Personal Coding Conventions

Author: Adomas Vensas

Version: 1.0

Date of first release: 2022-01-09

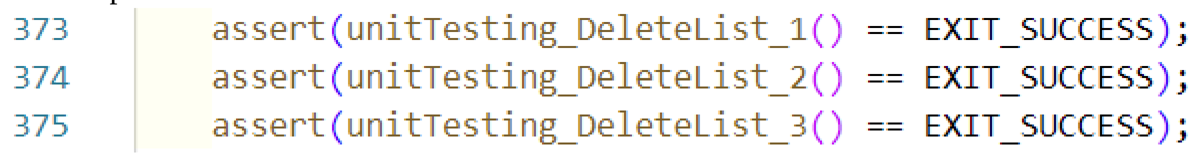
Table of Contents

Rules for Code Alignment **3**
Indentation 3
Brace Placement in Compound Statements 3
Line Separation and Grouping 3
Characters per Line of Code 4
Elements in Arrays 4
Vertical Alignment 4
Function Declaration Placement 5
Rules for Naming **6**
Variables 6
Functions 6
Structures 6
Rules for Commenting **7**
Rules for Code in General **7**

Rules for Code Alignment

Indentation

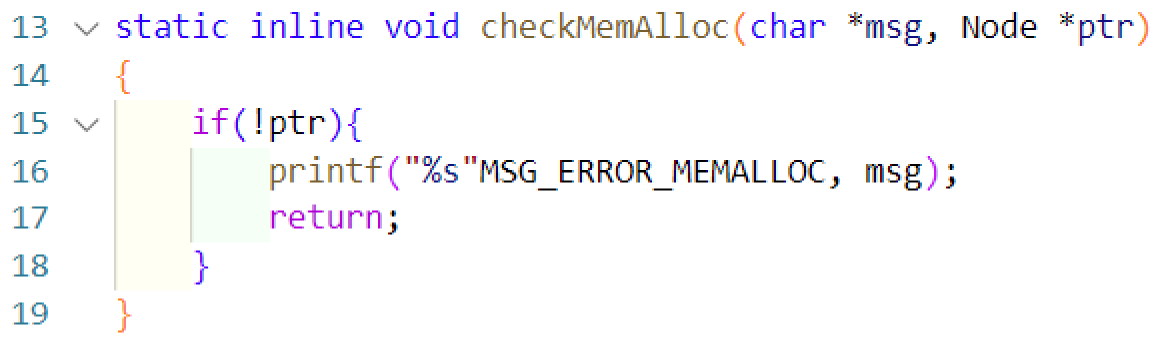
My code is written like a flight of stairs – one tab (four spaces) in each new line that is not of the same importance as the other line.



```
    assert(unitTesting_DeleteList_1() == EXIT_SUCCESS);
    assert(unitTesting_DeleteList_2() == EXIT_SUCCESS);
    assert(unitTesting_DeleteList_3() == EXIT_SUCCESS);
```

Brace Placement in Compound Statements

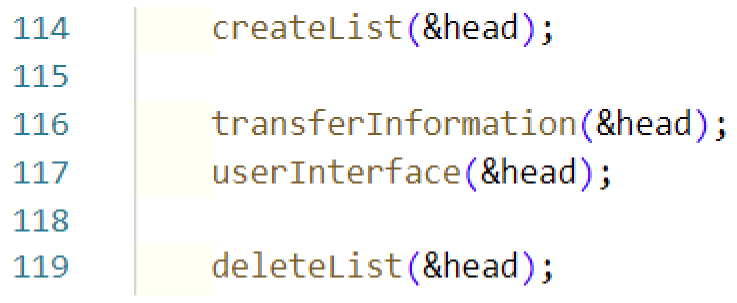
I exercise a combination of two placement styles: K&R and Allman. What it means is that for control statements I put the opening brace right after the round closing bracket. For functions, I put the opening brace in a new line after the definition of a function. If a control statement has no arguments, I put only a semicolon after it in a new line with no braces.



```
static inline void checkMemAlloc(char *msg, Node *ptr)
{
    if(!ptr){
        printf("%s"MSG_ERROR_MEMALLOC, msg);
        return;
    }
}
```

Line Separation and Grouping

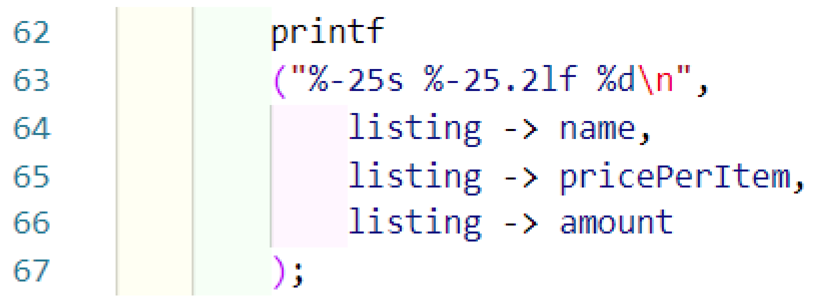
I separate lines by their meaning and purpose. If there is an inherent connection between certain blocks of code, those blocks will not be separated by whitespace.



```
    createList(&head);

    transferInformation(&head);
    userInterface(&head);

    deleteList(&head);
```

Characters per Line of Code

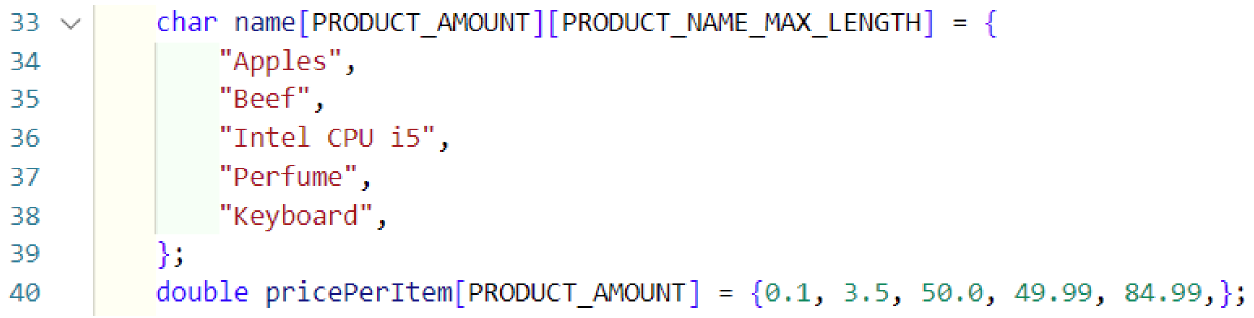
On my laptop's configuration, I can put 90 symbols in one line until they disappear from vision (I use half of my screen for programming). This characteristic is why I tend to divide longer statements into more spread out structures. Such a technique applies to control statements as well.



```
printf
("%-25s %-25.2lf %d\n",
    listing -> name,
    listing -> pricePerItem,
    listing -> amount
);
```

Elements in Arrays

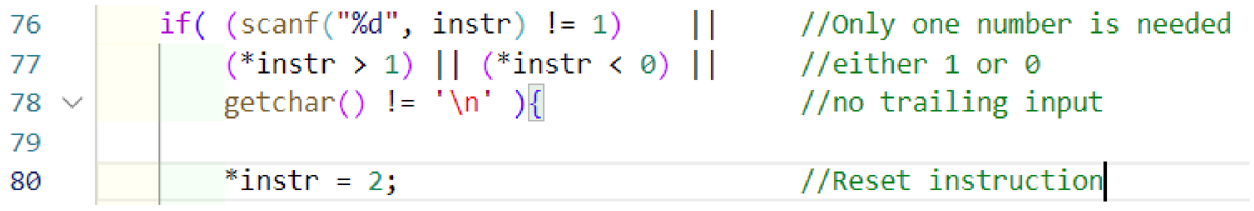
If an array does not take up much symbols, I write its elements on the same line. However, if the line gets too long, I break up the arrays' elements into an orderly list. On both occurrences I put a comma after the last member of the array.



```
char name[PRODUCT_AMOUNT][PRODUCT_NAME_MAX_LENGTH] = {
    "Apples",
    "Beef",
    "Intel CPU i5",
    "Perfume",
    "Keyboard",
};
double pricePerItem[PRODUCT_AMOUNT] = {0.1, 3.5, 50.0, 49.99, 84.99,};
```

Vertical Alignment

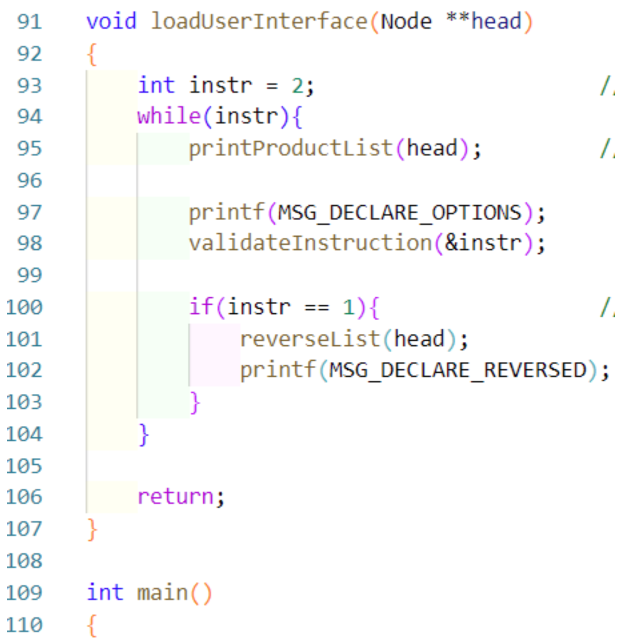
As mentioned before, I align lines of code according to their importance. Often, this extends to the elements of the lines and is expressed in vertical alignment (comments, operators in control statements, etc.). I tend to do such breaking up of code when: it helps to read the code better and maintain logic or I have to right-scroll.



```
if( (scanf("%d", instr) != 1)    ||     //Only one number is needed
    (*instr > 1) || (*instr < 0) ||     //either 1 or 0
    getchar() != '\n' ){                //no trailing input

    *instr = 2;                         //Reset instruction
```

Function Declaration Placement

Functions are declared above the *main* function. This is because its less lines of code and less time spent writing extra definitions.



```c
void loadUserInterface(Node **head)
{
    int instr = 2;                      /
    while(instr){
        printProductList(head);         /

        printf(MSG_DECLARE_OPTIONS);
        validateInstruction(&instr);

        if(instr == 1){                 /
            reverseList(head);
            printf(MSG_DECLARE_REVERSED);
        }
    }

    return;
}

int main()
{
```

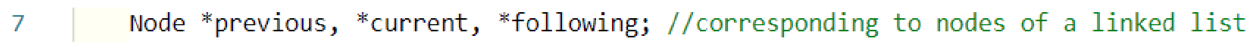
Rules for Naming

In general: all names are written in snakeCase (except preprocessor instructions and structs).

Variables

Variable naming depends on their importance. If a variable is only used once and primarily for one operation, I will not give much importance to the name. However, if the same variable is used in multiple places, its name becomes important. If a variable is used in one function as well as another function, I give the local variables the same name.

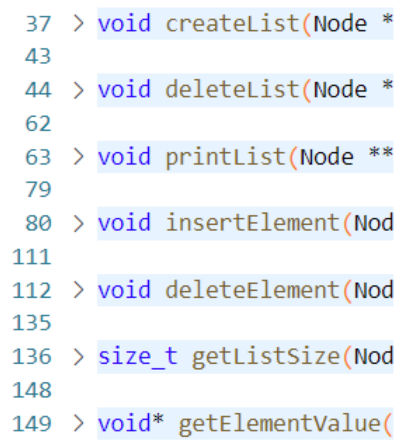
Names must reflect either an action or a property to which the variable is tied.



```
    Node *previous, *current, *following; //corresponding to nodes of a linked list
```

Functions

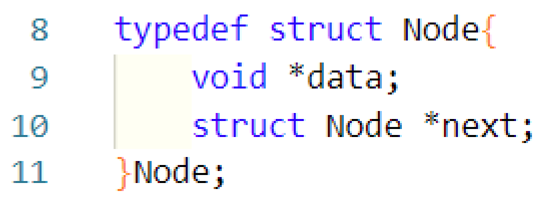
Function names must reflect an action in present time. Sometimes if function names have several compounding ideas, I will separate them with underscores.



```
> void createList(Node *

> void deleteList(Node *

> void printList(Node **

> void insertElement(Nod

> void deleteElement(Nod

> size_t getListSize(Nod

> void* getElementValue(
```

Structures

Same rules as variables but usually written as a noun rather than a verb.



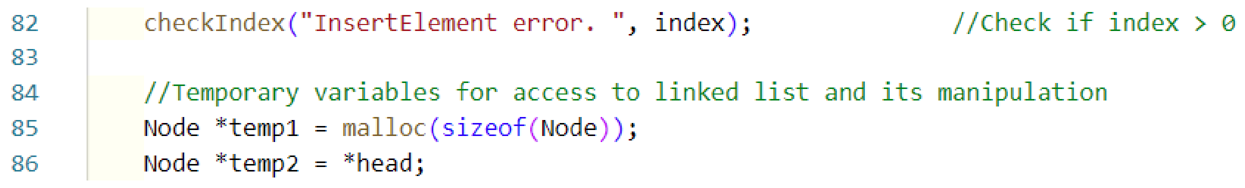
```
typedef struct Node{
    void *data;
    struct Node *next;
}Node;
```

Rules for Commenting

Comments must give additional readability to the code rather than echo already obvious things. Comments should be laconic and explicit.

Usually, I write comments in the same line as the code I want to highlight. However, if a line exceeds my screen (90 characters) I tend to put the comment above the line which I'm explaining. Also, If I'm commenting an entire block of code, I also put the comment above the block.

If code lines do not vary in length aggressively, I try to align all comments into a single column, no matter the distances between them, by using tabs.



```
checkIndex("InsertElement error. ", index);         //Check if index > 0

//Temporary variables for access to linked list and its manipulation
Node *temp1 = malloc(sizeof(Node));
Node *temp2 = *head;
```

Rules for Code in General

Code must be easily maintainable and, for most cases, easy to read from the first glance. Clutters of code blocks must be avoided and separated in accordance with the aforementioned rules. We write the code first, then we simplify it as much as possible: reducing complexity, refactoring, etc.

If a person thinks about how other people may perceive his code, then he might arise with some conclusions how to improve it. At the end of the day, you should write simple, not clever code.

Code should be consistent in its readability no matter the importance of different fragments, unless there is an explicit reason to do otherwise (for example, a program to only do very specific unit testing). Love over war but tabs over spaces.

Code should rely on the general principles of the language in which it was written. If personal conventions go against the language's standards, than either all people are on board with your convention, or you change that convention to fit the general standards.