Convenciones de Programación en C

1. Introducción

C es un lenguaje de programación creado en 1972 por Dennis M. Ritchie en los Laboratorios *Bell* como evolución del anterior lenguaje *B*, a su vez basado en BCPL. C es apreciado por la eficiencia del código que produce y es el lenguaje de programación más popular para crear software de sistemas, aunque también se utiliza para crear aplicaciones.

2. Ejemplo de un Programa en C

```
// Nombre del archivo con el código C (ej. prog.c)

/* Comentarios de un parrafo completo comprendidos entre
/*.....*/, sirven para aclarar el programa o una parte del
programa */

// Zona de archivos de cabecera de las librerías

#include <..... . h>   // h de Head

#include <..... . h>


// Zona de variables globales

int valor; float mediaTotal;


// Zona de prototipos de funciones

int Potencia (int x,y)


int main () // Prog. ppal. típico {

    // llave de inicio del programa

    // código del programa

    .......

    .......

    .......

     return(0)

    // fin del programa

}

//  Desarrollo del código de las funciones anteriores
```

3. Declaración de Constantes, Variables, Tipos y Funciones

Los nombres de las constantes, variables, tipos, funciones y acciones usados en un programa deben ser significativos (o mnemónicos), expresivos, claros, definitivamente relevantes a la solución del problema, y legibles. Aunque es posible escribir sus nombres de cualquier forma, es conveniente utilizar alguna convención. En la siguiente tabla se detallan las convenciones que usaremos en el curso de Introducción a la Algorítmica y Programación.

Nombre	Elemento declarado
variableLarga	Variable. Este estilo es el que usaremos cuando el nombre contiene más de una palabra.
variable_larga	Variable. "_" separa las palabras del identificador. No lo usaremos.
IVA	Constante. Siempre con una letras mayúsculas.
Tpersona	Tipo. Los nombres de tipos comienzan siempre con la letra T.
Ppersona	Puntero. Los nombres de punteros comienza siempre con la letra P.
potenciaDos(x)	Función. El nombre de una función comienza con minuscula.

Hay que destacar que cuando se usan varias palabras para formar un identificador, éstas deben ser muy legibles: estoSiSeVeBien, peroestocasiquenoseentiende. ¿O si?
Los únicos identificadores que comienzan con una minúscula son las variables; todos los demás identificadores que no pueden cambiar comienzan con una mayúscula: constantes, tipos, acciones, funciones, etc.

No se debe restringir el tamaño de un identificador, sino más bien debe escogerlo para que sea significativo. Sin embargo, ante la posibilidad de escoger entre uno largo y otro corto, deberá escogerse el corto si tiene la misma expresividad del largo. Si queremos denotar la altura de una persona, es mejor usar el identificador alt que alturaPersona, siempre y cuando alt sea suficientemente claro en el contexto de programación. En general, cuando se trata de varias palabras, es mejor pegar las palabras es el estilo que se está imponiendo como estilo general cuando se trata de dos o tres palabras.
Es importante utilizar constantes para evitar el uso de números en las expresiones, por ejemplo:

En lugar de escribir:

```
precioFinal = precioBase + precioBase*0,21;
```

es más conveniente escribir:

```
precioFinal = precioBase + precioBase*IVA;
```

La segunda expresión es más clara y más simple de modificar.

4. Palabras Reservadas

El lenguaje C posee un número reducido de palabras reservadas, tan solo 32, que define el standard ANSI-C. Las palabras reservadas deben escribirse en minúscula.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

5. Indentación

La indentación es la posición en que comienza una línea de código en el renglón. Es un anglicismo de uso común en informática. Por indentación se entiende mover un bloque de texto hacia la derecha insertando espacios o tabuladores para separarlo del texto adyacente. La indentación se utiliza para mejorar la legibilidad del código, teniendo en cuenta que los compiladores o intérpretes raramente los consideran. La indentación es uno de los factores más importantes para lograr mejores diseños de programas. Al indentar un programa adecuadamente se logra destacar su estructura de control. Son frecuentes las discusiones sobre cómo o dónde usar la indentación, si es mejor usar espacios en blanco o tabuladores. Para unificar la forma de escribir los algoritmos usaremos un estilo en particular, aunque no es el único.
La declaración de variables, constantes, tipos y funciones debe indentarse **dos (2) espacios** o **un (1) TAB**. El cuerpo principal del programa debe indentarse **dos (2) espacios** o **un (1) TAB**

Debe indentarse **dos (2) espacios** o **un (1) TAB** cada vez que se utilice una construcción anidada. Siempre debe indentarse una construcción anidada, y siempre debe indentarse en **dos espacios** o un TAB. La consistencia en la aplicación de esta regla permite predecir la forma del programa, y hacer la lectura más sencilla.

Ejemplo:

```
#include <stdio.h>

void main(){
  /* Variables */
  int  num;     //variable para almacenar el numero a analizar
  printf("Ingrese un numero entero: ");
  scanf("%d",&num);
  if (num<0) {
    //num es negativo
    printf("%d es negativo \n", num);
  }
  else {
    //num es positivo
    printf("%d es positivo \n", num);
  }
  printf("Muchas gracias! \n");
}
```

5.1. Indentación del if e if-else.

La indentación de `If` que se utilizará en los programas será:

```
if (num<0) {
  //num es negativo
  printf("%d es negativo \n", num);
}
```

La indentación de `If-else` que se utilizará en los programas será:

```
if (num<0) {
   //num es negativo
   printf("%d es negativo \n", num);
}
else {
   //num es positivo
   printf("%d es positivo \n", num);
}
```

En situaciones donde se requiere el anidamiento de instrucciones `If-else` la indentación será:

```
if (expresion_booleana 1){
   /* Se ejecuta cuando la expresion booleana 1 es verdadera */
} else if (expresion_booleana 2) {
   /* Se ejecuta cuando la expresion booleana 2 es verdadera */
} else if (expresion_booleana 3) {
   /* Se ejecuta cuando la expresion booleana 3 es verdadera */
} else {
    /* Se ejecuta cuando ninguna de las condiciones anteriores es verdadera */
}
```

5.2. Indentación del while

La indentación de `while` que se utilizará en los programas será:

```
int i=1;

suma=0;

while (i<=100) {

   suma = suma + i;

   i++;

}
```

5.3. Indentación del do

La indentación de `do` que se utilizará en los programas será:

```
int i,j;

do {

  scanf(“%d”,&i);

  scanf(“%d”,&j);

} while (i<j);
```

5.4. Indentación del for

La indentación de `for` que se utilizará en los programas será:

```
for(x=1;x<=100;x++){
  printf(“Estoy dentro del for”);
}
```

5.5. Indentación de las funciones

```
int potencia(int x) {

  int aux;

  aux= x * x;

  return aux;

}
```

6. Funciones como acciones:

C no posee acciones pero se pueden simular como un tipo especial de funciones que no retornan nada:

```
void potencia(int a)  // donde void (significa “nada”) es un tipo especial que indica que la función no retorna nada.
```

Ejemplo:

```
void numero(int x) {
  if (x>=0){
    printf("numero positivo");
  }
  else {
    printf("numero negativo");
  }
}
```

7. Comentarios

Es difícil indicar exactamente cuándo incluir comentarios en el programa. Sin embargo, todos los comentarios deben cumplir con lo siguiente:

c1) Deben ser completos
c2) Deben ser válidos
c3) Deben ser pertinentes
c4) Deben ser claros
c5) Deben ser coherentes
c6) Deben ser concisos

En cuanto a la forma de los comentarios, es conveniente que estén alineados unos con otros. Si están dentro de un bloque, deben estar adecuadamente indentados, y siempre que sea posible debe dejarse un espacio separando a las barras \\ del comentario. Las barras que enmarcan a un comentario deben estar alineadas. Por ejemplo:

```
/* Comentarios de un parrafo completo comprendidos entre
/*.....*/, sirven para aclarar el programa o una parte del
programa */

int x;                // Comentarios de 1 sola línea

int potencia(int a); // función que retorna ...
```

Los comentarios deben explicar en relación al programa los siguientes asuntos:

- el cometido del programa. Esto debe reflejarse al comienzo del mismo.
- el papel de un identificador cuando no se explique por sí mismo.
- en los subprogramas, si el parámetro es de entrada o de salida.

El cometido de un fragmento de programa se puede anticipar con un comentario:

```
{
  // lectura de datos
  instrucciones para la lectura de los datos

  //cálculos
  instrucciones para los cálculos
```

```
  //presentación de los resultados
    instrucciones de salida de resultados
}
```

8. Buenas prácticas de programación

Las buenas prácticas de programación se refieren a un conjunto de reglas o convenciones que se adoptan con el fin de mejorar el proceso de construcción de programas. La aplicación de reglas facilita la legibilidad y el entendimiento del programa, la búsqueda y corrección de errores (proceso de depuración), la optimización del código, entre otras.

8.1 Limitar la utilización de variables globales.

Una variable global tiene la característica de ser accesible a todos los módulos de un programa (módulo principal, acciones y funciones). Usar variables globales se presenta como interesante porque permite evitar tener que estar pasando datos a los módulos como parámetros. Una desventaja de las variables globales es que puede producir efectos indeseables. Por ejemplo, una variable global tiene espacio de memoria asignado durante todo el programa, entonces, si una cantidad importante de variables globales es usada por muy pocos módulos, tendríamos espacios de memoria ocupados y con poco uso. Otro problema, es que la variable es globalmente accesible, entonces alguna acción o función podría modificarla asignando a la variable un valor no deseado. Cuando una variable global tiene un valor no deseado puede producir fallas en cadena de los módulos del programa. El hecho de que la variable sea accesible por un número probablemente grande de funciones, hace más compleja y tediosa la tarea de depuración de un programa.

8.2 Evitar el uso de sentencias goto, break y continue.

Un principio básico de la programación estructurada es evitar romper el flujo secuencial de ejecución de un programa que tiene lugar cuando se usan sentencias como las citadas. Cuando un programa realiza saltos arbitrarios se complica su legibilidad y se dificultan las tareas de depuración.

También cuando las funciones hacen uso de estas sentencias se hace más difícil la reutilización de las mismas. Sin embargo, hay lenguajes donde estas sentencias son necesarias. Por ejemplo, en el lenguaje C se requiere la sentencia break para controlar el comportamiento de la sentencias switch.

8.3 Evaluar la conveniencia del uso de varias sentencias return en las funciones.

La recomendación de usar un único return por función, que se coloca como última sentencia de la función, se inspira en la aplicación de uno de los principios de la programación estructurada el cual establece que los programas deberían tener un único punto de entrada y un único punto de salida. La recomendación no es para mejorar el rendimiento sino que se hace por una cuestión de diseño. Tener varios puntos de salida en una función tiene el efecto de hacer más difícil seguir la lógica de la función y con ello complicar las tareas de depuración.

Es cierto que, el uso de un único return por función, colocado al final de la misma,

facilita la depuración y adaptabilidad de las funciones.

8.4 Evitar escribir acciones demasiado extensas.

Las acciones demasiado extensas es probable que contengan código para manejar varios detalles del problema que resuelven. Esto trae dificultades en su legibilidad y comprensión y por lo tanto complican las tareas de mantenimiento.
Generalmente en las acciones demasiado extensas se pueden identificar porciones de código que cumplen tareas diferenciadas. Entonces, cada uno de estos bloques puede conformar una acción separada.

8.5 Evitar copiar y pegar trozos de código casi idénticos a lo largo de un programa.

Frecuentemente sucede que en varios puntos de un programa se necesita una misma funcionalidad. Puede suceder, que en estos el grupo de sentencias que proveen tal funcionalidad se repita en cada uno de los puntos en cuestión. Por ejemplo, si un trozo de código que imprime una estructura de datos (vector, matriz, etc.) lo copiamos y pegamos en cada punto del programa donde se necesita esa funcionalidad, tendremos un programa con bloques de código repetidos.
Si la estructura de datos requiere algún cambio, por ejemplo, se modifica el tamaño del vector, será necesario retocar todos los bloques de código que están dispersos en el programa. Esta es una maniobra poco deseable porque puede ocasionar errores.
En estos casos es recomendable encapsular el bloque de código en una acción o función que se invocará cada vez que se requiera.

8.6 Colocar al lado de una llave que cierre un bloque de código un indicativo de que tipo de estructura cierra.

Es frecuente que cuando se anidan bloques de código se escriban al final del anidamiento una secuencia de cierre de llaves (o sea }}}}). En tal caso, no es muy fácil identificar qué tipo de bloque de código, o estructura, cierra cada llave. Es aconsejable poner al lado de cada llave un comentario que indique el nombre de la estructura, o el nombre de la acción que cierra.
Por otro lado, en situaciones donde se requiere el anidamiento de bloques, es muy recomendable aplicar técnicas de indentación.

8.7 Inicializar las variables cuando se declaran.

En muchos lenguajes, las variables declaradas se inicializan automáticamente con un valor adecuado según su tipo. Por ejemplo, si una variable es de tipo entero se inicializa con cero. Hay otros lenguajes (C ó C++), que no inicializan las variables al ser declaradas. De modo que las variables no contienen un valor de inicialización por defecto, estas variables contienen entonces un valor indeterminado que influirá en el comportamiento del programa.

8.8 No usar caracteres propios del castellano en la codificación de programas.

Las vocales acentuadas y la ñ, al no ser caracteres básicos ASCII, pueden sufrir alteraciones cuando un archivo de código se abre en dos computadoras diferentes,

incluso si ambas contienen el mismo sistema operativo. Para evitar estos problemas se recomienda evitar el uso de estos caracteres.

8.9 Evitar desbordamientos cuando se accede a vectores.

Cuando se accede a un vector puede suceder que la posición de acceso esté fuera de los límites del vector (arreglo). O sea que es conveniente verificar si el código de acceso puede llegar a generar un valor de posición inválido (fuera de los límites).

8.10 Evaluar la conveniencia de usar elementos no habituales de un lenguaje.

Muchos lenguajes, como C, contienen elementos que constituyen excepciones a la regla general. Normalmente, estos elementos son solamente conocidos por aquellos programadores que han leído el manual de referencia del lenguaje en profundidad buscando conocer los secretos más ocultos del lenguaje de programación.
Es recomendable que el código sea escrito para que sea entendido por la mayor cantidad de programadores. Por tanto, se procura obtener una solución sencilla y que pueda ser bien entendida.
Sin embargo, en muchos casos, el tipo o la complejidad del problema a resolver, o requerimientos de eficiencia del código hará necesario recurrir a elementos no habituales del lenguaje.

9. Estructura del código fuente de un programa

En todo programa se pueden identificar secciones o unidades de código que cumplen tareas específicas. La disposición de estas secciones en el programa influye fuertemente en aspectos como el entendimiento del código, el seguimiento de los posibles flujos del programa, las tareas de depuración, el mantenimiento del código, etc.

De modo que, por las razones expuestas, establecer convenciones sobre la ubicación a dar a cada una de estas secciones dentro de un programa, es una práctica necesaria en los ámbitos de desarrollo de software.

Independientemente del lenguaje a utilizar, una convención para la estructura de un programa sería:

Sección		Contenido
Comentarios		Texto describiendo las funcionalidades del programa.
Cabecera		Nombre del programa.
Declaraciones	Unidades o librerías	Declaraciones de constantes, tipos de datos, variables, acciones (funciones y/o procedimientos) que pueden emplearse en el programa.
	Etiquetas	Identificadores para realizar saltos incondicionales en la secuencia de instrucciones de un programa (no recomendable).

	Constantes	Datos que no cambian durante la ejecución del programa.
	Tipos	Tipos de datos estructurados.
	Variables	Identificadores de espacios de memoria donde se almacenarán datos durante la ejecución del programa.
	Acciones	Prototipos de acciones.
Cuerpo principal		Instrucciones que controlan la ejecución del programa.
Acciones		Conjunto de instrucciones que implementan la funcionalidad de las acciones.

Una implementación de esta estructura para un programa en lenguaje C se puede observar en el punto 2.