Computing Laboratory

Programming Style, Efficient Programming, Code Optimization

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
Indian Statistical Institute, Kolkata

January, 2022

Indentation style

- Brace placement in compound statements
- Tabs, spaces, and size of indentations

Indentation style

- Brace placement in compound statements
- Tabs, spaces, and size of indentations

```
unsigned char i = 0;
for(;i<=0;i++);
printf("%d\n",i);
```

Indentation style

- Brace placement in compound statements
- Tabs, spaces, and size of indentations

```
unsigned char i = 0;
for(;i<=0;i++);
printf("%d\n",i);
```

It should be written as follows.

```
unsigned char i;
for(i=0 ; i<=0 ; i++)
    ;
printf("%d\n",i);
```

Note: Refer to B. W. Kernighan and P. J. Plauger, The Elements of Programming Style, McGraw-Hill, New York

Internal/External documentation

- "Block comments" should be placed at the head of every subprogram detailing the purpose of the subprogram, list of all parameters, direction of data transfer (into this routine, out from the routine back to the calling routine, or both), and their purposes.
- Meaningful variable names.
- A brief comment next to the declaration of each variable and constant.
- Embedded comments for every complex process.

External documentation includes what the code does, who wrote it and when, which common algorithms it uses, library dependencies, which systems it was designed to work with, what form and source of input it requires, the format of the output it produces, etc.

Variable and function names

Required properties:

- They must start with a letter or underscore (_)
- They can contain only letters, underscores, digits
- They cannot match *reserved* words
- Variables and functions are case-sensitive

Variable and function names

Required properties:

- They must start with a letter or underscore (_)
- They can contain only letters, underscores, digits
- They cannot match *reserved* words
- Variables and functions are case-sensitive

Recommended properties:

- Use "meaningful" names
- Use `under_scores` or `CamelCase` for long names
- Hungarian notation (much debated!)

Using relational operators on constants

It is a good programming practice to place the constants on the left of relational operators and variables on the right.

```
int var = 0;
if(0 == var) // Not ‘var == 0’
    printf("Equal");
else
    printf("Not equal");
```

Using relational operators on constants

It is a good programming practice to place the constants on the left of relational operators and variables on the right.

```
int var = 0;
if(0 == var) // Not ‘var == 0’
    printf("Equal");
else
    printf("Not equal");
```

- If ‘0 = var’ is written (by mistake) in place of ‘0 == var’ it will be detected as an error (l-value required).
- If ‘var = 0’ is written (by mistake) in place of ‘var == 0’ it will remain undetected.

Avoid equality checking with floating variables

Floating variables approximate the values they store. Hence, what you expect is not what you actually get.

Example:

```
float f = 0.8;
if(0.8 == f)
    printf ("No entry");
else
    printf("Entry");
```

Avoid equality checking with floating variables

Floating variables approximate the values they store. Hence, what you expect is not what you actually get.

Example:

```
float f = 0.8;
if(0.8 == f)
    printf ("No entry");
else
    printf("Entry");
```

Output:

```
Entry
```

Suggestions

1. Write clearly without being clever.
2. Use library functions whenever feasible.
3. Avoid too many temporary variables.
4. Parenthesize to avoid ambiguity.
5. Avoid unnecessary branches.
6. Choose a data representation that makes the program simple.
7. Modularize using procedures and functions.
8. Completely avoid the use of `goto`.
9. Use recursive procedures for recursively-defined data structures.
10. Use self-identifying input. Allow defaults. Echo both on output.
11. Test programs at their boundary values.

Suggestions

12. Terminate input by end-of-file marker, not by count.
13. Make sure all variables are initialized before use.
14. Use debugging compilers.
15. Take care to branch the right way on equality.
16. Be careful if a loop exits to the same place from the middle and the bottom.
17. 10.0 times 0.1 is hardly ever 1.0.
18. 5/7 is zero but 5.0/7.0 is not zero.
19. Make it right before you make it faster.
20. Make it fail-safe before you make it faster.
21. Let your compiler do the simple optimizations.
22. Instrument your programs. Measure before making efficiency changes.
23. Do not over-comment

Local and global variables

- Local variables: Variables defined within a function (or block).
  – Stored in a region of memory called an activation record
- Global variables: Variables defined outside of the body of any function.
  – Stored in the data segment

Where are the activation records (AR) stored?

- Simple solution: AR == one fixed block of memory per function
- Better solution: AR allocated/deallocated when function is called/returns
  – Variables created when function is called; destroyed when function returns
  – Need to keep track of nested calls
  – Function calls behave in last in first out manner (use stack to keep track of ARs)

Static variables

Static variables are defined within a function, but not destroyed when function returns, i.e., retains value across calls to the same function.

Static variables

Static variables are defined within a function, but not destroyed when function returns, i.e., retains value across calls to the same function.

Example:

```
void f(void){
    static int i = 1;
    printf("This function has executed %d time(s)\n",i);
    i++;
}
```

Storage classes

	Automatic	**Register**	**Static**	**External**
Keyword	auto	register	static	extern
Storage	Memory	CPU Register	Memory	Memory
Default Value	Garbage value	Garbage value	Zero	Zero
Scope	Local to the block	Local to the block	Local to the block	Global
Life	Ends out of the block	Ends out of the block	Stays dormant out of the block	Ends out of the program

Storage classes

	Automatic	**Register**	**Static**	**External**
Keyword	auto	register	static	extern
Storage	Memory	CPU Register	Memory	Memory
Default Value	Garbage value	Garbage value	Zero	Zero
Scope	Local to the block	Local to the block	Local to the block	Global
Life	Ends out of the block	Ends out of the block	Stays dormant out of the block	Ends out of the program

Note: If unspecified, compiler will assume a storage class of a variable depending on the context in which it is used.

The appropriate use of storage classes

The best declaration for an integer variable in a tight loop would be as follows.

```
register unsigned int <variable_name>;
```

The appropriate use of storage classes

The best declaration for an integer variable in a tight loop would be as follows.

```
register unsigned int <variable_name>;
```

- A variable will get placed into CPU registers if it is available. Otherwise, it will be placed into memory.
- Typically a CPU register is less than 64 bits in size. Hence, variables requiring larger bits cannot accommodated in CPU registers.

Initializing a large 2-D array

Instead of assigning values to a large 2-D array defined within a program, we can read the values from an external file.

Example:

```
double MATRIX[SIZE][SIZE] = {
    #include "VALUES.txt"
};
```

Note: The preprocessor directive must start in a separate line.

Lazy evaluation

Conditionals/Boolean expressions in C are evaluated from left to right. Evaluation stops as soon as the value of the expression is known. Remaining sub-expressions are not evaluated.

Lazy evaluation

Conditionals/Boolean expressions in C are evaluated from left to right. Evaluation stops as soon as the value of the expression is known. Remaining sub-expressions are not evaluated.

Examples:

1. `(x > y) && (a != b)`: If x is less than y, then the expression is FALSE, irrespective of the value of the second sub-expression
2. `(n > 0) || (i == j)`: If n is greater than 0, the expression is TRUE, irrespective of the value of the second sub-expression

Lazy evaluation

Typical usage:

```
while(i < N && A[i] >= 0){
    ...
```

- If $i >= N$, $A[i]$ is not checked.
- This is useful because checking $A[i] >= 0$ when $i >= N$ may lead to memory faults.
- In such expressions, $i >= N$ serves as a guard condition.

Avoiding division

In standard processors, divisions are time-consuming because they take a constant time plus a time for each bit to divide.

```
if((a / b) > c)
    printf("More");
```

Avoiding division

In standard processors, divisions are time-consuming because they take a constant time plus a time for each bit to divide.

```
if((a / b) > c)
    printf("More");
```

It can be efficiently written as follows:

```
if(a > (b * c))
    printf("More");
```

Here, the only assumptions are b is non-negative and $b * c$ fits into an integer. The latter one is also safe if $b = 0$.

Combining division and remainder

To compute the sum of digits, both dividend (a / b) and remainder (a % b) are needed to be derived. In this case, the compiler can combine both by calling the division function once returning both dividend and remainder.

Example:

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

Using array indices

To set a variable to a particular character, depending upon the value of something, one might do this:

```
if(index == 0)
    letter = ‘D’;
else if(index == 1)
    letter = ‘F’;
else
    letter = ‘S’;
```

Using array indices

To set a variable to a particular character, depending upon the value of something, one might do this:

```
if(index == 0)
    letter = ‘D’;
else if(index == 1)
    letter = ‘F’;
else
    letter = ‘S’;
```

A faster approach is to simply use the value as an index into a character array, e.g.:

```
static char *classes="DFS";
letter = classes[index];
```

Measuring time

- The amount of (real) time taken to execute a part of the program is measurable
- Relevant headers, types, system calls (e.g., gettimeofday())

Measuring time

- The amount of (real) time taken to execute a part of the program is measurable
- Relevant headers, types, system calls (e.g., gettimeofday())

The definition:

```
#include<sys/time.h>
struct timeval{
    __time_t tv_sec;        /* seconds */
    __suseconds_t tv_usec; /* microseconds */
};

int gettimeofday(struct timeval *tv, struct timezone *tz);
```

Note: gettimeofday() stores the number of seconds and microseconds elapsed since the "Epoch" (00:00 of 01.01.1970).

Measuring time

Auxilliary macro:

```
#define DURATION(start, end) ((end.tv_usec - start.tv_usec) +
(end.tv_sec - start.tv_sec) * 1000000)
```

Usage:

```
struct timeval start_time, end_time;
/* Store the starting time */
if(gettimeofday(&start_time, NULL) == -1)
ERR_MESG("gettimeofday failed");

/* The code whose running time is to be measured */

/* Store the ending time */
if(gettimeofday(&end_time, NULL) == -1)
    ERR_MESG("gettimeofday failed");

/* Compute the number of microsecs between start and end */
printf("%d\n", (int) DURATION(start_time, end_time));
```

Lessons

1. Use the most appropriate data type for variables, as it reduces code and data size and increases performance considerably.
2. It is best to avoid using `char` and `short` as local variables. For them, the compiler needs to do sign-extending for signed variables and zero extending for unsigned variables.
3. Use registers for keeping frequently-used variables.
4. Global variables are never allocated to registers. So, we should not use them inside critical loops.
5. If possible, pass structures by reference (using a pointer to the structure), otherwise the whole thing will be copied onto the stack and passed, which will slow things down.
6. Reading chunk of characters at a time from a file is faster that reading character by character.

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.

```
unsigned int mod_func(unsigned int counter){
    return (++counter % 100);
}
```

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.

```
unsigned int mod_func(unsigned int counter){
    return (++counter % 100);
}
```

It should be written as follows.

```
unsigned int mod_func(unsigned int counter){
    if(++counter >= 100)
        counter = 0;
    return (counter);
}
```

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.

```
unsigned int mod_func(unsigned int counter){
    return (++counter % 100);
}
```

It should be written as follows.

```
unsigned int mod_func(unsigned int counter){
    if(++counter >= 100)
        counter = 0;
    return (counter);
}
```

The latter one is preferable because it is much faster. However, it works only if the range of count on input is 0-99.

Optimizing the use of operators

Use the right ($>>$) and left ($<<$) shift operations instead of integer multiplication and division, wherever respectively possible. Bit level operations are much faster.

For basic data types, use the operators $+$, $-$, $*$, and $/$ instead of $+=$, $-=$, $*=$, and $/=$, respectively.

Efficient use of bitwise operators – Multiplication

Try to avoid arithmetic multiplication as and when possible.

```
int m;
printf("%d", m * 17);
```

Efficient use of bitwise operators – Multiplication

Try to avoid arithmetic multiplication as and when possible.

```
int m;
printf("%d", m * 17);
```

It should be written using *bitwise left shift* as follows.

```
int m;
printf("%d", (m << 4) + m);
```

Note: `m * n` will return the same result as that of `(m << p) + m`, where `n` can be represented as $2^p + 1$.

Efficient use of bitwise operators – Power of 2

We can test whether a number is a power of 2 with bitwise AND as follows:

```
int m;
scanf("%d", &m);
if((m & (m - 1)) == 0)
    printf("%d is a power of 2", m);
else
    printf("%d is not a power of 2", m);
```

Efficient use of bitwise operators – Sign match

We can test whether a pair of integers have the same sign with *bitwise XOR* as follows:

```
int m, n;
scanf("%d%d", &m, &n);
if((m ^ n) < 0)
    printf("%d and %d have different signs", m, n);
else
    printf("%d and %d have the same sign", m, n);
```

Efficient use of bitwise operators – Increment

Note that, x + (2's complement of x) = 0
$\implies$ x + ((1's complement of x) + 1) = 0
$\implies$ $x + 1 = -$ (1's complement of x).

Hence, We can use the following representation to increment the value of a variable x by 1:

```
-~x
```

Efficient use of bitwise operators – Modular division

Never perform modular division with a power of 2.

```
int m;
printf("%d", m % 8);
```

Efficient use of bitwise operators – Modular division

Never perform modular division with a power of 2.

```
int m;
printf("%d", m % 8);
```

It should be written using *bitwise AND* as follows.

```
int m;
printf("%d", m & 7);
```

Note: `m % n` will return the same result as that of `m & (n-1)`, where `n` is a power of 2.

Efficient use of bitwise operators – Checking alphabets

Never perform case insensitive alphabet checking as follows.

```
if(ch == 'a' || ch == 'A') vowel++;
if(ch == 'e' || ch == 'E') vowel++;
if(ch == 'i' || ch == 'I') vowel++;
if(ch == 'o' || ch == 'O') vowel++;
if(ch == 'u' || ch == 'U') vowel++;
```

Efficient use of bitwise operators – Checking alphabets

Never perform case insensitive alphabet checking as follows.

```
if(ch == 'a' || ch == 'A') vowel++;
if(ch == 'e' || ch == 'E') vowel++;
if(ch == 'i' || ch == 'I') vowel++;
if(ch == 'o' || ch == 'O') vowel++;
if(ch == 'u' || ch == 'U') vowel++;
```

It can be made faster using *bitwise OR* as follows.

```
ch = ch | 0x20;
if(ch == 'a') vowel++;
if(ch == 'e') vowel++;
if(ch == 'i') vowel++;
if(ch == 'o') vowel++;
if(ch == 'u') vowel++;
```

Optimizing mathematical operations – Avoiding division

Instead of repeatedly dividing by x, compute 1/x and multiply accordingly. It is really beneficial if you do more than 3 divides.

Example:

```
int i, n = 10;
float x = 0.5, result = 1.0;
for(i = 0; i < n; i++)
    result /= x;
```

This can be efficiently done in the following alternative way:

```
int i, n = 10;
float x = 0.5, result = 1.0;
x = result / x;
for(i = 0; i < n; i++)
    result = result * x;
```

Optimizing mathematical operations – Avoiding `sqrt()`

The function `sqrt()` can often be avoided, especially in comparisons where comparing the value squared gives the same result.

In many equations, terms cancel out, either always or in some special cases. Work on them before writing the program.

Optimizing mathematical operations – Avoiding pow()

Avoid the use of pow() for computing small integer powers.

```
int m;
printf("%d", (int)pow(m, 3.0));
```

Optimizing mathematical operations – Avoiding pow()

Avoid the use of pow() for computing small integer powers.

```
int m;
printf("%d", (int)pow(m, 3.0));
```

It should be written as follows.

```
int m;
printf("%d", m * m * m);
```

Type-specific mathematical operations

On modern CPUs, floating-point operations have essentially the same throughput as integer operations. In compute-intensive programs, this leads to a negligible difference between integer and floating-point costs. So, its better to stick to the original data type.

Double precision floating-point operations may not be slower than single precision floats, particularly on 64-bit machines.

Type casting

Avoid `type casting` wherever possible. Integer and floating point instructions often operate on different registers, so a casting requires a copy.

Example:

```
float i = 2.7;
printf("i = %0.1f", (int)i); // Prints i = 2.0
printf("i = %0.1f", i); // Prints i = 2.7
```

Type casting

Avoid `type casting` wherever possible. Integer and floating point instructions often operate on different registers, so a casting requires a copy.

Example:

```
float i = 2.7;
printf("i = %0.1f", (int)i); // Prints i = 2.0
printf("i = %0.1f", i); // Prints i = 2.7
```

Shorter integer types (char and short) still require the use of a full-sized register, and they need to be padded to 32/64-bits and then converted back to the smaller size before storing back in memory. (However, this cost must be weighed against the additional memory cost of a larger data type.)

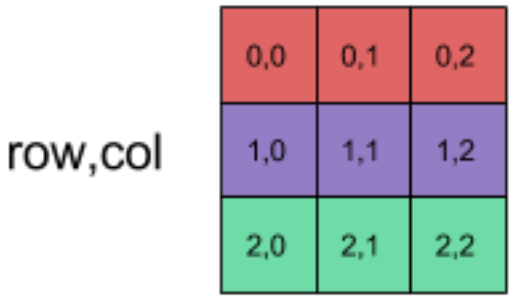
Memory organization in arrays

Two and higher dimensional arrays are still stored in one dimensional memory. This means `ARR[i][j]` and `ARR[i][j+1]` are adjacent to each other, whereas `ARR[i][j]` and `ARR[i+1][j]` may be arbitrarily far apart.

When modern CPUs load data from main memory into processor cache, they fetch more than a single value. Instead they fetch a block of memory containing the requested data and adjacent data (a cache line). This means after `ARR[i][j]` is in the CPU cache, `ARR[i][j+1]` has a good chance of already being in cache, whereas `ARR[i+1][j]` is still likely to be in the main memory.

Memory organization in arrays

Accessing data in a more-or-less sequential fashion, as stored in physical memory (in row major fashion), can dramatically speed up the code.



Dynamic memory allocation

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

As a matter of fact, 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.

Loop jamming

Never use two loops where one will suffice.

```
for(i=0; i<10; i++)
    <Statement 1>
for(i=0; i<10; i++)
    <Statement 2>
```

Loop jamming

Never use two loops where one will suffice.

```
for(i=0; i<10; i++)
    <Statement 1>
for(i=0; i<10; i++)
    <Statement 2>
```

It should be written as follows:

```
for(i=0; i<10; i++){
    <Statement 1>
    <Statement 2>
}
```

Note: If a single loop contains a lot of operations (it might not fit into the processor's instruction cache), then two separate loops may be faster than a single combined one.

Loop unrolling

The loop overhead can be reduced by decreasing the number of iterations and replicating the body of the loop.

```
for(i=0; i<100; i++)
    <Statement>
```

Loop unrolling

The loop overhead can be reduced by decreasing the number of iterations and replicating the body of the loop.

```
for(i=0; i<100; i++)
    <Statement>
```

The above code can be written as follows:

```
for(i=0; i<100; i+=2){
    <Statement>
    <Statement>
}
```

Note: The need for condition check adds some overhead to the program.

Loop inversion

We should always write count-down-to-zero loops and use simple termination conditions for a better efficiency.

Loop inversion

We should always write count-down-to-zero loops and use simple termination conditions for a better efficiency.

```
int factorial1(int n){
    int i, fact = 1;
    for(i = 1; i <= n; i++)
        fact *= i;
    return (fact);
}
```

Loop inversion

We should always write count-down-to-zero loops and use simple termination conditions for a better efficiency.

```
int factorial1(int n){
    int i, fact = 1;
    for(i = 1; i <= n; i++)
        fact *= i;
    return (fact);
}
```

The above code can be made efficient by writing as follows:

```
factorial2(int n){
    int i, fact = 1;
    for(i = n; i > 0; i--)
        fact *= i;
    return (fact);
}
```

Loop inversion

The previous code can be improved as follows:

```
factorial3(int n){
    int i, fact = 1;
    for(i = n; i; i--)
        fact *= i;
    return (fact);
}
```

Loop inversion

The previous code can be improved as follows:

```
factorial3(int n){
    int i, fact = 1;
    for(i = n; i; i--)
        fact *= i;
    return (fact);
}
```

The above code can be further optimized as follows:

```
factorial4(int n){
    int i, fact = 1;
    for(i = n+1; --i;)
        fact *= i;
    return (fact);
}
```

Function calls

Move loops inside function calls. Replace the following code

```
for(i=0;i<n;i++){
    Function();
}
```

with this code

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

Note: Jumps/branches are expensive and hence should be avoided whenever possible. Note that, function calls require two jumps, in addition to stack memory manipulation.

Inline functions

Use inline functions for replacing short functions to eliminate the function overhead. Hence, the following function

```
int Function(x, y){
    x = x - y;
    y++;
    x = x * y;
    return x;
}
```

can be effectively written as follows:

```
#define Function(x, y) (((x)-(y)) * ((y)+1))
```

Lessons

1. Prefer iteration over recursion.
2. Long `if...else if...else if...` chains require lots of jumps for cases near the end of the chain (in addition to testing each condition). If possible, convert to a `switch` statement, which the compiler sometimes optimizes into a table lookup with a single jump. If a `switch` statement is not possible, put the most common clauses at the beginning of the if chain.
3. If you do not need a `return` value from a function, do not define one.
4. Use optimized library functions like `mallopt()` instead of `malloc()`.

Let us practice!!!

Write an optimized C program that takes an integer n from stdin and prints the first n elements (separated by comma) of Fibonacci series on stdout. Recall that Fibonacci series appears as:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...$$

Measure and print the time taken by the program.

Let us practice!!!

Standard recursive implementation:

```
int i = 0;
for(i = 0 ; i <= n ; i++){
    printf("%d, ", Fibonacci_Recur(i));
}

int Fibonacci_Recur(int n){     // n is user input
    if(n <= 1)
       return n;
    if(n >= 2)
       return Fibonacci_Recur(n-1) + Fibonacci_Recur(n-2);
}
```

Let us practice!!!

Standard iterative implementation:

```
int i, n1 = 0, n2 = 1, n3, n;   // n is user input
printf("%d, %d, ", n1, n2);
for(i = 0 ; i <= n-2 ; i++){
    n1 = n2;
    n2 = n3;
    n3 = n1 + n2;
    printf("%d, ", n3);
}
```

Let us practice!!!

An optimized implementation:

```
int n1 = 0, n2 = 1, n;              // n is user input
printf("%d, %d, ", n1, n2);
for(; n >> 1 ; n -= 2){
    printf("%d, %d, ", n1, n2);
    n1 = n1 + n2;
    n2 = n1 + n2;
}
if(n & 1)
    printf("%d", n1);
```