

LOS N MANDAMIENTOS DE PROGRAMACIÓN



No nombrarás funciones ni variables de forma no descriptiva.

No desearás el código del prójimo.

Usarás mayúsculas y snake case para nombrar constantes.

No mezclarás idiomas en tu código.

Usarás pre y post condiciones.

Separarás los operadores de las variables.

No harás ciclos innecesarios.

No cortarás iteraciones con breaks o returns.

No implementarás funciones que hacen más de una cosa.

No declararás variables dentro de ciclos for ni while.

Mantendrás la indentación del código.

No pondrás comentarios excesivos.

En las siguientes páginas se especificarán los n mandamientos. Dícese que aquellos quienes cumplieron los mandamientos, promocionaron la materia Algoritmos y Programación II.

Disclaimer: El orden de los mandamientos no representa relevancia alguna a la importancia de cada uno.



Regla de oro indica que a continuación se presenta un tip importante



Excepción a la regla indica que a continuación aparece un caso que no aplica a lo anteriormente dicho



Recordatorio indica que a continuación aparece algo que es importante recordar

1. No nombrarás funciones ni variables de forma no descriptiva.

- Las funciones deben tener un nombre que identifique inequívocamente que es lo que hace.

Por ejemplo:

Si tengo una función que imprime por pantalla un menú se puede llamar¹:

- `mostrar_menu()`
- `imprimir_menu`

Pero NO se podría llamar:

- `menú()`
- `procesar_menu()`
- `obtener_menu()`



En la gran mayoría de los casos es suficiente utilizar un **verbo en infinitivo** (-ar, -er, -ir) + **sustantivo**



En el caso de que la función retorne un valor booleano, entonces su nombre debe tener la forma de una **afirmación**. Por ejemplo: `es_mayor_de_edad()`

- Las variables deben tener un nombre que dé a entender el contenido de esta.

Por ejemplo:

Si tengo una variable en la que guardo la cantidad de manzanas que tengo se puede llamar:




- `cantidad_manzanas`
- `cantidad_de_manzanas`

Pero **NO** la podría llamar:

- `manzanas`
- `cant_manz`
- `c_m`

Pero esto no es todo en este caso, ya que hay otras consideraciones para tener en cuenta:

¹ Si bien los nombres están en snake_case son igual de válidos usando camelCase, es decir, `mostrar_menu()` podría ser `mostrarMenu()`

-  Variables incrementales (resultado1, resultado2, ...)
-  Variables booleanas (es_primo, es_valido), aplica la misma regla que para las funciones que retornan valores booleanos
-  Variable i del for (i, j, k **sólo válido al usar for**)

2. No desearás el código del prójimo.

Aunque parezca redundante, es necesario aclarar que el código utilizado debe ser de autoría de cada uno. En caso de trabajar en grupo, el código utilizado debe ser de autoría de **todos** los miembros del grupo (o **al menos uno**), salvo que se indique lo contrario.

En caso de utilizar un código externo, se debe aclarar de dónde se obtuvo y una breve descripción de qué hace y cómo lo hace.

3. Usarás mayúsculas y snake case para nombrar constantes.

Las mayúsculas son una parte importantísima del código, no sólo nos ayudan a modificar valores cruciales del código de forma simple, sino que también a mejorar el entendimiento del código.



“El código se escribe una vez pero se lee muchas veces”

Ejemplos de constantes:

```
//includes

const int VIDA_MAXIMA = 100;

const int ERROR = -1;

const char OCEANOS = 'O';

//funciones
```



Todo valor numérico que tenga un significado de negocio (es decir que se pueda explicar con palabras porque existe, por ejemplo “1” es la **CANTIDAD_INTEGRANTES_TP_INDIVIDUAL**), debe ser una constante.



No todos los números sueltos son constantes! Por ejemplo inicializar un contador en 0 no amerita una constante, el 0 no es parte de una lógica de negocio, por lo tanto no tiene una descripción asociada.

4. No mezclarás idiomas en tu código.

No todo el mundo tiene el mismo manejo de idiomas por lo que mezclar lenguajes puede entorpecer el entendimiento de nuestro código.



Cuando se define el idioma en el que se desarrollará el proyecto todo deberá ser hecho en ese idioma, es decir, si trabajo en inglés **TODO** debe estar en inglés: comentarios, commits, funciones, etc.

5. Usarás pre y post condiciones.

La creación de funciones tiene como uno de sus objetivos la reutilización del código y para facilitar la reutilización del mismo se agrega junto a la firma de la función un comentario con las pre y post condiciones de la misma.

Precondiciones: son condiciones que debe cumplir lo que recibe mi función para poder llegar al resultado esperado.

Por ejemplo:

```
/*Precondiciones: La opción es un número entre 0
y 5 inclusive. */
//Postcondiciones: Realiza la opción indicada.
void procesar_opcion(int opcion);
```

Que no poner:

```
//Precondiciones: La función recibe un número.
//Postcondiciones: Realiza la opción indicada.
void procesar_opcion(int opcion);
```

En la firma de la función podemos ver que opción es un int por lo que la precondition no está agregando ninguna información relevante.



Violar una precondition implica generar una excepción. Si no se envía el tipo de dato que indica la firma de la función entonces nunca se ingresa a la función, por lo tanto nunca se genera una excepción y eso implica que no es una precondition.

Postcondiciones: Es lo que hace la función, que obtenemos si la llamamos correctamente.

Por ejemplo:

```
/*Precondiciones: recibe la opción elegida
precargada */
/*Postcondiciones: Si la opción no es un
número entre 0 y el valor de OPCIONES_VALIDAS
vuelve a solicitar los datos hasta que lo sea
*/
void validar_opcion_elegida(int &opcion_elegida);
```

Que no poner:

```
/*Precondiciones: recibe la opción elegida
precargada */
//Postcondiciones: valida la opción.
void validar_opcion_elegida(int &opcion_elegida);
```

Esta postcondición no nos indica que vamos a obtener de la función por lo que no sabemos que tanto cubre la función y que tanto tendremos que hacer nosotros luego de llamarla.



Hay funciones que no tienen precondiciones pero TODAS las funciones tienen postcondiciones.

6. Separarás los operadores de las variables.

La legibilidad del código es sumamente importante, como antes mencionamos el código se escribe una vez y se lee muchas veces por lo que es importante desarrollar código que sea agradable a la vista.

Hoy en día no es necesario escatimar con los caracteres que incluimos en nuestro código por lo que para ayudar a la legibilidad se deben separar los operadores de las variables.

Algunos ejemplos de operadores son : * , / , - , + , >> , << , -> , < , > , = , etc.

Ejemplo:


```
bool existe_el_numero(long numero, Agenda* agenda) {
    bool existe_el_numero = false;
    int contador_contactos = 0;

    while(!existe_el_numero && contador_contactos < agenda ->
cantidad_de_contactos) {
        if(numero == agenda -> contactos[contador_contactos] -> numero) {
            existe_el_numero = true;
        }
        contador_contactos++;
    }

    return existe_el_numero;
}
```

Que **no** hacer:

```
bool existe_el_numero(long numero, Agenda* agenda) {
    bool existe_el_numero=false;
    int contador_contactos=0;
    while(!existe_el_numero&&contador_contactos<agenda->cantidad_de_contactos) {
        if(numero==agenda->contactos[contador_contactos]->numero) {
            existe_el_numero=true;
        }
        contador_contactos++;
    }
    return existe_el_numero;
}
```




7. No harás ciclos innecesarios.

Las bucles son complejos y es sencillo equivocarse y definir bucles de más si no analizamos cuidadosamente el problema que queremos resolver. Es por esto que se recomienda analizar el problema antes de pasarlo a código y luego de hacerlo revisar que los ciclos no puedan simplificarse.


Por ejemplo

```
bool contiene_el_numero(int numeros[], int tamano_numeros, int
numero_buscado){
    bool numero_encontrado = false;
    for(int i = 0; i < tamano_numeros; i++){
        if(numeros[i] == numero_buscado){
            numero_encontrado = true;
        }
    }
    return numero_encontrado;
}
```



```
bool contiene_el_numero(int numeros[], int tamano_numeros, int
numero_buscado){
    bool numero_encontrado = false;
    int i = 0;

    while(!numero_encontrado && i < tamano_numeros){
        if(numeros[i] == numero_buscado){
            numero_encontrado = true;
        }
        i++;
    }
    return numero_encontrado;
}
```




8. No cortarás iteraciones con breaks o returns.

Todos los bucles tienen su condición de corte, en el for es la cantidad de iteraciones y por esto solo se usa cuando sabemos cuántas veces queremos iterar y si no lo sabemos utilizamos while con una variable booleana como condición de corte.

```
bool contiene_el_numero(int numeros[], int tamano_numeros, int
numero_buscado){
    bool numero_encontrado = false;
    int i = 0;

    while(i < tamano_numeros){
        if(numeros[i] == numero_buscado){
            numero_encontrado = true;
            break;
        }
        i++;
    }
    return numero_encontrado;
}
```



```
bool contiene_el_numero(int numeros[], int tamano_numeros, int
numero_buscado) {
    int i = 0;

    while(i < tamano_numeros){
        if(numeros[i] == numero_buscado){
            return true;
        }
        i++;
    }
    return false;
}
```



9. No implementarás funciones que hacen más de una cosa.

Uno debe preguntarse *¿qué es lo que hace esta función?*, si la respuesta es algo del estilo: *la función hace **una cosa Y otra cosa***, entonces todavía se puede modularizar la función más.



Por cada una de las responsabilidades identificamos en la función podemos crear una función.



Se recomienda tener cuidado y criterio, no modularizar es tan malo como modularizar de forma excesiva, buscar un punto medio

10. No declararás variables dentro de ciclos for ni while.

Las variables que sean necesarias para los bucles deben ser declaradas antes de los mismos, ya sea en el inicio de la función o inmediatamente antes del bucle al que pertenecen.



La única excepción es el contador del for que, además, puede tener como nombre un caracter. Usualmente se usa "i", "j" o "k".

11. Mantendrás la indentación del código.

Si bien hay lenguajes que no obligan a mantener la indentación es una buena práctica hacerlo para mantener la legibilidad.



Recomendamos hacer que el tab sea equivalente a 4 espacios

12. No pondrás comentarios excesivos.

Si bien los comentarios aportan legibilidad el exceso de los mismos puede causar el efecto contrario.




Si mi código es tan complejo que requiere un comentario más allá de las pre y post condiciones intento simplificarlo. Si no existe forma de simplificación de un comentario corto.

Que NO hacer:

```
bool existe_el_numero(long numero, Agenda* agenda) {
    bool existe_el_numero = false; //pongo en falso la existencia
    int contador_contactos = 0; //inicializo el contador en 0

    while(!existe_el_numero && contador_contactos < agenda -> cantidad_de_contactos){
        if(numero == agenda -> contactos[contador_contactos] -> numero){
            existe_el_numero = true; //si encuentro el numero cambio la existencia a
true
        }
        contador_contactos++; //avanzo el contador
    }

    return existe_el_numero; //retorno si existe o no el número
}
```




Que podría pasar:

```
void funcion_mega_compleja(long numero ... ){
    //Este while se encarga de hacer esta cosa súper compleja haciendo que ...
    while(condicion_de_corte_mega_compleja()){
        ...
    }
}
```

Incluso en esta situación se pone un comentario antes del bloque de código que es complejo y no se llenan todas las líneas de comentarios ya que haría que el código que de por si era complejo sea más complejo de leer.