g(void); // we assume that they DO NOT modify the global variable
int errs; // <- global variable
void test1(void)
{
  //we have a two calls to functions, and the result is added to the global variable
  errs += f();
  errs += g();
}
```

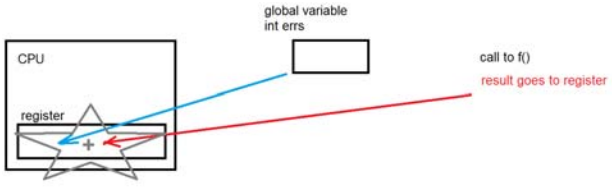
Call to f()



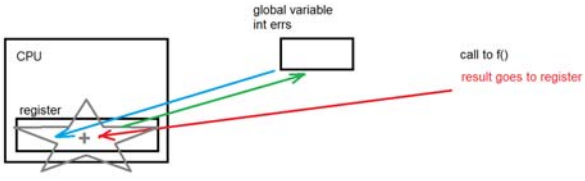
Result goes into register // 1 mops [memory transfer operation]
Actual value of errs is transferred from memory to register //1 mops



Addition //1 iops



Result is transferred back to global memory // 1 mops [integer operation]



Call to g()
Result goes into register // 1 mops

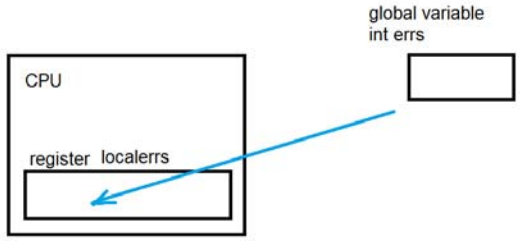
Actual value of errs is transferred from memory to register //1 mops
Addition //1 iops
Result is transferred back to global memory // 1 mops

Total: 6 mops, 2 iops

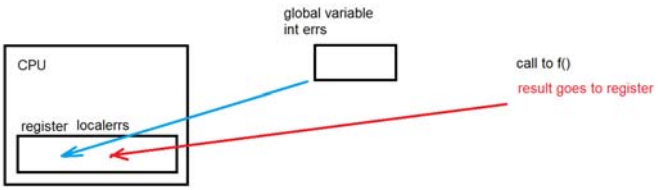
How to speed it up?

```
void test2(void)
{
  register int localerrs = errs; //make a local copy of global variable
  localerrs += f(); //make 3 calls that compute something and add to local variables
  localerrs += g();
  errs = localerrs; //store the result to the global variable
}
```

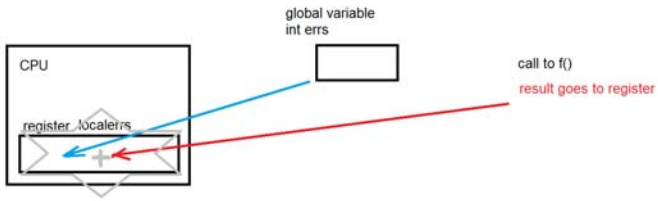
Transfer value of errs into register variable //1 mops



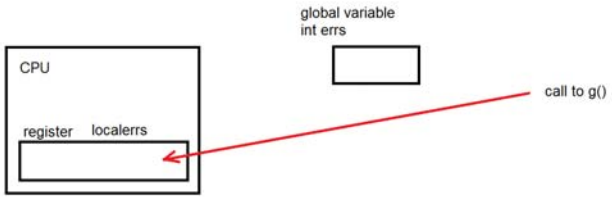
Call to f()
Result goes into register // 1 mops



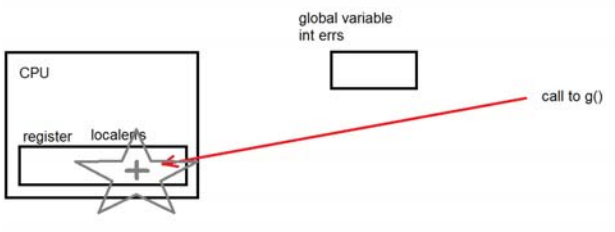
Addition //1 iops



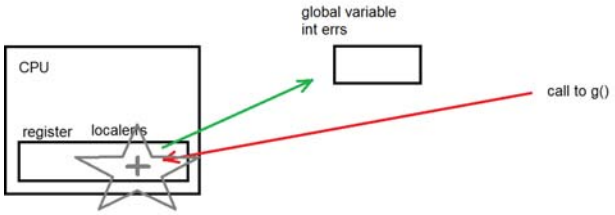
Call to g()
Result goes into register // 1 mops



Addition //1 iops



Result is transferred back to global memory // 1 mops



Total: 4 mops, 2 iops

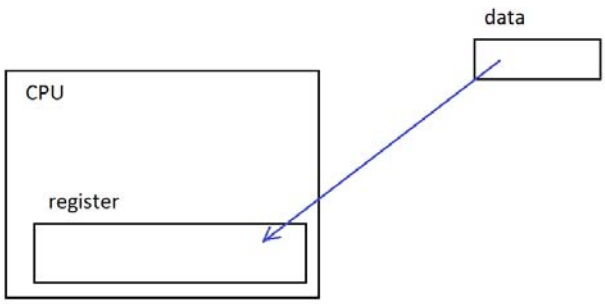
9. Using aliases

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

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

calls anyfunc

Value of data has to be read into register each time anyfunc is called



10 mops

Better solution:

Make a register local copy of data once, and then call anyfunc 10 times, each time using the value that may be already in the register

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

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

We can make a local copy first at the register, and if possible, compiler **may** keep localdata into register (however this is not guaranteed, but this implementation gives compiler better chance for optimization)

(adding register int localdata does not really guarantees the data will be kept in the register)

10. Live and dead variables

Live variable is a variable that is active in given moment of the code

```
    void example(int localvar1)
    {
        int localvar2;
        //here we have 2 live variables: localvar1, localvar2
        //here wa heve 2 dead variables: localvar3, localvar4
        for(i=0; i<10; i++)
        {
          int localvar3;
          //here we have 3 live variables: localvar1, localvar2, localvar3
          //here wa heve 1 dead variable: localvar4
          for(i=0; i<10; i++)
          {
              int localvar4;
              //here we have 4 live variables: localvar1, localvar2, localvar3, localvar4
              //here we have 0 local variables
          }
          //here we have 3 live variables: localvar1, localvar2, localvar3
          //here wa heve 1 dead variable: localvar4
        }
        //here we have 2 live variables: localvar1, localvar2
        //here wa heve 2 dead variables: localvar3, localvar4
    }
```

Compiler is trying to keep live variables in registry
However number of register variable is limited

--> it is reasonable to optimize number of live variables (minimize number of local variables, decompose difficult functions into many small functions)

--> using register for frequently used variables

11. Variables types

What is wrong with these codes?

```
short shortinc (short a)
    {
```

```
        return a + 1;
    }

char charinc (char a)
    {
        return a + 1;
    }
```

Avoid using **char** and **short** types

- after each operation compiler needs to reduce size of the local variable to 8 or 16 bites
- so-called *signed extending* for signed variables or *zero extending* for unsigned variables
- this is implemented by shifting register left by 16 or 24 bits

Use **int** or **unsigned int** instead

```
    int wordinc (int a)
    {
       return a + 1;
    }
```

This code will run much faster than previous one.

12. Pointers

What is wrong with this code:

```
    void pilot ()
    {
        CObject object;
        print_the_object(object)
    }

    void process_the_object (CObject object)
    {
        ...printf contents of the object...
    }
```

Do not pass objects to routines!

```
    void pilot ()
    {
        CObject object;
        print_the_object(&object)
    }

    void process_the_object (CObject* ptr_object)
    {
        ...printf contents of the object throught pointer ptr_object...
    }
```

Do not pass objects to routines!

You can use references:

```
    void pilot ()
    {
        CObject object;
        print_the_object(object)
```

```
    }

    void process_the_object (CObject& object_reference)
    {
        ...printf contents of the object throught reference object_referece...
    }
```

or pointers:

```
    void pilot ()
    {
        CObject object;
        print_the_object(object)
    }

    void process_the_object (CObject* ptr_object)
    {
        ...printf contents of the object throught pointer ptr_object...
    }
```

Additionally, functions that do not modify the object should get constant pointer

```
    void print_data_of_a_structure ( const Thestruct  *data_pointer)
    {
        ...printf contents of the structure...
    }
```

14. Pointer chains

```
    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;
    }
```

This requires to jump two times in memory (from p to pos, and from pos to x)
It is better to reduce number of jumps:

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

Another possibility:

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

void InitPos1(Object *p)
{
   Point3 p = p->pos;
   pos.x = 0;
   pos.y = 0;
   pos.z = 0;
}
```

Why it is better?

15. Boolean expressions and range checking

```
bool PointInRectangelArea (Point p, Rectangle *r)
{
   return (p.x >= r->xmin && p.x < r->xmax &&
                     p.y >= r->ymin && p.y < r->ymax);
}
```

r->xmin <-- read 1 time

r->ymax <-- read 1 time

r->ymin <-- read 1 time

r->ymax <-- read 1 time

p.x <-- read 2 times

p.y <-- read 2 times

6 mops plus 4 comparison, and 3 "and" boolean operations

```
bool PointInRectangelArea (Point p, Rectangle *r)
{
    return ((unsigned) (p.x - r->xmin) < r->xmax &&
   (unsigned) (p.y - r->ymin) < r->ymax);

}
```

r->xmin <-- read 1 time

r->ymax <-- read 1 time

r->ymin <-- read 1 time

r->ymax <-- read 1 time

p.x <-- read 1 times

p.y <-- read 1 times

6 mops plus 2 subtractions, 2 convertions, 2 comparison, and 1 "and" boolean operation

Why it is better?

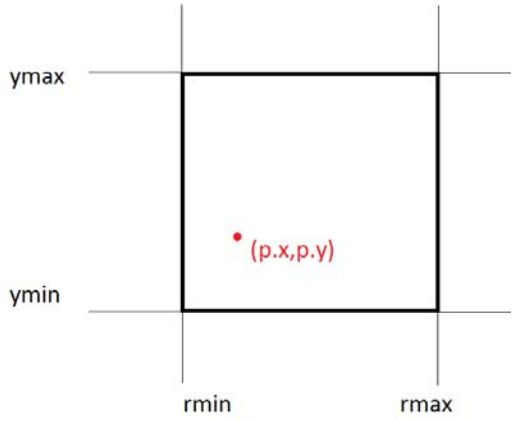
16. Boolean expressions and range checking

```
bool PointInRectangelArea (Point p, Rectangle *r)
    {
       return (p.x >= r->xmin && p.x < r->xmax &&
                          p.y >= r->ymin && p.y < r->ymax);
    }
```

We want to check if

p.x is in the range of [r->xmin,r->xmax]

py is in the range of [r->ymin,r->ymax]



Instead of

`(x >= min && x < max)`

it is better to do

`(unsigned)(x-min) < (max-min)`

Explanation:

x<max / - min

x-min<max-min

x>=min --> x-min >= 0

so

max-min > x-min >= 0

`(unsigned)(x-min) < (max-min)`

```
bool PointInRectangelArea (Point p, Rectangle *r)
    {
    return ((unsigned) (p.x - r->xmin) < r->xmax &&
            (unsigned) (p.y - r->ymin) < r->ymax);
    }
```

17. Boolean expressions and range checking

???

18. Lazy evaluation exploitation

```
if(a>10 && b=4)
```

Make sure that a>10 happens more frequently then b=4

since the second expression is not going to be evaluated if the first one is true

18. Switch instead of if ... else

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

It is faster to utilize the switch concept:

```
    switch( val )
    {
        case 1: dostuff1(); break;
        case 2: dostuff2(); break;
        case 3: dostuff3(); break;
    }
```

If we have

```
    if( val == 1)
        dostuff1();
    else if (val == 2)
        dostuff2();
    else if (val == 3)
        dostuff3();
    else
        dostuff4();
```

In this case **all the if statements will be tested always**.

Here the **testing is cut** when we reach the **actual value of** *val*

```
    switch( val )
    {
        case 1: dostuff1(); break;
        case 2: dostuff2(); break;
        case 3: dostuff3(); break;
        default: dostuff4(); break;
    }
```

18. Binary breakdown

If you need if ... else, than you can do it in binary tree manner.

Instead 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) {
    }
```

You can use:

```
   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 */
            }
        }
    }
```

19. Comparison to symbols

```
        c=getch();
        switch(c){
            case 'A':
            {
                do something;
                break;
            }
            case 'H':
            {
                do something;
                break;
            }
            case 'Z':
            {
                do something;
                break;
            }
        }
```

It is much faster to compare ASCI codes than symbols

```
c=getch();
        switch(c){
            case 0:
            {
                do something;
                break;
            }
            case 1:
```

```
            {
                do something;
                break;
            }
            case 2:
            {
                do something;
                break;
            }
        }
```

20. Switch statements vs lookup symbols

```
char * Condition_String1(int condition) {
      switch(condition) {
         case 0: return "EQ";
         case 1: return "NE";
         case 2: return "CS";
         case 3: return "CC";
         case 4: return "MI";
         case 5: return "PL";
         case 6: return "VS";
         case 7: return "VC";
         case 8: return "HI";
         case 9: return "LS";
         case 10: return "GE";
         case 11: return "LT";
         case 12: return "GT";
         case 13: return "LE";
         case 14: return "";
         default: return 0;
      }
    }
```

It is much faster and efficient to use lookup table:

```
char * Condition_String2(int condition) {
       if ((unsigned) condition >= 15) return 0;
          return
          "EQ\0NE\0CS\0CC\0MI\0PL\0VS\0VC\0HI\0LS\0GE\0LT\0GT\0LE\0\0" +
           3 * condition;
    }
```

The first routine takes 240 bytes, the second one 72 bytes.

The first routine checks many case statements, the second one just computes the index in the lookup table.

20. Loop termination

Checking loop termination condition inside the loop is very expensive

Example 1

It is much faster to compare to 0 then to evaluate some logical condition

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

Here we compare i to n many times

This is much faster because we only compare to zero

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

Example 2

Recall:

```
for( Initial; Stopping condition; Increment operation){ ... }
```

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

Here we compare i to 10 11 times

i loops 0,1,2,3,4,5,6,7,8,9

It is much faster just to compare i to zero, which goes like this:

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

Here i loops 9,8,7,6,5,4,3,2,1,0

i-- return the number = value of i, and this value is used to determine the stopping condition for the loop

The increment operation is optional

Example 3

```
for( i=10;  i>0;  i--){ ... }
```
This can be replaced by

```
for(i=10; i; i--){}
```
Loops stop at 0, so we do not need to check if i>0

21. Loop jamming

If you have two loops with identical counters, maybe it is possible to make one out of them

```
//Original Code :


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

        for(i=0; i<100; i++){
            morestuff();
        }
```
And *morestuff()* can be processed independently on *stuff()*

then we can jam the loops:

```
        //It would be better to do:
        for(i=0; i<100; i++){
            stuff();
            morestuff();
        }
```

22. Function looping

```
for(i=0 ; i<100 ; i++)
    {
        func(t,i);
    }
    -
    -
    -
    void func(int w,d)
    {
        lots of stuff.
    }
```
We have an expensive function that depends on integers t and i

We loop through i and call the function many times.

Then, it is better to move the loop inside the function

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

```
        }
    }
```

23. Loop unrolling

Example 1

If the loop is small

```
for(i=0; i<3; i++){
            something(i);
        }
```

it is much faster to unroll the loop:

```
something(0);
something(1);
something(2);
```

This is because here we do not have to deal with initializing, incrementing, and checking the value fo the counter

Example 2

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

The general loop cannot be directly unrolled, but we can unroll the loops in blocks.

This example shows how can we unroll the loop in blocks of 8:

```
    #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);
        }
    }

    }
```

Example 3

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

We count 1's in binary representation of unsigned integer n

We still can unroll the loop in blocks of e.g.4

```
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;
    }
```

23. Early loop breaking

This is really trivial observation.
Sometimes it is not necessary to go through entire loop

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

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

It is better to break the loop when we find -99

```
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");
```

24. Function call overhead

The number of parameters that can be passed to a function through registry is limited

These arguments can be integer-compatible (char, shorts, ints and floats all take one word), or structures of up to four words (including the 2-word doubles and long longs).

It may be up to 4 parameters (depending on the processor)

Also:

This works much faster

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

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

then this one:

```
    double f2(double a, double b, double c, double d, double e, double f) {
```

```
  return a + b + c + d + e + f;
}

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

- **Try to ensure that small functions take four or fewer arguments. These will not use the stack for argument passing.**
  This works faster

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

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

then this one:

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

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

since in the first one we pass 4 parameters, in the second one we pass 6 parameters.

- **If a function needs more than four arguments, try to ensure that it does a significant amount of work, so that the cost of passing the stacked arguments is outweighed.**

```
int f2(int a, int b, int c, int d, int e, int f) {
  SIGNIFICANT AMOUNT OF WORK!!!
}

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

- **Pass pointers to structures instead of passing the structure itself.**

```
void pilot ()
{
    CObject object;
    print_the_object(object)
}

void process_the_object (CObject object)
{
    ...printf contents of the object...
}
```

Do not pass objects to routines!

```
void pilot ()
{
```

```
    CObject object;
    print_the_object(object)
}

void process_the_object (CObject* ptr_object)
{
    ...printf contents of the object throught pointer ptr_object...
}
```

- **Put related arguments in a structure, and pass a pointer to the structure to functions. This will reduce the number of parameters and increase readability.**

```
class CObject {
    double m_a;
    double m_b;
    double m_c;
    double m_d;
    int m_n;
    ...
}
void pilot ()
{
    CObject object;
    object.m_a=1.0;
    object.m_b=2.0;
    object.m_c=3.0;
    object.m_d=3.0;
    object.m_n=4;
    ...
    print_the_object(object)
}

void process_the_object (CObject* ptr_object)
{
    ...printf contents of the object throught pointer ptr_object...
}
```

- **Minimize the number of long parameters, as these take two argument words.**
  This works faster

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

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

then this one:

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

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

- **This also applies to doubles if software floating-point is enabled.**
  This works faster

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

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

then this one:

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

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

- **Avoid functions with a variable number of parameters. Those functions effectively pass all their arguments on the stack.**

25. General remarks

- Avoid using ++ and -- etc. within loop expressions. E.g.: while(n--){}, as this can sometimes be harder to optimize.
- Minimize the use of global variables.
- Declare anything within a file (external to functions) as static, unless it is intended to be global.
- Use word-size variables if you can, as the machine can work with these better (instead of char, short, double, bit fields etc.).
- Don't use recursion. Recursion can be very elegant and neat, but creates many more function calls which can become a large overhead.
- Avoid the sqrt() square root function in loops - calculating square roots is very CPU intensive.
- Single dimension arrays are faster than multi-dimension arrays.
- Compilers can often optimize a whole file - avoid splitting off closely related functions into separate files, the compiler will do better if it can see both of them together (it might be able to inline the code, for example).
- Single precision math may be faster than double precision - there is often a compiler switch for this.
- Floating point multiplication is often faster than division - use val * 0.5 instead of val / 2.0.
- Addition is quicker than multiplication - use val + val + val instead of val * 3. puts() is quicker than printf(), although less flexible.

- Use #defined macros instead of commonly used tiny functions - sometimes the bulk of CPU usage can be tracked down to a small external function being called thousands of times in a tight loop. Replacing it with a macro to perform the same job will remove the overhead of all those function calls, and allow the compiler to be more aggressive in its optimization..
- Binary/unformatted file access is faster than formatted access, as the machine does not have to convert between human-readable ASCII and machine-readable binary. If you don't actually need to read the data in a file yourself, consider making it a binary file.
- If your library supports the mallopt() function (for controlling malloc), use it. The MAXFAST setting can make significant improvements to code that does a lot of malloc work. If a particular structure is created/destroyed many times a second, try setting the mallopt options to work best with that size.