1 Contenido

Unidad 2: Descripción de un programa

2 Un poco de historia

C es el lenguaje de programación de propósito general asociado, de modo universal, al sistema operativo UNIX. Sin embargo, la popularidad, eficacia y potencia de C, se ha producido porque este lenguaje no está prácticamente asociado a ningún sistema operativo, ni a ninguna máquina, en especial. Ésta es la razón fundamental, por la cual C, es conocido como el lenguaje de programación de sistemas, por Excelencia.

C nació realmente en 1978, con la publicación de The C Programming Languaje, por Brian Kernighan y Dennis Ritchie (Prentice Hall, 1978). Desde su nacimiento, C fue creciendo en popularidad y los sucesivos cambios en el lenguaje a lo largo de los años junto a la creación de compiladores por grupos no involucrados en su diseño, hicieron necesario pensar en la estandarización de la definición del lenguaje C.

Así, en 1983, el American National Estándar Institute (ANSI), una organización internacional de estandarización, creó un comité (el denominado X3J11) cuya tarea fundamental consistía en hacer «una definición no ambigua del lenguaje C, e independiente de la máquina». Había nacido el estándar ANSI del lenguaje C. Con esta definición de C se asegura que cualquier fabricante de software que vende un compilador ANSI C incorpora todas las características del lenguaje, especificadas por el estándar. Esto significa también que los programadores que escriban programas en C estándar tendrán la seguridad de que correrán sus modificaciones en cualquier sistema que tenga un compilador C.

Hoy, en el siglo XXI, C sigue siendo uno de los lenguajes de programación más utilizados en la industria del software, así como en institutos tecnológicos, escuelas de ingeniería y universidades. Prácticamente todos los fabricantes de sistemas operativos, UNIX, LINUX, MacOS, MS Windows, SOLARIS, ... soportan diferentes tipos de compiladores de lenguaje C.

Algunas ventajas que justifican el uso todavía creciente del lenguaje C en la programación de computadoras son:

- poderoso y flexible, con órdenes, operaciones y funciones de biblioteca que se pueden utilizar para escribir la mayoría de los programas que corren en la computadora
- Se puede utilizar C para desarrollar sistemas operativos, compiladores, sistemas de tiempo real y aplicaciones de comunicaciones.
- Debido a que existen muchos programas escritos en C, se han creado numerosas bibliotecas C para programadores profesionales que soportan gran variedad de aplicaciones. Existen bibliotecas del lenguaje C que soportan aplicaciones de bases de datos, gráficos, edición de texto, comunicaciones, etc.

En la actualidad son muchos los fabricantes de compiladores C, aunque los más populares entre los fabricantes de software son: Microsoft, Imprise, etc.

Una evolución de C, el lenguaje C++ (C con clases) que contiene entre otras, todas las características de ANSI C. Los compiladores más empleados Visual C++ de Microsoft. Builder C++ de lmprise antigua Borland, C++ bajo UNIX y LINUX.

En el verano del 2000, Microsoft patentó una nueva versión de C++, que es C#, una evolución del C++ estándar, con propiedades de Java y diseñado para aplicaciones en línea, Internet (on line) y fuera de línea.

3 Estructura General de un Programa en C

Un programa en C se compone de una o más funciones, donde una de las funciones debe ser obligatoriamente la función main. Las funciones son módulos que se identifican por un nombre propio. Estos módulos comprenden un grupo de instrucciones que realizan una o más acciones específicas.

Si, por ejemplo, se desea crear un programa que dado un número natural muestre la suma de sus dígitos, las tareas a realizar son:

a- ingresar un número

b- calcular la suma de sus dígitos

c- mostrar el resultado del cálculo del punto b-

Entonces, el punto b puede ser resuelto por un módulo que reciba un número natural y sume sus dígitos y los demás puntos, de a- a c- por un segundo módulo que invoca al módulo que resuelve el punto b-

La función `main` es la función principal en un programa en C y desde ella se convocan y coordinan las demás funciones. El programa correspondiente al enunciado anterior sería:

```
#include <stdio.h>                          /*Incluye la librería stdio.h necesaria para entrada y salida de datos*/

int sumadig(int);                           /*Declaración del prototipo del módulo sumadig*/

int main(void)                              /* Definición del módulo principal llamado main*/
{                                           /*Inicio del cuerpo del módulo main*/
        int numero,suma;                    /*Declaración de variables utilizadas en main*/
        printf("\n ingrese un natural");    /*Salida de cartel*/
        scanf("%i",&numero);                /*Ingreso de un dato en la variable numero*/
        suma=sumadig(numero);               /*Invocación del módulo sumadig y asignación en suma del resultado calculado*/
        printf("\n la suma es: %d", suma);  /*Salida de resultado*/
        return 0;                           /*Valor retornado por main, indicando que no hubo error */
}                                           /*Fin del cuerpo del módulo main*/
```

```
int sumadig(int num)                /* Definición del módulo sumadig. El parámetro num
                                       contiene en dato con el que se realiza el cálculo*/
{                                   /*Inicio del cuerpo del módulo sumadig*/
        int sum;                    /*Declaración de la variable utilizada en sumadig*/
        sum=0;                      /*Inicialización de la variable suma*/
        while (num!=0)              /*Cabecera del ciclo condicionado*/
        {                           /*Inicio del cuerpo del ciclo condicionado*/
                sum=sum+num % 10;   /*Acumulación del dígito menos significativo de num*/
                num=num / 10;       /*Recálculo de num*/
        }                           /*Fin del cuerpo del ciclo condicionado*/
        return sum;                 /*Valor de retorno del módulo sumadig*/
}                                   /*Fin del cuerpo del módulo sumadig*/
```

Un programa en C puede incluir:

- directivas de preprocesador,
- declaraciones globales,
- la función `main()`,
- funciones definidas por el usuario,
- comentarios del programa.

El formato general de un programa en C se muestra a continuación:

Sintaxis:

```
#include ...          /*Directiva de preprocesador*/
#define ...           /*Macro de preprocesador*/

/*Declaraciones globales de prototipos de funciones y variables
globales*/
int main() /*Función principal*/
{
  ...
}

/*Definición de otras funciones*/
tipo1 func1( . . . )
{
}
...
```

El siguiente programa ejemplifica la estructura típica y completa de un programa C:

Ejemplo 1:

```
1. /*Listado DEMO-UN0.C. Programa de saludo * /
2. /*programa sencillo en C*/
3. #include<stdio.h>
```

```
4. / * Este programa imprime: Bienvenido a la programación en C * /
5. int main()
6. {
7.    printf("Bienvenido a la programación en C\n");
8.    return 0;
9. }
```

El texto presentado entre `/*...*/`, líneas 1, 2 y 4, son comentarios, es decir, texto que no es interpretado ni traducido por el compilador. El comentario tiene el sólo efecto de servir como documentación interna de un programa.

Mediante la directiva de preprocesador `#include` se incluyen archivos externos llamados librerías, que contienen declaraciones de tipos de datos y de funciones que pueden ser usadas en el programa y que no están definidas en dicho programa. Cada librería es creada a partir de dos archivos, uno llamado archivo de cabeceras, cuya extensión es `.h` y el otro cuya extensión es `.c`, el cual contiene la definición de los módulos declarados en el archivo de cabecera. Con estos archivos `.h` y `.c` se crea la librería.

Al incluir, mediante directiva de preprocesador, el archivo de cabecera se está indicando al compilador que agregue al código fuente las declaraciones contenidas en los archivos de cabecera incluidos y las librerías que deben ser enlazadas para generar el programa ejecutable. En el ejemplo se incluye el archivo `stdio.h` (línea 3), el cual contiene declaraciones de funciones relacionadas con el ingreso y salida de datos del programa desde y hacia dispositivos estándar como lo son el teclado y la pantalla. Este archivo fue incluido en el programa ejemplo para poder luego, desde la función `main()`, invocar a la función `printf()`, en la línea 7, declarada en `stdio.h`. Obsérvese que los ángulos < y > no son parte del nombre del archivo; se utilizan para indicar que el archivo es un archivo de la biblioteca estándar C.

El programa contiene la función `main()`, la cual se define estableciendo en primer lugar la **cabecera** de la misma, como se puede observar en la línea 5 del Ejemplo 1. La palabra `int` indica que la función, al finalizar, retornará un valor entero. Este valor es generalmente utilizado para codificar posibles errores que se produzcan durante la ejecución de dicha función `main()`. Para el ejemplo, el valor que retorna la función `main()` es `0`, indicando que no se produjeron fallas de funcionamiento. Esta acción la realiza mediante la sentencia `return 0`. Los paréntesis de apertura y cierre que acompañan al nombre de la función indican que la misma no requiere de parámetros, es decir, que no requiere de valores externos para su funcionamiento. También se puede escribir la palabra reservada `void` entre los paréntesis. Esta palabra también indica al procesador que no se requieren parámetros.

Luego de la cabecera, la función presenta el **cuerpo** de la misma. El cuerpo de una función inicia con una llave que abre, `{` y finaliza con una llave de cierre, `}`. Entre estas llaves se presentan las sentencias que ejecutará la función. En el ejemplo 1 es el código comprendido por las líneas 6 a 9.Cada sentencia finaliza con un punto y coma (`;`). El cuerpo de la función `main()` del ejemplo 1 posee dos sentencias, la primera, `printf("Bienvenido a la programación en C\n");` es la invocación a la función `printf()`, con el valor de parámetro `"Bienvenido a la programación en C\n"`. Esta invocación permite que la función `printf()` registre en la pantalla el cartel de bienvenida pasado por parámetro. El símbolo barra n (`\n`) es el símbolo de nueva línea. Poniendo este símbolo al final de la cadena entre comillas, indica al sistema que comience una nueva línea después de imprimir los caracteres precedentes, terminando, por consiguiente, la línea actual. La segunda sentencia, `return 0;`, establece el valor que tomará la función `main()`, en este caso, valor cero (`0`). Esta sentencia termina la ejecución del programa y devuelve el control al sistema operativo de la computadora.

3.1 Directivas de Preprocesador

El preprocesador es una parte del compilador que se ejecuta en primer lugar, cuando se compila un código fuente C y que realiza unas determinadas operaciones, independientes del propio lenguaje C. Estas operaciones se realizan a nivel léxico: la inclusión de otros textos en un punto del código fuente, realizar sustituciones o eliminar ciertas partes del programa fuente. Debemos tener en cuenta que el preprocesador trabaja únicamente con el texto del código fuente y no tiene en cuenta ningún aspecto sintáctico ni semántico del lenguaje.

El control del preprocesador se realiza mediante determinadas directivas incluidas en el código fuente. Todas las directivas del preprocesador comienzan con el signo `#`, que indica al compilador que lea las directivas antes de compilar las demás sentencias del programa.

Las *directivas* son instrucciones al compilador, no son generalmente sentencias -obsérvese que su línea no termina en punto y coma-, sino instrucciones que se dan al compilador antes de que el programa se compile.

Las directivas pueden definir macros, nombres de constantes, archivos fuente adicionales, etc.

La directiva `#define` se utiliza para definir una macro. Las macros proporcionan principalmente un mecanismo para la sustitución léxica. Una macro se define de la forma:

Sintaxis:

```
#define id secuencia
```

Cada ocurrencia de `id` en el fuente es sustituida por `secuencia`.

Por ejemplo, la directiva:

```
#define TAM-LINEA 65
```

Sustituirá `TAM-LINEA` por el valor `65` cada vez que aparezca en el programa.

Un uso frecuente de las directivas de preprocesador consiste en la inclusión de archivos de cabecera. Existen archivos de cabecera estándar que se utilizan ampliamente, tales como `stdio.h`, `stdlib.h`, `math.h`, `string.h`. En la implementación de los programas también se utilizarán otros archivos de cabecera definidos por el usuario. La directiva `#include` es utilizada para incluir tanto archivos definidos por el usuario como archivos de estándar.

La directiva `#include` indica al compilador que lea el archivo fuente que viene a continuación de ella y su contenido lo inserte en la posición donde se encuentra dicha directiva.

Los archivos de cabecera (archivos con extensión `.h` contienen código fuente .c que se sitúan en un programa C. Estos archivos forman bibliotecas. El uso de bibliotecas permite la reutilización de código en distintos programas.

La directiva del preprocesador `#include` tiene el siguiente formato:

Sintaxis:

```
#include <nombrearch.h>
```
o bien
```
#include "nombrearch.h"
```

`nombrearch` debe ser un archivo de texto ASCII (su archivo fuente) que reside en su disco. En realidad, la directiva del preprocesador mezcla un archivo de disco en su programa fuente.

El nombre del archivo entre ángulos significa que los archivos se encuentran en el directorio por defecto *include*. El segundo formato, donde el nombre del archivo se encierra entre comillas, significa que el

archivo está en el directorio actual. Los dos formatos no son excluyentes y pueden coexistir en el mismo programa. Si desea utilizar un archivo de cabecera que se creó y no está en el directorio por defecto, se encierra el archivo de cabecera y el camino entre comillas, tal como:

```
#include 'D: \MIPROG\cabeza.h"
```

La mayoría de los programadores C sitúan las directivas del preprocesador al principio del programa, aunque esta posición no es obligatoria.

El archivo de cabecera más frecuente es `stdio.h`. Este archivo proporciona al compilador C la información necesaria sobre las funciones de biblioteca que realizan operaciones de entrada y salida. En general, los programas imprimen información en pantalla y leen datos de teclado, por lo que necesitan invocar a las funciones `scanf()` y `printf()`, cuyos prototipos, también llamados cabeceras, están declarados en el archivo `stdio.h`.

3.2 Declaraciones Globales

Las declaraciones globales indican al compilador que las funciones definidas por el usuario o variables así declaradas son comunes a todas las funciones de su programa, es decir, que estas funciones y variables pueden ser utilizadas desde cualquier lugar del programa y desde cualquier función, lo cual torna complicado el control de errores en el programa. Esta es la razón por las que se busca minimizar su uso. Las declaraciones globales se sitúan antes de la función `main()`.

Las declaraciones de función se denominan prototipos, por ejemplo:

```
int media(int a, int b);
```

El siguiente programa es una estructura modelo que incluye declaraciones globales.

Ejemplo 2:

```
1.  / * Programa demo.C * /
2.  #include<stdio.h>
3.  / * Definición de macros * /
4.  #define MICONST1 0.50
5.  #define MICONS2 0.75
6.  / * Declaraciones globales * /
7.  int Calificaciones ;
8.  int main( )
9.  {
10.    ...
11. }
```

3.3 Función `main()`

Cada programa C tiene una función main( ) que es el punto de entrada al programa. Su estructura es:

Sintaxis:

```
int main( )
{
  /* Secuencia de sentencias */
```

```
}
```

Las sentencias incluidas entre las llaves `{ ... }` se denominan *bloque o cuerpo.* Un programa debe tener sólo una función `main()` . Además de la función `main()`, un programa C consta de una colección de funciones.

Una función es un subprograma que posee una función establecida y retorna un valor. El retorno del valor lo realiza en el nombre de la función, convirtiéndose ese nombre en un lugar de memoria capaz de albergar datos del tipo asociado y declarado en el prototipo de la función.

En un programa corto, el programa completo puede incluirse totalmente en la función `main()`. Un programa largo, sin embargo, tiene demasiado código para incluirlo en esta función. La función `main()` en un programa largo consta prácticamente de llamadas a las funciones definidas por el usuario.

Las variables y constantes globales se declaran y definen fuera de la definición de las funciones, generalmente al inicio del programa, antes de `main()`, mientras que las variables y constantes locales se declaran y definen al inicio del cuerpo o bloque de la función principal, o al inicio de cualquier bloque.

3.4 Funciones Definidas por el Usuario

Un programa C se crea a partir de la integración de funciones. Se debe procurar diseñar las funciones de manera tal que cada una de ellas realice una única tarea, a partir de una serie de sentencias definidas en el bloque o cuerpo de la función.

Las funciones definidas por el usuario se invocan por su nombre y el listado de parámetros actuales que puedan tener. Una vez que la función es invocada, el control del programa pasa a dicha función, desde la cual se ejecuta el conjunto de sentencias definidas en el cuerpo de la misma. Cuando la función finaliza, el control del programa retorna a función llamadora.

Todas las funciones tienen nombre y una lista de valores que reciben, llamados parámetros. Se puede asignar cualquier nombre a su función, pero normalmente se procura que dicho nombre describa el propósito de la función. En C, las funciones requieren una declaración o prototipo en el programa, por ejemplo:

```
void muestra_pantalla(int a, real b);
```

o bien

```
void muestra_pantalla(int, real);
```

La palabra reservada `void` indica que la función no retorna valor. Puede observarse que en el primer ejemplo de declaración, los parámetros reciben un nombre, `a` y `b` respectivamente, mientras que en el segundo ejemplo, los parámetros sólo son declarados por tipos y no se acompañan nombres. En este segundo caso sólo se determina el orden y el tipo que se requiere. El nombre de los parámetros será establecido en el momento de la definición de la función, lo que generalmente se realiza debajo de la definición de la función `main()`.

La definición de la función se realiza mediante la siguiente estructura:

Sintaxis:

```
/*Cabecera de la función*/
tipo-retorno nombre-función (lista-de-parámetros)
{
  ...            /*sentencias cuerpo de la función*/
  return ...;    /*retorno de lafunción*/
}                /*finde la función*/
```

`tipo-retorno`: Es el tipo de valor, o `void`, devuelto por la función.
`nombre-función`: Nombre de la función
`lista-de-parámetros`: Lista de parámetros, o `void`, pasados a la función. Se conoce también como argumentos formales.

Ejemplo 3:

```
1.  / * ejemplo funciones definidas por el usuario */
2.  #include <stdio.h>
3.  void visualizar(); /* Declaración de la función */
4.  int main ( )
5.  {
6.     visualizar();
7.     return 0;
8.  }
9.  /*Definición de la función*/
10. void visualizar()
11. {
12.    printf ( "primeros pasos en C\n");
13. }
```

3.5 Comentarios

Los comentarios en C estándar comienzan con la secuencia / * y terminan con la secuencia * /.Todo el texto situado entre las dos secuencias es un comentario ignorado por el compilador.

```
/* PRUEBA1.C - Primer programa C * /
```

Si se necesitan varias líneas de programa se puede hacer lo siguiente:

```
/ *
Programa: PRUEBA1.C
Programador: Pepe Mortimer
Descripción: Primer programa C
Fecha creación: Septiembre 2000
Revisión: Ninguna
*/
```

También se pueden situar comentarios de la forma siguiente:

```
scanf ("%d" , &x) ; / * sentencia de entrada de un valor entero*/
```

El uso de comentario es muy recomendado para ser utilizado como documentación interna del programa.

4 Los Elementos de un Programa en C

Un programa C consta de uno o más archivos. Un archivo es traducido en diferentes fases. La primera fase es el preprocesado, que realiza la inclusión de archivos y la sustitución de macros. El preprocesador se

controla por directivas introducidas por líneas que contienen # como primer carácter. El resultado del preprocesador es una secuencia de tokens, también llamados elementos léxicos de los programas, que son: identificadores, palabras reservadas, constantes literales, operadores y signos de puntuación y otros separadores.

4.1 Identificador

Un identificador es una secuencia de caracteres, letras, dígitos y subrayados _ utilizado para dar nombre a constantes, variables, tipos de datos, funciones, etc. La regla que sigue la formación de los identificadores es la siguiente:

1. Debe ser una secuencia de letras o dígitos; el primer carácter debe ser una letra (algún compilador admite carácter de subrayado). En algunas versiones de C, los identificadores que comienzan con _ son utilizados en las librerías estándares de C.
2. Los identificadores son sensibles a las mayúsculas:

   `CantNum` es distinto a `cantnum`
3. Los identificadores pueden tener cualquier longitud, pero sólo son significativos los 32 primeros caracteres.
4. Los identificadores no pueden ser palabras reservadas, tales como if, switch o else.

4.2 Palabras Reservadas

Una palabra reservada (keyword o reseved word), tal como `void` son palabras del lenguaje de programación asociada con algún significado especial.

Los siguientes identificadores están reservados para utilizarlos como palabras reservadas, y no se deben emplear para otros propósitos.

`auto`	`double`	`long`
`asm`	`else`	`register`
`break`	`enum`	`return`
`case`	`extern`	`short`
`char`	`float`	`signed`
`const`	`for`	`sizeof`
`continue`	`goto`	`static`
`default`	`int`	`struct`
`do`	`if`	`switch`

```
typedef          unsigned          volati1e
union            void              while
```

4.3 Comentarios

Como se dijo anteriormente, sólo sirven para documentación interna del programa ya que no son tenidos en cuenta por el compilador.

Los comentarios se encierran entre / * y * / pueden extenderse a lo largo de varias líneas.

```
/ * Titulo: Demo-uno por Mr. Martinez * /
```

Otra forma, el comentario en dos líneas:

```
/ * Cabecera del programa text-uno
Autor: J.R. Mazinger * /
```

4.4 Signos de Puntuación y Separadores

Son signos especiales que permiten al compilador separar y reconocer las diferentes unidades sintácticas del lenguaje.

Todas las sentencias deben terminar con un punto y coma. Otros signos de puntuación son:

```
!   %   ^   &   *   (   )   -   +   =   {   }   ~
[   ]   \   ;   '   :   <   >   ?   ,   .   /   "
```

Los separadores son espacios en blanco, tabulaciones, retornos de carro y avances de línea.

5 Los Datos y su Representación

Los datos están constituidos por el registro físico de los hechos, acontecimientos, transacciones, etc. Es un símbolo físico utilizado para representar la información. La información implica que los datos estén procesados de manera útil y significativa para quien lo recibe.

5.1 Tipos de Datos

Un tipo de dato es un patrón que determina el conjunto de valores que pueden tomar las variables asociadas a dicho tipo. También define la representación interna de los datos y las operaciones definidas para dicho tipo.

5.1.1 Métodos de Representación Interna de los Tipos de Datos Numéricos

Los datos enteros pueden ser positivos o negativos. Existen distintos métodos para la representación interna de estos datos, el binario puro, módulo y signo, complemento a 1 y complemento a 2. Todos estos métodos dependen de la cantidad de dígitos disponibles para representar los números, lo cual también se relaciona estrechamente con la palabra del ordenador, que es la cantidad de bits que se pueden transferir de una vez, siendo generalmente 16 bits.

El rango de representación es el intervalo de valores permitidos para un tipo de datos. Al operar con los datos enteros pueden calcularse resultados que quedan fuera de tal rango, es decir que se produce un error por desbordamiento de rango.

El método binario puro consiste en expresar el dato en sistema de numeración de base 2. El rango de representación, considerando que se tiene disponible n dígitos binarios, es 0 < valor < 2^n - 1. Entonces, si se disponen de 2 bytes para representar los enteros, el rango será [0, 65535].

El método módulo y signo consiste en disponer, del total de bits asignados para la representación del número, un bit llamado bit de signo para representar el signo del número entero, los bits restantes representan el módulo del número. Entonces, si se destinan n bits para representar el número, se dispondrá de n-1 bits para la mantisa, por lo que el rango de representación es $-2^{n-1}+1 \leq$ valor $\leq 2^{n-1}-1$.

El método módulo y signo tiene la particularidad de que el cero tiene una doble representación, con bit de signo positivo y con bit de signo negativo. Entonces, si se tiene 2 bytes, 16 bits para representar los enteros, se destina uno para signo, quedando 15 bits para mantisa, por lo que el rango de representación es [-32767, 32767], existiendo en este método, dos representaciones para el cero.

El método Complemento a 1 representa al igual que en el de módulo y signo el número el signo mediante el bit más significativo, 0 para los positivos y 1 para los negativos. El módulo es representado con los bits restantes, utilizando el binario puro para los positivos y el complemento a 1, incluyendo el bit de signo, para representar los negativos. Entonces, si se utiliza 1 byte para representar a los enteros, se destinan 7 bits para el módulo del número. Con este método, la representación del número 25 será 0 0011001, mientras que la representación del número -25 será el complemento a 1 del 25, incluyendo el signo, 1 1100110.Este método es simétrico y el rango de representación es el mismo que el método Módulo y signo,$-2^{n-1}+1 \leq$ valor $\leq 2^{n-1}-1$; por lo que persiste la doble representación para el cero.

El método Complemento a 2 al igual que los dos anteriores, utiliza el bit más significativo para representar el signo, 0 para positivo y 1 para negativo. El módulo del número positivo es representado en binario puro, mientras que el del número negativo es representado como el complemento a 2 del correspondiente positivo. La forma de representación genera un rango asimétrico, $-2^{n-1} \leq$ valor $\leq 2^{n-1}-1$. La representación del número 25 será 0 0011001, mientras que la representación del número -25 será el complemento a 2 del 25, incluyendo el signo, 1 1100111. La ventaja del método es que existe una única representación para el cero.

Los datos reales se representan en notación exponencial o científica, donde el número = mantisa x base_de_exponenciación^{exponente}. Así, el número 25.7 puede ser representado de infinitas formas, como por ejemplo $2.57x10^1$; o bien $0.257x10^2$; o $257x10^{-1}$Atento a esta situación, se procura una representación normalizada del número, donde el valor absoluto de la mantisa de cualquier número excepto el cero esté comprendido en el intervalo 0.1<|mantisa|<1; siendo 0 el valor de la mantisa para el cero. Para la representación de los datos se utiliza entre 4 y 8 bytes. Si la representación es de 4 bytes se denomina representación de simple precisión, mientras que si la representación es de 8 bytes se denomina representación de doble precisión. En general, en cualquiera de las dos representaciones se destina un byte para representar el exponente, de este byte, un bit es para el signo del exponente y los otros 7 bits restantes para el valor del exponente, representado en binario puro, complemento a 1 o complemento a 2. Los otros 3 o 7 bytes son utilizados para representar la mantisa normalizada, destinando un bit para el signo del número.
Por ejemplo, si el número a representar es el valor 23.375, en binario es $10111.011=0.10111011x2^5$. Por lo tanto, la representación en simple precisión será:

31	30 23	22 0
0	00000101	00000000000000010111011
Signo del número	Exponente	Mantisa

5.1.2 Tipos de Datos en C

Todos los tipos de datos simples o básicos de C son números. Los tres tipos de datos básicos son: enteros, números de coma flotante (reales) y caracteres.

Estos tipos de datos son atómicos o simples, lo que significa que las variables asociadas a estos tipos de datos permiten contener un solo dato por vez.

Tipo	Significado	Tamaño en Bytes	Rango
`char`	Un carácter	1	[0, 255]
`int`	Entero	2 ó 4	short [-128,127] int[-32768, 32767] unsigned int[0, 65535] long [-2147483648, 2147483637]
`float`	Real de simple precisión	4	$[-3.4^{-38}, 3.4^{38}]$
`double`	Real de doble precisión	8	$[-1.7^{-308}, 1.7^{308}]$
`void`	Sin valor	0	Sin valor
`Tipo de dato *`	Puntero	1	Contiene la dirección de memoria de un dato de tipo “Tipo de dato”

El tipo `void` es utilizado como tipo de datos asociados a aquellas funciones que no retornan valor en su nombre, es decir, para crear procedimientos. También cuando una función o procedimiento no requiere de parámetros y para crear punteros genéricos en asignación dinámica de memoria, temas que se verán más adelante.

5.1.3 Modificadores de Tipos

Los modificadores de tipo son utilizados para alterar el significado o rango de los tipos de datos básicos. Estos modificadores pueden ser:

Modificadores de longitud

- `short`: aplicable a enteros, indica entero corto, que es el formato por defecto, por lo que `int`, `short int` o `short` son sinónimos.
- `long`: aplicable a enteros, duplica el rango del tipo porque duplica la cantidad de bytes utilizados para su representación.

Modificadores de signo, aplicables a tipos de datos enteros y caracter.

- `signed`: es el tipo por defecto, por lo que `signedint` e `int`son iguales, lo mismo que `signed short int`.
- `Unsigned`: enteros sin signos.

6 Variables

Una variable es una posición de memoria representada por un identificador, que tiene asociado un tipo de dato y que almacena un dato del tipo asociado, cuyo valor puede cambiar durante la ejecución del programa.

Las variables se declaran antes de ser utilizadas. La declaración asocia un tipo de dato a un identificador de una dirección de memoria RAM. Las variables pueden ser globales cuando se declaran antes de la función `main()` o locales cuando se declaran dentro de cada subprograma, incluyendo la función `main()`.

Se debe evitar el uso de variables globales, como lo indica María Asunción Criado Clavero en el Capítulo 4 [1].

La definición de variables se de la siguiente manera,

Sintaxis:

```
tipo variable
```

donde

`tipo` es el nombre de un tipo de dato conocido por el C.

`variable` es un identificador (nombre) válido en C.

Son ejemplos de declaración de variables las siguientes:

```
float sueldo_neto, sueldo_bruto;
unsigned int edad;
```

El siguiente ejemplo ilustra la declaración de una variable de ámbito global.

Ejemplo 4:

```
1. #include<stdio.h>
2. int edad; /* variable declarada antes de main, de ámbito global * /
3. int main()
4. {
5.    printf ('¿Cuál es su edad?');
6.    scanf ("%d",&edad) ;
7.    return 0;
8. }
```

El siguiente ejemplo ilustra la declaración de variables de ámbito local, interno a la función `main()`.

Ejemplo 5:

```
1. #include <stdio.h>
2. int main()
3. {
4.    int edad; /* variable declarada dentro de main, de ámbito local a main * /
5.    printf ('¿Cuál es su edad?');
6.    scanf ("%d",&edad);
7.    return 0;
8. }
```

El siguiente ejemplo ilustra la declaración de variables de ámbito local, interno al bloque de ciclo `for`.

Ejemplo 6:

```
1.  int main()
2.  {
3.     int i;
4.     for (i=0; i<9; i++)
5.     {
6.     double suma; /*suma es una variable declarada dentro del bloque for, de ámbito
       local a este bloque*/
7.
8.        ...
9.     }
10.    ...
11. }
```

En C las declaraciones se han de situar siempre al principio del bloque. Su ámbito es el bloque en el que están declaradas. Se debe evitar la declaración de variables de bloque procurando sólo el uso de variables locales a los subprogramas a fin de aportar claridad a los programas. La inicialización de las variables refiere a la asignación de valor a cada variable. Las variables se pueden inicializar a la vez que se declaran, o bien, inicializarse después de la declaración. El siguiente ejemplo muestra una inicialización de variables al momento de la definición.

Ejemplo 7:

```
1.  int main()
2.  {
3.     char respuesta='s';
4.     int contador=0;
5.     ...
6.  }
```

Ejemplo de inicialización de variables posteriormente a su definición.

Ejemplo 8:

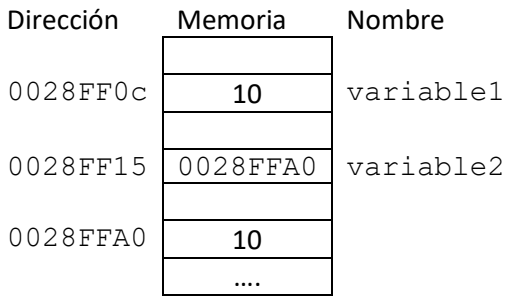
```
1.  int main()
2.  {
3.     char respuesta;
4.     int contador;
5.     respuesta='s';
6.     contador=0;
7.     ...
8.  }
```

Una variable posee por cinco atributos: nombre, dirección, tipo de valor, tiempo de vida y ámbito. Al declarar una variable de manera estática, se asocian tres atributos fundamentales: su nombre, su tipo y su dirección en memoria. Cuando la declaración es de forma estática, como se presentó en los ejemplos previos, el compilador, antes de la ejecución del programa, reserva memoria RAM suficiente para almacenar el tipo de dato asociado a esa variable. Cada vez que se ejecute el programa, la variable puede almacenarse

en un sitio diferente; dependiendo de la memoria disponible y de otros varios factores. Dependiendo del tipo de variable que se declare, se reservará más o menos memoria. Como ya vimos anteriormente cada tipo de variable ocupa más o menos bytes de memoria. Por ejemplo, si se declara un `char`, se reserva 1 byte (8 bits). Cuando finaliza el programa todo el espacio reservado queda libre.

6.1 Variables de tipo puntero

Una variable de tipo puntero tiene asociados los mismos atributos que cualquier otro tipo de variable en un programa: nombre, dirección de memoria, valor, etc. Así como la declaración `int variable1 = 10` se asocian todos los atributos de una variable, en la declaración de un puntero, por ejemplo, llamado `variable2` también se asocian tales atributos. En el primer caso se asocia el nombre `variable1` a una dirección de memoria determinada, supongamos `0028FF0c`, que permite almacenar un tipo de dato entero, en este caso `10`. En el caso de la variable de tipo puntero, se asocia un nombre, en este caso `variable2`, a una dirección de memoria que permite guardar como dato una dirección de memoria en la que, efectivamente, se almacena un dato.



Los punteros se rigen por estas reglas básicas:

- un puntero es una variable como cualquier otra;
- una variable puntero contiene una dirección que apunta a otra posición en memoria;
- en esa posición se almacenan los datos a los que apunta el puntero;

La declaración de variables de tipo puntero se rige por la siguiente sintaxis:

Sintaxis:

```
<tipo_dato> * <nombre_variable>
```

`<tipo_dato>` ***** indica que la variable `<nombre_variable>` contendrá la dirección de memoria capaz de alojar datos del tipo `<tipo_dato>`. La variable `<nombre_variable>` es un puntero a direcciones que contienen datos del tipo <tipo_dato>, por lo que no podrá alojar direcciones de memorias que contengan otros tipos de datos.

Ejemplo:

```
int * num_ent;              /*num_ent puntero a entero*/
float * num_real;           /*num_real puntero a punto flotante*/
char * caracter;            /*caracter puntero a char*/
```

Por otro lado, en C se puede declarar un puntero de modo que apunte a cualquier tipo de dato, es decir, **NO** se asigna a un tipo de dato específico. Este tipo de puntero es declarado como un puntero `void *`, denominado puntero genérico. En otras unidades de esta materia se tratará con más profundidad este tema.

Cuando se declara un puntero mediante la sentencia `<tipo_dato> * <nombre_variable_ptr>;` se crea una dirección de memoria de un byte de tamaño y de nombre `<nombre_variable_ptr>`. El dato que contiene esta variable es imprecisa, por lo que el puntero apunta a alguna dirección de memoria no determinada. Esta situación es muy arriesgada ya que se puede acceder de forma indeseada a posiciones de memoria sensibles para el correcto funcionamiento del propio programa o de otros programas, Para evitar esta situación, los punteros siempre y en todo momento deben contener direcciones de memoria válida. Una forma de inicializar la variable puntero es mediante la asignación de la dirección válida de memoria **NULL**, con lo cual, el puntero se dice que es nulo y no direcciona ningún dato de memoria. Otra forma es asignar al puntero la dirección de memoria de un dato válido, generalmente contenido en otra variable.

El operador & retorna la dirección de memoria de una variable dada. La declaración y asignación para almacenar en una variable de tipo puntero la dirección de otra variable será:

```
<tipo_dato>  <nombre_variable_dato>;

<tipo_dato> * <nombre_variable_ptr>;
```

…

```
<nombre_variable_ptr> = &<nombre_variable_dato>;
```

Ó

```
<nombre_variable_ptr> = NULL;
```

Ejemplo 9:

```
1. #include<stdio.h>
2. int main(){
3.     int num;              /*declaración de variable de datos*/
4.     int* puntEntero;      /*declaración de variable puntero*/
5.     puntEntero = &num;    /*asignación de la dirección de memoria donde se
       encuentra num*/
6.     printf("direccion de num = %p\n",&num);
7.     printf("direccion de punt_num = %p\n",puntEntero);
8. }
```

El operador * es llamado operador de indirección, es utilizado para obtener el valor al que apunta un puntero, es decir, su dato.

Ejemplo 10:

```
1.  #include<stdio.h>
2.  int main(){
3.      int num = 25;
4.      int* puntEntero;
5.      puntEntero = &num;
6.      printf("direccion de num = %p\n",&num);
7.      printf("valor de num = %d\n",num);
8.      printf("direccion de punt_num = %p\n",puntEntero);
9.      printf("valor de punt_num = %d\n",*puntEntero);
10.     return 0;
11. }
```

7 Constantes

En C existen cuatro tipos de constantes: constantes literales, constantes definidas, constantes enumeradas, constantes declaradas.

Las **constantes literales** o constantes, en general, son los valores que se utilizan tal cual en los programas. Se clasifican también en cuatro grupos, cada uno de los cuales puede ser de cualquiera de los tipos: constantes enteras, constantes caracteres, constantes de coma flotante, constantes de cadena.

Las contantes enteras no utilizan punto ni coma como delimitador. Si se desea que la constante sea de tipo `long` se coloca **`l`** o **`L`** al finalizar el valor, por ejemplo `2013L`. Si se desea que sea `unsigned` se termina con la letra **`u`**. Para constantes enteras expresadas en sistema octal se las precede de la letra **`O`** y si es en sistema hexadecimal se la precede con la combinación **`Ox`**, así una constante en hexadecimal puede ser `OxA5B`.

Las contantes reales poseen signo y son aproximaciones, por ejemplo: `12.4; .76; 1.25E7`.

Las constantes de tipo caracter son constantes de un solo caracter correspondiente a un código ascii, que se encierra entre comillas simples, por ejemplo `'A'`. Existen constantes de caracter llamados códigos de escape, los cuales pueden ser consultados en el capítulo 3 del libro de Joyanes Aguilar [2].

Las constantes de cadena es una secuencia de caracteres encerrados entre comillas dobles, por ejemplo, `"esta es una cadena"`. En memoria, las cadenas se representan por una serie de caracteres ASCII más un `\0` o nulo. El carácter nulo marca el final de la cadena y se inserta automáticamente por el compilador C al final de las constantes de cadenas.

Las **constantes definidas** o simbólicas reciben un nombre mediante la directiva **`#define`**, la cual es una directiva de preprocesador y no una sentencia, por lo que no finaliza en punto y coma, por ejemplo:

```
#define PI 3.141592
#define VALOR 54
```

El compilador reemplazará cada vez que se encuentre el identificador de la constante por el correspondiente valor, antes de analizar sintácticamente el programa.

Las **constantes enumeradas** son utilizadas para crear listas de elementos. Por ejemplo, un enumerado de días de la semana:

```
enum dias {Lunes, Martes, Miercoles, Jueves, Viernes};
```

Cuando se procesa esta sentencia, el compilador asigna un valor secuencial a cada elemento, comenzando con el valor cero para el lunes, uno para el martes, etc. Después de declarar un tipo de dato enumerado, se pueden crear variables de ese tipo, como con cualquier otro tipo de datos. Por ejemplo puede ser:

```
enum Boolean { False, True };/*asigna el valor 0 al rótulo False y el valor 1 al rótulo
True.
```

Para crear una variable de tipo lógico declarar:

```
enum Boolean Interruptor = True;
```

El calificador **`const`** permite declarar constantes asociando un tipo de datos a la misma. La sintaxis es la siguiente:

Sintaxis:

```
const tipo nombre = valor;
```

Si se omite tipo, C utiliza `int` (entero por defecto)

Un ejemplo de declaración de constantes con la cláusula `const` es el siguiente:

```
const int meses=12;
```

Es equivalente a

```
const meses=12;
```

8 Entrada y Salida de Datos

La biblioteca C proporciona facilidades para entrada y salida, para lo que todo programa deberá tener el archivo de cabecera `stdio.h` . En C la entrada y salida se lee y escribe de los dispositivos estándar de entrada y salida, se denominan `stdin` y `stdout` respectivamente. La salida, normalmente, es hacia la pantalla y la entrada se capta del teclado.

En el archivo `stdio.h` están definidas macros, constantes, variables y funciones que permiten intercambiar datos con el exterior. A continuación se muestran las más habituales y fáciles de utilizar.

8.1 La función `printf()`

Para la salida se puede utilizar la función `printf()`.

Sintaxis:

```
printf (cadena-de-control, datol, dato2, . . . )
```

dónde:

`cadena-de-control` contiene los tipos de los datos y forma de mostrarlos.
`datol, dato2 ...` son variables, constantes, datos de salida.

Un ejemplo de utilización de la función es el siguiente:

```
int i=3,j=5;
printf("%d \n %d", i,j);
```

Mostrará en pantalla dos enteros, eso indicado por los códigos `%d` en la cadena de control. Cada valor se mostrará en renglones distintos, lo cual está indicado por el código `\n` también indicado en la cadena de control. Los códigos más utilizados son:

Código	Formato
%c	Carácter
%s	Cadena de caracteres
%d, %i	Entero con signo en notación de base decimal
%u	Entero sin signo en notación de base decimal
%X	Entero sin signo en notación de base hexadecimal, usando mayúsculas para los dígitos extendidos
%x	Entero sin signo en notación de base hexadecimal, usando minúsculas para los dígitos extendidos
%o	Entero sin signo en notación de base octal
%ld	Entero largo
%h	Entero corto
%e	Real en notación científica indicando el exponente con "e"
%E	Real en notación científica indicando el exponente con "E"
%f	Real en formato de punto flotante
%lf	Real doble precisión
%g	Real con el formato más corto entre "%e" y "%f"
%p	Puntero
%n	Puntero a un entero en el cual se deposita la cantidad de caracteres escritos hasta el momento

C utiliza secuencias de escape para visualizar caracteres que no están representados por símbolos. Las secuencias de escape proporcionan flexibilidad en las aplicaciones mediante efectos especiales.

```
printf("\n Error  Pulsar una tecla para continuar \n");
```

Dos de las secuencias de escape más utilizadas son `\n` que permite imprimir en pantalla un fin de línea, produciendo como efecto, un salto de línea en pantalla y `\t` que imprime una tabulación (un salto) en pantalla.

8.2 La función `scanf()`

Para la salida se puede utilizar la función `scanf()`.

Sintaxis:

```
scanf(cadena-de-control, var1, var2, var3, ...)
```

donde

`cadena-de-control` contiene los tipos de los datos y si se desea su anchura
`var1, var2, var3, ...` variables del tipo de los códigos de control.

Los códigos de formato más comunes son los ya indicados en la salida. Se pueden añadir, como sufijo del código, ciertos modificadores como `l` o `L`. El significado es "largo", entonces, aplicado a `float (%lf)` indica tipo `double`, aplicado a `int (%ld)` indica entero largo, como ya se mostró en la tabla anterior.

Un ejemplo de utilización de esta función es el siguiente:

```
int n; double x;
scanf ("%d %lf",&n,&x);
```

Como puede observarse, las variables que se pasan a `scanf()` son precedidas por `&`, esto indica que se pasa la dirección de memoria asociada a la variable, de manera que la función `scanf()` pueda alojar en dicha dirección el dato ingresado para tal variable .

9 Operadores

9.1 El Operador de Asignación

El operador = asigna el valor de la expresión derecha a la variable situada a su izquierda.

```
codigo = 3467;
```

Este operador es asociativo por la derecha, eso permite realizar asignaciones múltiples. Así:

`a = b = c = 45;` equivale a `a = (b = (c = 45));`

Además del operador de asignación =, C proporciona cinco operadores de asignación adicionales. Estos operadores de asignación actúan como una notación abreviada para expresiones utilizadas con frecuencia. A continuación se presentan los seis operadores de asignación:

Símbolo	Uso del símbolo	Descripción
=	a=b	Asigna en la variable a el valor de la variable b
*=	a*=b	Multiplica a por b y asigna el resultado a la variable a
/=	a/=b	Divide a por b y asigna el resultado a la variable a
%=	a%=b	Calcula el resto de la división de a por b y asigna el resultado a la variable a
+=	a+=b	Suma a y b, asigna el resultado a la variable a
-=	a-=b	Resta a y b, asigna el resultado a la variable a

9.2 Operadores Aritméticos

Operador	Tipo Entero	Tipo Real	Ejemplo
+	Suma	Suma	a+b
-	Resta	Resta	a-b
*	Producto	Producto	a*b
/	Cociente de división entera	Cociente de división en punto flotante	a/b
%	Resto de división entera	----	a%b

El tipo de datos del resultado de las operaciones aritméticas dependerá del tipo de datos de los operadores participantes en la operación, por ejemplo, si en la suma al menos uno de los operandos es real, el resultado será real, si en cambio ambos operandos son enteros el resultado será de tipo entero.

La precedencia de los operadores aritméticos es la siguiente:

Operador	Ejemplo	Nivel de precedencia
+, – (operadores unitarios)	+25 (indica que 25 es positivo) -12(indica que 12 es negativo)	1º
*, /, %	5*5 es 25 7/3 es 2 7%2 es 1	2º
+, –	5+3 es 8 5-6 es -1	3º

Cuando en una expresión se presentan operadores de igual nivel de precedencia, la misma se resuelve de izquierda a derecha. Esta propiedad se llama propiedad asociativa.

9.3 Operadores de Incremento y Decremento

Los operadores ++ y -- suman o restan 1 al argumento. Estos operadores pueden utilizarse como sufijo o prefijo según el contexto, por ejemplo

`a++;` es igual a `++a;` igual a `a=a+1;`

Sin embargo,

`m=++a;(1)` es distinto a `m=a++;(2)`

Considerando por ejemplo que `a=3;` entonces, en el primer caso (1) `a` y `m` tienen el valor `4`, mientras que en el segundo caso (2), primero se asigna el valor de `a` en `m`, el `3`, y luego incrementa el valor de a tomando el valor `4`.

Si los operadores `++` y `––` son prefijos, la operación de incremento o decremento se efectúa antes que la operación de asignación; si los operadores `++` y `––` son sufijos, la asignación se efectúa en primer lugar y el incremento o decremento, a continuación.

9.4 Operadores Relacionales

C no tiene tipos de datos lógicos o booleanos, como Pascal, para representar los valores verdadero (`true`) y falso (`false`). En su lugar se utiliza el tipo `int` para este propósito, con el valor entero **0** que representa a **falso** y **distinto de cero** a **verdadero**.

Los operadores relacionales son:

Operador	Significado	Ejemplo
==	Igual a	a== b
!=	No igual a	a != b
>	Mayor que	a>b
<	Menor que	a<b
>=	Mayor o igual que	a >= b
<=	Menor o igual que	a <= b

Los operadores relacionales tienen menor prioridad que los operadores aritméticos, y asociatividad de izquierda a derecha.

Las cadenas de caracteres no pueden compararse directamente. Para una comparación alfabética entre cadenas se utiliza la función `strcmp()` de la biblioteca de C (`string.h`)

9.5 Operadores Lógicos

Los operadores lógicos de C son: not (`!`) , and (`&&`) y or (`||`)

Prioridad	Operador	Sintaxis	Ejemplo
Mayor ⇓ Menor	Negación (`!`)	No lógica	`! (x >= y)`
	Y lógica (`&&`)	`operando1 && operando2`	`m < n && i > j`
	O lógica `||`	`operando1 || operando2`	`m = 5 ||n != 10`

La asociatividad es de izquierda a derecha.

La precedencia entre distintos tipos de operadores es la siguiente

Prioridad	Operadores
Mayor ⇓ Menor	Matemáticos
	Relacionales
	Lógicos

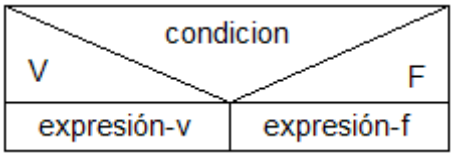
9.6 Operador Condicional

El operador condicional, **`?:`**, es un operador ternario que devuelve un resultado cuyo valor depende de la condición comprobada. Este operador se utiliza para reemplazar a la sentencia de alternativa lógica en algunas situaciones. El formato del operador condicional es:

Sintaxis:

```
c? expresión-v : expresión-f;
```

La cual equivale a



Por ejemplo:

```
(ventas> 150000) ? comision = 100 : comision = 0;
```

La precedencia de`?` y `:` es menor que la de los otros operadores. Su asociatividad es a derecha.

9.7 El operador coma

El operador coma permite combinar dos o más expresiones separadas por comas en una sola línea. Se evalúa primero la expresión de la izquierda y luego las restantes expresiones de izquierda a derecha.

Sintaxis:

```
expresión , expresión , expresión , ... , expresión
```

Por ejemplo, en

```
int i = 10, j = 25;
```

Dado que el operador coma se asocia de izquierda a derecha, la primera variable declarada e inicializada es `i` y luego se declara e inicializa `j`.

El operador coma tiene la menor prioridad de todos los operadores C, y se asocia de izquierda a derecha.

El resultado de la expresión global se determina por el valor de expresión. Por ejemplo,

```
int i, j, resultado;
resultado = j = 10, i = j, ++i;
```

El valor de esta expresión y valor asignado a resultado es `11`. En primer lugar, a `j` se asigna el valor `10`, a continuación a `i` se asigna el valor de `j`. Por último, `i` se incrementa a `11`.

La técnica del operador coma permite operaciones interesantes

```
i = 10;
j = (i = 12, i + 8);
```

Cuando se ejecute la sección de código anterior, `j` vale 20, ya que `i` vale `10` en la primera sentencia, en la segunda toma `i` el valor`12` y al sumar `i + 8`, `j` resulta `20`, mientras que `i` continúa valiendo `12`.

10 Conversiones de Tipos

Con frecuencia, se necesita convertir un valor de un tipo a otro sin cambiar el valor que representa. Las conversiones de tipos pueden ser implícitas (ejecutadas automáticamente) o explícitas (solicitadas específicamente por el programador). C hace muchas conversiones de tipos automáticamente:

- cuando se asigna un valor de un tipo a una variable de otro tipo
- cuando se combinan tipos mixtos en expresiones
- cuando se pasan argumentos a funciones

Cuando se mezclan tipos de datos en las expresiones, los operandos de tipo más bajo se convierten en los de tipo más alto.

```
int i = 12;
double x = 4;
x = x+i;      /*valor de i se convierte en double antes de la suma */
x = i/5;      /* primero hace una división entera i/5, cuyo resultado es 2.
                 Luego 2 se convierte a double, 2.0 para ser asignado a x.*/
x = 4.0;
x = x/5       /* convierte 5 a tipo double, hace una división real: 0.8 y se
                 asigna a x*/
```

La conversión también se puede hacer explícitamente, utilizando el operador cast, el cual tiene el siguiente formato:

Sintaxis:

```
(tipo_nombre)valor
```

Los siguientes son ejemplos de conversión explícita son:

```
(float) k; /*convierte k a float*/
```

```
(int) 3.14 /*convierte a entero resultando el valor 3*/
```

11 Las Sentencias

Las instrucciones o sentencias se organizan en tres tipos de estructuras de control que sirven para controlar el flujo de la ejecución: secuencia, selección (decisión) y repetición.

Las sentencias pueden ser simples o compuestas. Las sentencias simples en C finalizan con punto y coma (`;`) mientras que las sentencias compuestas son conjuntos de sentencias encerradas entre llaves (`{}`) que se utiliza para especificar un flujo secuencial.

Sintaxis:

```
{
   sentencia;
   sentencia;
   sentencia;
}
```

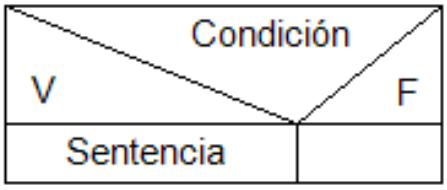
11.1 La Sentencia `if`

En C, la estructura de control de selección principal es una sentencia `if`. La sentencia `if` tiene dos formatos posibles.

El formato más sencillo es el que solo posee sentencias del lado verdadero de la sentencia.

Sintaxis:

```
if (condición) sentencia;
```



donde

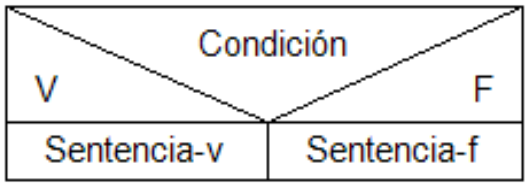
`condición` es una expresión entera(lógica).

`sentencia` es cualquier sentencia ejecutable, simple o compuesta, que se ejecutará sólo si la condición toma un valor verdadero, es decir, distinto de cero, ya que cero es entendido por C como falso y cualquier valor distinto de cero es entendido como verdadero.

El otro formato posible es el que posee sentencias tanto en el lado verdadero como en el falso de la sentencia.

Sintaxis:

```
if (condición)
  sentencia-v;
else
  sentencia-f;
```



Donde

`condición` es una expresión entera(lógica).

`sentencia-v` es cualquier sentencia ejecutable, simple o compuesta, que se ejecutará sólo si la condición es verdadera (toma un valor distinto de cero).

`sentencia-f` es cualquier sentencia ejecutable que se ejecutará en caso que la condición tome el valor falso (es decir, toma el valor cero).

Ejemplo 11:

```
1.  #include <stdio.h>
2.  int main() {
3.     int x;
4.     scanf("%d",&x);
5.     if (x % 2 ==0)
6.        printf("es par");
7.     else
8.        printf("es impar");
9.     return 0;
10. }
```

Una sentencia `if` puede ser anidada con otra, para esto la sentencia de la rama verdadera o la rama falsa, es a su vez una sentencia `if`. Una sentencia `if` anidada se puede utilizar para implementar decisiones con varias alternativas o multi-alternativas.

Sintaxis:

```
if (expresión-lógica )
      sentencia1;
else
      if (expresión-lógica )
            sentencia2;
      else
            sentencia3;
```

ó bien

```
if (expresión-lógica )
      sentencia1;
else  if (expresión-lógica )
            sentencia2;
      else
            sentencia3;
```

Ejemplo 12:

```
1.  #include <stdio.h>
2.  int main() {
3.     int x,y,z;
4.     printf("Introduzca tres números naturales");
5.     scanf("%d  %d  %d",&x,&y,&z);
6.     printf("los valores introducidos son %d\t%d\t%d\n",x,y,z);
7.     if ((x > y)&&(x > z))
8.         printf("\n%d es el mayor", x);
9.     else if (y > x && y > z)
10.           printf("\n%d es el mayor", y);
11.         else
12.             printf("\n%d es el mayor", z);
13.    return 0;
```

```
14. }
```

11.2 Sentencia de Control `switch`

La sentencia `switch` es una sentencia C que se utiliza para seleccionar una de entre múltiples alternativas. Reemplaza a las alternativas anidadas pero sólo se utiliza en el caso en que la selección se base en una variable simple o en una expresión simple denominada expresión de control o selector. El valor de esta expresión puede ser de tipo `int` o `char`, pero no de tipo `float` ni `double`.

Sintaxis:

```
switch (selector)
{
  case etiqueta, case etiqueta : sentencias;
  case etiqueta, case etiqueta : sentencias;
  case etiqueta, case etiqueta : sentencias;
  default: sentencias; /* opcional. */
}
```

Cada etiqueta debe tener un valor diferente de los otros. Si el valor de la expresión selector es igual a una de las etiquetas case entonces la ejecución comenzará con la primera sentencia de la secuencia sentencias y continuará hasta que se encuentra el final de la sentencia de control `switch`, o hasta encontrar la sentencia `break`. Es habitual que después de cada bloque de sentencias correspondiente a una secuencia se desee terminar la ejecución del `switch`; para ello se sitúa la sentencia `break` como última sentencia del bloque.

Sintaxis:

```
switch ( selector)
{
  case etiqueta, case etiqueta : sentencias;
      break;
  case etiqueta, case etiqueta : sentencias;
      break;
  case etiqueta, case etiqueta : sentencias;
      break;
  default: sentencias; /* opcional * /
}
```

Una sentencia `break` consta de la palabra reservada `break` seguida por un punto y coma. Cuando la computadora ejecuta las sentencias siguientes a una etiqueta `case`, continúa hasta que se alcanza una sentencia `break`. Si la computadora encuentra una sentencia `break`, termina la sentencia `switch`. Si se omiten las sentencias `break`, después de ejecutar el código de `case`, la computadora ejecutará el código que sigue a la siguiente `case`.

Ejemplo 13:

```
1.  #include<stdio.h>
2.  Int main() {
3.     char letra;
4.     printf("Ingrese una letra ");
5.     scanf ("%c",&letra);
6.     printf("\n letra ingresada \t %c",&letra);
```

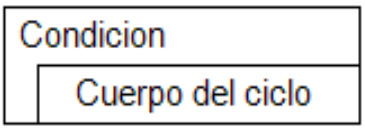
```
7.     switch (letra)
8.     {
9.        case 'a' : case 'A' : case 'á':
10.          printf("\n la letra ingresada es la primer vocal");
11.          break;
12.       case 'e' : case 'E' : case 'é':
13.          printf("\n la letra ingresada es la segunda vocal");
14.          break;
15.       case 'i': case'I': case 'í':
16.          printf("\n la letra ingresada es la tercer vocal");
17.          break;
18.       case 'o' : case 'O' : case 'ó':
19.          printf("\n la letra ingresada es la cuarta vocal");
20.          break;
21.       case 'u' : case 'U' : case 'ú':
22.          printf("\n la letra ingresada es la quinta vocal");
23.          break;
24.       default:
25.          printf("\n no es una vocal");
26.    }
27.    return 0;
28. }
```

11.3 La Sentencia `while`

Una de las estructuras de control para repetición de sentencias, es el ciclo condicionado, `while`.

Sintaxis:

```
while( condicion )
  sentencia;
```



Donde:

`condicion`	es una expresión que puede ser evaluada como verdadera o falsa
`Sentencia`	es el cuerpo del ciclo, es la sentencia simple o compuesta que se repetirá con cada iteración del ciclo

El comportamiento o funcionamiento de una sentencia (bucle) `while` es:

1. Se evalúa el valor de verdad de `condicion`
2. Si `condicion` es verdadera (distinto de cero):
   a. Las sentencias especificadas en `cuerpo-del-ciclo` se ejecutan.
   b. Vuelve el control al paso 1.
3. En caso contrario
   a. El control se transfiere a la sentencia siguiente al bucle o sentencia `while`.

Ejemplo 14:

```
1.  #include <stdio.h>
```

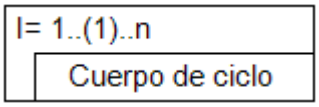
```
2.  int main() {
3.     int num, sum;
4.     sum=0;
5.     printf("Ingrese números enteros distintos de cero: ");
6.     scanf("%d",&num);
7.     while (num){
8.        sum+=num;
9.        printf("\n Ingrese otro entero, 0 para terminar: ");
10.       scanf("%d",&num);
11.    }
12.    printf("\n La suma es: %d",sum);
13.    return 0;
14. }
```

11.4 Repetición: El Bucle `for`

La estructura de control que permite la repetición una cantidad predeterminada de veces, de un conjunto de sentencias es el ciclo incondicionado:

Sintaxis:

```
for (inicializacion ; condicionIteracion ; incremento)
     sentencia;
```



Donde:

`inicializacion`	inicializa la o las variables de control del bucle. Se pueden utilizar variables de control del bucle simples o múltiples. Es decir que cada ciclo puede trabajar con más de una variable de control, iniciándolas todas al mismo tiempo
`CondicionIteracion`	contiene una expresión lógica que hace que el bucle realice las iteraciones de las sentencias, mientras que la expresión sea verdadera
`Incremento`	incrementa o decrementa la variable o variables de control del bucle
`Sentencia;`	es el cuerpo del ciclo, puede estar compuesto por una sentencia simple o compuesta. Son las acciones o sentencias que se ejecutarán por cada iteración del bucle

El siguiente ejemplo muestra la ejecución de un ciclo for:

```
for (i=0, j=5; i<3; i++, j--)
  printf ("%d      %d\n", i, j);
```

Generará como salida

0 5

1 4

2 3

Es decir, el ciclo `for` está trabajando con las variables de control `i` y `j`, incrementando la primera y decrementando la segunda.

La sentencia `for` es equivalente al siguiente código que utiliza la sentencia `while`.

```
inicialización;
while ( condiciónIteración)
{
       sentencia del cuerpo del bucle for;
       incremento;
}
```

Ejemplo 15:

```
1.  #include <stdio.h>
2.  int main() {
3.     intnum, i, cant,sum;
4.     sum=0;
5.     printf("ingrese cant de datos a procesar: ");
6.     scanf("%d",&cant);
7.     for (i=1; i<=cant; i++)
8.     {
9.        printf("\n Ingrese un núm: ");
10.       scanf("%d",&num);
11.       sum+=num;
12.    }
13.    printf("\n la suma de los datos es %d",sum);
14.    return 0;
15. }
```

Un error común es situar un punto y coma después del paréntesis inicial del bucle for. Es decir, por ejemplo:

```
for (i = 1; i <= 10; i++);
       printf("Hola") ;
```

Se espera que la palabra Hola se visualice diez veces, sin embargo sólo se visualizará una vez. Esto sucede porque el ciclo for se cerró en el punto y coma que se escribió inmediatamente después del paréntesis, por lo tanto es un ciclo vacío y la sentencia `printf()` se ejecuta una vez finalizado el ciclo, por lo tanto sólo se ejecuta una vez.

11.5 Repetición: El Bucle `do-while`

La sentencia `do-while` se utiliza para especificar un bucle condicional que se ejecuta al menos una vez.

Sintaxis:

```
do
  cuerpo-del-ciclo;
while (expresion);
```

Esta estructura de control comienza ejecutando la primera sentencia del cuerpo del ciclo, a continuación se evalúa la expresión, si la expresión es verdadera, entonces se repite la ejecución de sentencia. Este proceso continúa hasta que la expresión sea falsa.

Programa ejemplo donde se utiliza el ciclo `do-while` para controlar el ingreso de un número natural.

Ejemplo 16:

```
1.  #include <stdio.h>
2.  int main() {
3.     int x;
4.     do{
5.        printf("\nIngrese un número natural ");
6.        scanf("%d",&x);
7.     }while (x<=0);
8.     return 0;
9.  }
```

Bibliografía:

Cap. 3, 4 y 5 - Joyanes Aguilar y Zahonero - Programación en C. Metodología, algoritmos y estructura de datos. Ed. Mc Graw Hill.

Cap 2 y 4 Criado Clavero, María Asunción. Programación en lenguajes estructurados. Ed. Alfaomega/Ra-ma

ANEXO: ejercicios propuestos

Recordando la estructura simplificada de un programa en C

```
#include <stdio.h>
int main(void) {
        …
      return (0);
}
```

De ahora en adelante se incluirá en los ejercicios solo la parte dentro de las { … }

Para los ejercicios dados ejecute una prueba de escritorio y muestre las salidas de todas las variables incluidas y compare las propuestas

Ejercicio 1:

Propuesta A:
```
int a, b;
a=15;
b=a--;
a++;
```

Propuesta B:
```
int a, b;
a = 0;
b = 0;
a++;
b=a--;
```

Propuesta C:
```
int a, b;
a =  0;
b = 0;
++a;
b=--a;
```

Ejercicio 2:

Propuesta A:
```
int a, b;
char c;
a=0; b=0;
c = 64;
a = b++ +c;
```

Propuesta B:
```
int a, b;
a = 0;
b = 0;
char c;
c = 64;
a = ++b + c--;
```

Ejercicio 3:

Propuesta A:
```
int a, b;
a=2, b=a*12,  a=b--;
```

Propuesta B:
```
int a,b;
a=2;
b=a*12;
a=b--;
```

Propuesta C:
```
int a, b;
a=2;  b=a*12;  a=--b;
```

Ejercicio 4:

Propuesta A:

```
int a, b;
        b=0;
        a=2;
        do{
                printf("%d ",b);
                b += a ;
        }while(b<10);
```

Propuesta B:

```
int a;
a = 1;
        while(a <= 10){
                printf("%d\n",a);
                a += 1;
                }
```

Ejercicio 5: modificar la propuesta B del ejercicio anterior para obtener la misma salida, utilizando ciclo for y otro utilizando ciclo do - while

Ejercicio 6:

```
int a , b, c, d, e;
  a = 10, b = 1, c = 2, d = 3, e = 4;
  a = (c=e++, a= b++ + c++, 12);
```

Ejercicio 7:

```
int a , b, c, d, e;
  a = 10, b = 1, c = 2, d = 3, e = 4;
  a = (c=e++, 12, a= b++ + c++);
```

Ejercicio 8:

```
  int a , b, c, d, e;
  a = 10, b = 1, c = 2, d = 3, e = 4;
  a = c=e++, b++ + c++, 12;
```