Data and File Structures Laboratory

Programming Style, Efficient Programming

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
Indian Statistical Institute, Kolkata

August, 2018

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);

...
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 names

Required properties:

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

Variable names

Required properties:

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

Example:

```
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.
Example:

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

- If '0 = var' is used (by mistake) in place of '0 == var' it will be detected as an error.
- If 'var = 0' is used (by mistake) in place of 'var == 0' it will remain undetected.

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. Do not compare floating point numbers solely for equality.
20. Make it right before you make it faster.
21. Make it fail-safe before you make it faster.
22. Let your compiler do the simple optimizations.
23. Instrument your programs. Measure before making efficiency changes.
24. 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

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

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){
     ...
}
```

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){
    ...
}
```

It can be efficiently written as follows:

```
if(a > (b * c)){
    ...
}
```

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

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

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 mod_func1(uint counter){
    return (++counter % 100);
}
...
uint mod_func2(uint 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. Consider the following two examples:

```
uint mod_func1(uint counter){
    return (++counter % 100);
}
...
uint mod_func2(uint counter){
    if(++counter >= 100)
        counter = 0;
    return (counter);
}
```

The use of the if statement, rather than the remainder operator, is preferable, as it is much faster. However, it works only if the range of count on input is 0-99.

Loop termination

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

Loop termination

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

```
int factorial1(int n){ // Slower code
    int i, fact = 1;
    for (i = 1; i <= n; i++)
        fact *= i;
    return (fact);
}
...
int factorial2(int n){ // Faster code
    int i, fact = 1;
    for (i = n; i != 0; i--)
        fact *= i;
    return (fact);
}
```

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.

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:

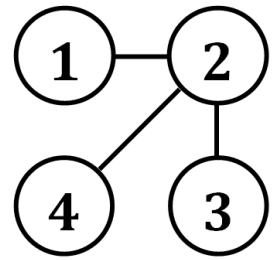
```
struct timeval start_time, end_time;
/* Store the starting time */
if(gettimeofday(&start_time, NULL) == -1)
ERR_MESG("gettimeofday failed");

/* 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. For large decisions involving `if...else...else...`, it may be faster to use a `switch`.
6. 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.
7. Reading chunk of characters at a time from a file is faster that reading character by character.

Problems – Day 6

1. Given the adjacency matrix of a graph, having a dimension of 10000×10000, find out the nodes that can be reached in k hops from each node. Report the time taken by the program.



hop_count(1,4) = 2

	1	2	3	4
1	0	1	0	0
2	1	0	1	1
3	0	1	0	0
4	0	1	0	0

2. Read the Wikipedia page titled "Indentation style", reverse its contents, and write back to the same file. Report the time taken by the program.