Lenguajes de Programación.

Propósito: Conocer y aprender a realizar programas en lenguajes de programación ampliamente difundidos y de uso en la actualidad.

Profesor: Óscar Carrasco Vera (usm.oscar.carrasco@gmail.com)

Lenguajes a estudiar para el año 2016: C y PHP

Descripción: La asignatura se dividirá en 2 bloques. En el primero se estudiará la programación en lenguaje C y en el segundo bloque el lenguaje PHP.

Evaluaciones: 2 evaluaciones, uno por cada bloque.

Fechas de Evaluaciones: 10 de Mayo y 28 de Junio (Paralelo A).

12 de Mayo y 30 de Junio (Paralelo B).

Modalidad de Trabajo: En cada clase se estudiarán las herramientas que provee el lenguaje y cómo aplicarlas en situaciones tipos. Además, en cada clase se desarrollará un trabajo grupal, el cual deberá ser enviado al correo usm.oscar.carrasco@usm.cl. Estos trabajos serán utilizados como parte de la evaluación final en los siguientes casos: Situaciones conflictivas de aprobación (promedios entre 50 y 54) y/o por suspensión de actividades por motivos ajenos a los programados.

Material de Apoyo: En cada clase se entregará un apunte que permitirá tener un registro de los temas a abordar. Este material corresponde a una recolección de fragmentos de apuntes obtenidos desde internet, de libros y manuales.

LENGUAJE C

1. INTRODUCCIÓN.

C es un lenguaje de programación creado en 1972 por **Dennis M. Ritchie** en los Laboratorios Bell como resultado de la evolución del lenguaje B, el que a su vez está basado en el lenguaje BCPL.

Lenguaje C, debido a sus características, es un lenguaje muy utilizado en la implementación de sistemas operativos, como UNIX. Debido a su popularidad, es un lenguaje muy utilizado en la implementación a aplicaciones en diversos ámbitos. Se trata de un lenguaje clasificado como de medio nivel, pero que posee muchas características de bajo nivel.

La primera estandarización del lenguaje C fue en ANSI, con el estándar X3.159-1989, por lo que este fue conocido como ANSI C. Al establecerse un estándar cuya aceptación es amplia, implica que los programas creados respetándolo, se transforman en código portable entre plataformas y arquitecturas.

C++ es un lenguaje de programación diseñado a mediados de los años 1980 por Bjarne Stroustrup. La intención de su creación fue el extender al exitoso lenguaje de programación C con mecanismos que permitan la manipulación de objetos. En ese sentido, desde el punto de vista de los lenguajes orientados a objetos, el C++ es un lenguaje híbrido.

Posteriormente se añadieron facilidades de programación genérica[1], que se sumó a los otros dos paradigmas que ya estaban admitidos (programación estructurada y la programación orientada a objetos). Por esto se suele decir que el C++ es un lenguaje de programación multiparadigma.

Actualmente existe un estándar, denominado ISO C++, al que se han adherido la mayoría de los fabricantes de compiladores más modernos.

2. ESTRUCTURA DE UN PROGRAMA EN LENGUAJE C

```
O  #include <stdio.h>          // Encabezado          O
O  main()                      // Cuerpo              O
   {
O                                                     O
O  }                                                  O
```

Un programa en lenguaje C, básicamente está compuesto por 2 áreas: Encabezado y Cuerpo del Programa. En el encabezado se incorporan sentencias tales como declaración de constantes simbólicas, directivas al compilador y declaración de variables globales. El cuerpo del programa está compuesto por

[1] *La **programación genérica** es un tipo de programación que está mucho más centrada en los algoritmos que en los datos. La idea de esta forma de programar pretende generalizar las funciones utilizadas para que puedan usarse en más de una ocasión.*

la especificación de funciones que el usuario decida incorporar en su programa, teniendo presente que debe existir un función llamada ***main()***, la cual es considera como el programa principal desde donde se invocan las demás funciones.

3. REGLAS GENERALES.

- Lenguaje C distingue entre minúsculas y mayúsculas.
- Cada sentencia finaliza con un punto y coma.
- Un bloque de código son todas aquellas sentencias agrupadas que están encerradas entre paréntesis de llaves, similares a las sentencias Begin… End de Pascal.
- El concepto de procedimiento no existe en lenguaje C, todos los módulos son funciones. Lo más parecido a procedimiento son aquellos que en el encabezado comienza con la palabra ***void*** (nulo), lo cual significa que la función no devuelve valor alguno. De todas formas, aquellas funciones que sí devuelven valor, estos pueden ser utilizados o no, de esta manera se podría dar un carácter de procedimiento a una función en determinadas circunstancias en la ejecución del programa.

4. PALABRAS RESERVADAS.

En C, como en cualquier otro lenguaje, existen una serie de palabras clave (*keywords*) que el usuario no puede utilizar como identificadores (nombres de variables y/o de funciones). Estas palabras sirven para indicar al computador que realice una tarea muy determinada (desde evaluar una comparación, hasta definir el tipo de una variable) y tienen un especial significado para el compilador. El C es un lenguaje muy conciso, con muchas menos palabras clave que otros lenguajes. A continuación se presenta la lista de las 32 palabras clave del ANSI C, para las que más adelante se dará detalle de su significado (algunos compiladores añaden otras palabras clave, propias de cada uno de ellos. Es importante evitarlas como identificadores):

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

5. ENCABEZADO DEL PROGRAMA.

El encabezado del programa, permite proporcionar información al compilador.

5.1. EL PREPROCESADOR.

El preprocesador analiza el archivo fuente antes de la fase de compilación real (pre compilación), y realiza las sustituciones de macros y procesa las directivas del preprocesador. El preprocesador también elimina los comentarios.

Una directiva de preprocesador es una línea cuyo primer carácter es un `#`. A continuación se describen algunas de ellas:

5.1.1. #include

El preprocesador reemplaza la línea `#include <stdio.h>` con el archivo de cabecera del sistema con ese nombre, en este caso el archivo 'stdio.h'.

Se puede utilizar cualquier extensión para los archivos de cabecera (bibliotecas). Pero, por convención, se utiliza la extensión *.h* para los archivos de cabecera.

Algunos archivos de cabecera importantes:

a) Stdio.h

Significa "standard input-output header", es la biblioteca estándar del lenguaje C. Este archivo de cabecera contiene las definiciones de macros, constantes, declaraciones de funciones y las definiciones de tipos usados por varias operaciones estándar de entrada y salida. Por motivos de compatibilidad, el lenguaje de programación C++ (derivado de C) también tiene su propia implementación de estas funciones, que son declaradas con el archivo de cabecera cstdio.

Funciones de manipulación de ficheros	
`fclose`	Cierra un fichero a través de su puntero.
`fopen`, `freopen`, `fdopen`	Abre un fichero para lectura, para escritura/reescritura o para adición.
`remove`	Elimina un fichero.
`rename`	Cambia al fichero de nombre.
`rewind`	Coloca el indicador de posición de fichero para el stream apuntado por stream al comienzo del fichero.
`tmpfile`	Crea y abre un fichero temporal que es borrado cuando cerramos con la función fclose().

Funciones de manipulación de entradas y salidas.	
clearerr	Despeja los indicadores de final de fichero y de posición de fichero para el stream apuntado por stream al comienzo del fichero.
feof	Comprueba el indicador de final de fichero.
ferror	Comprueba el indicador de errores.
fflush	Si stream apunta a un stream de salida o de actualización cuya operación más reciente no era de entrada, la función fflushenvía cualquier dato aún sin escribir al entorno local o a ser escrito en el fichero; si no, entonces el comportamiento no está definido. Si stream es un puntero nulo, la función fflush realiza el despeje para todos los streams cuyo comportamiento está descrito anteriormente.
fgetpos	Devuelve la posición actual del fichero.
fgetc	Devuelve un carácter de un fichero.
fgets	Consigue una cadena de caracteres de un fichero.
fputc	Escribe un carácter en un fichero.
fputs	Escribe una cadena de caracteres en un fichero.
ftell	Devuelve la posición actual del fichero como número de bytes.
fseek	Sitúa el puntero de un fichero en una posición aleatoria.
fsetpos	Cambia la posición actual de un fichero.
fread	Lee diferentes tamaños de datos de un fichero.
fwrite	Envía, desde el array apuntado por puntero, hasta nmemb de elementos cuyo tamaño es especificado por tamaño. El indicador de posición de ficheros es avanzado por el número de caracteres escritos correctamente. Si existe un error, el valor resultante del indicador de posición de ficheros es indeterminado.
getc	Devuelve un carácter desde un fichero.
getchar	Igual que getc.
gets	Lee caracteres de entrada hasta que encuentra un salto de línea, y los almacena en un único argumento.
printf, fprintf, sprintf snprintf	Usados para imprimir salidas de datos.
vprintf	También utilizado para imprimir salidas.
perror	Escribe un mensaje de error a stderr.
putc	Devuelve un carácter de un fichero.
putchar, fputchar	Igual que putc(stdout).
scanf, fscanf, sscanf	Utilizado para introducir entradas.
vfscanf, vscanf, vsscanf	También utilizado para introducir entradas.

setbuf	Esta función es equivalente a la función setvbuf pasando los valores _IOFBF para modo y BUFSIZ para tamaño, o (si acumulador es un puntero nulo), con el valor _IONBF para modo.
setvbuf	Sólo puede ser usada después de que el stream apuntado por stream ha sido asociado con un fichero abierto y antes de otra operación cualquiera es llevada a cabo al stream. El argumento modo determina cómo stream será almacenado según lo siguiente: _IOFBF ocasiona la entrada/salida a ser completamente almacenado; _IOLBF ocasiona la entrada/salida a almacenar por líneas; _IONBF ocasiona la entrada/salida a no ser almacenado. Si acumulador no es un puntero nulo, el array al que es apuntado puede ser usado en vez de la acumulación adjudicada por la función setvbuf. El argumento tamaño especifica el tamaño del array.
tmpnam	Genera una cadena de caracteres que es un nombre válido para ficheros y que no es igual al nombre de un fichero existente. La función tmpnam genera una cadena diferente cada vez que es llamada, hasta un máximo de TMP_MAX veces. Si la función es llamada más veces que TMP_MAX, entonces el comportamiento de la función está definido según la implementación del compilador.
ungetc	
puts	Imprime una de cadena de caracteres.

Constantes Miembro	
EOF	Entero negativo (int) usado para indicar "fin de fichero".
BUFSIZ	Entero que indica el tamaño del buffer utilizado por la función setbuf().
FILENAME_MAX	Tamaño máximo de la cadena de caracteres que contienen el nombre de un fichero para ser abierto
FOPEN_MAX	Número máximo de ficheros que pueden estar abiertos simultáneamente.
_IOFBF	Abreviatura de input/output fully buffered (buffer entrada/salida totalmente lleno); es un entero que se puede pasar como parámetro de la función setvbuf() para requerir bloqueo del buffer en la entrada y salida del stream abierto.
_IOLBF	Abreviatura de input/output line buffered (...??); es un entero que se puede pasar como parámetro a la funciónsetvbuf() para requerir line buffered (??) en la entrada y salida del stream abierto.
_IONBF	Abreviatura de "input/output not buffered" (entrada/salida sin buffer); es un entero que se puede pasar como parámetro a la función setvbuf() para requerir que la entrada salida del stream abierto funcione sin buffer.
L_tmpnam	Tamaño de la cadena de caracteres con la longitud suficiente para almacenar un nombre de fichero temporal generado por la función tmpnam().
NULL	Macro que representa la constante puntero nulo; representa un valor de puntero que no apunta a ninguna dirección válida de objeto alguno en memoria.
SEEK_CUR	Entero que se puede pasar como parámetro a la función fseek() para indicar posicionamiento relativo a la posición actual del fichero.
SEEK_END	Entero que se puede pasar como parámetro a la función fseek() para indicar posicionamiento relativo al final del fichero.
SEEK_SET	Entero que se puede pasar como parámetro a la función fseek() para indicar posicionamiento relativo al inicio del fichero.
TMP_MAX	El número máximo de nombres de ficheros únicos generables por la función tmpnam().

Variables Miembro	
stdin	Puntero a FILE que referencia la entrada estándar, normalmente el teclado.
stdout	Puntero a FILE que referencia la salida estándar, normalmente el monitor.
stderr	Puntero a FILE que referencia la salida de error estándar, normalmente el monitor.

Tipos Miembro
▪ FILE - estructura que contiene información sobre el fichero o stream (*flujo*) de texto necesario para realizar las operaciones de entrada/salida sobre él. incluye: ▪ posición actual de stream ▪ indicador de fin de fichero (EOF) ▪ indicador de error ▪ puntero al buffer del stream
▪ fpos_t - tipo *no-array* capaz de identificar unívocamente la posición de cada byte en un archivo
▪ size_t - tipo entero sin signo (positivo); es el tipo devuelto por el operador sizeof

b) Asser.h:

Contiene macros así como información para añadir diagnósticos que ayudan a la depuración de programas.

c) Ctype.h:

Contiene prototipos de función para funciones que prueban caracteres en relación con ciertas propiedades, y prototipos de funciones que pueden ser utilizadas para la conversación de minúsculas a mayúsculas y viceversa.

d) Errno.h:

Define macros que son útiles para la información de condiciones de error.

e) Float.h:

Contiene los límites de tamaño de punto flotante del sistema.

f) Limits.h:

Incluye los límites de tamaño integral del sistema.

g) Locale.h:

Contiene los prototipos de función y otra información que le permite a un programa ser modificado en relación con la localización actual en la cual se está ejecutando. La noción de localización le permite al sistema de cómputo manejar diferentes reglas convencionales para la expresión de datos como son fechas, horas, monedas y números grandes en diferentes áreas del mundo.

h) Math.h:

Contiene prototipos para funciones matemáticas de biblioteca.

i) Setjmp.h:

Contiene prototipos para funciones que permiten pasar por alto la secuencia usual de llamadas de función y regreso.

j) Signal.h:

Contiene prototipos de función y macros para manejar varias condiciones que pudieran ocurrir durante la ejecución de un programa.

k) Stdlib.h:

Contiene prototipos de función para las conservaciones de números a texto y de texto a números, para la asignación de memoria, para números aleatorios y otras funciones de utilería.

l) String.h:

Contiene prototipos para las funciones de procesamiento de cadenas.

m) Time.h:

Contiene prototipos de función y tipos para manipular hora y fecha.

5.1.2. #define

```
#define identificador_de_macro <secuencia>
```

La directiva #define, sirve para definir macros. Las macros suministran un sistema para la sustitución de palabras, con y sin parámetros.

El preprocesador sustituirá cada ocurrencia del *identificador_de_macro* en el fichero fuente, por la *secuencia*. Cada sustitución se conoce como una expansión de la macro. La secuencia es llamada a menudo cuerpo de la macro.

Si no se especifica una *secuencia*, el *identificador_de_macro* sencillamente, será eliminado cada vez que aparezca en el fichero fuente.

Después de cada expansión individual, se vuelve a examinar el texto expandido a la búsqueda de nuevas macros, que serán expandidas a su vez. Esto permite la posibilidad de hacer macros anidadas.

Existen algunas restricciones a la expansión de macros:

1. Si la nueva expansión tiene la forma de una directiva de preprocesador, no será reconocida como tal.
2. Las ocurrencias de macros dentro de literales, cadenas, constantes alfanuméricas o comentarios no serán expandidas.
3. Una macro no será expandida durante su propia expansión, así **`#define A A`**, no será expandida indefinidamente.
4. No es necesario añadir un punto y coma para terminar una directiva de preprocesador. Cualquier carácter que se encuentre en una secuencia de macro, incluido el punto y coma, aparecerá en la expansión de la macro. La secuencia termina en el primer retorno de línea encontrado. Las secuencias de espacios o comentarios en la secuencia, se expandirán como un único espacio.

Ejemplo:

```
#define PI 3.1514
#define MIN(a,b) ((a)>(b)?(b):(a))
#define RADAGRA(x) ((x) * 57.29578)
#define ENTRECOMILLAR(x) #x

main()
{ int a;
  a = MIN(5,6);                       // se expandirá como a = ((5)>(6)?(6):(5))
  printf("%s\n", ENTRECOMILLAR(1+2)); // se expandirá como printf("%s\n", "1+2");
  printf("%f\n", RADAGRA(3.4));       // se expandirá como printf("%f\n", (3.4) * 57.29578);
}
```

El primer #define actúa como una declaración de una constante simbólica, en donde en el precompilador cambiará cada palabra PI por el valor asociado.

El segundo #define actúa de una manera un poco más compleja, en donde el precompilador cambiará cada palabra MIN, la cual será reemplazada por la expresión condicional en línea **((a)>(b)?(b):(a))**. Esta expresión es evaluada **(a)>(b)** de tal forma que si es verdadera (el ? simboliza un entonces) el resultado será el valor de **(b)**, de lo contrario (el : simboliza sino) será el valor de **(a)**.

#define permite, además, algo muy interesante: Programar en otro idioma, por ejemplo:

```
#define SI if
#define SINO else
#define INICIO {
#define FIN }

int main()
INICIO                              // cambiaría a:   {
  SI(a > b)                         // cambiaría a:     if(a > b)
    INICIO                          // cambiaría a:       {
     printf("a es mayor que b");    // se mantiene:        printf("a es mayor que b");
    FIN                             // cambiaría a:       }
  SINO{                             // cambiaría a:     else
    INICIO                          // cambiaría a:       {
     printf("a es menor que b");    // se mantiene:        printf("a es menor que b");
    FIN                             // cambiaría a:       }
FIN                                 // cambiaría a:  }
```

Las macros pueden extenderse tantas líneas como hagan falta. Para ello tan solo habrá que terminar cada línea con un BackSlash (\). La macro terminará en la última línea sin una raya a la izquierda.

5.1.3. #undef

```
#undef identificador_de_macro
```

Sirve para eliminar definiciones de macros previamente definidas. La definición de la macro se olvida y el identificador queda indefinido.

5.2. DIRECTIVAS #ifdef... #else... #endif e #ifndef... #else... #endif

Estas directivas permiten comprobar si un identificador está o no actualmente definido, es decir, si un **#define** ha sido previamente procesado para el identificador y si sigue definido.

Ejemplo:

```
#define __WINDOWS__

#ifdef __WINDOWS__
   #include <windows.h>
#else
   #include <unistd.h>
#endif
```

5.3. DIRECTIVA #error

Esta directiva se suele incluir en sentencias condicionales de preprocesador para detectar condiciones no deseadas durante la compilación. En un funcionamiento normal estas condiciones serán falsas, pero cuando la condición es verdadera, es preferible que el compilador muestre un mensaje de error y detenga la fase de compilación. Para hacer esto se debe introducir esta directiva en una sentencia condicional que detecte el caso no deseado.

```
#error mensaje_de_error
```

Ejemplo:

```
#ifndef BFD_HOST_64_BIT
   #error No 64 bit integer type available
#endif
```

5.4. #if, #elif, #else y #endif

Permiten hacer una compilación condicional de un conjunto de líneas de código.

```
#if expresión-constante-1
<sección-1>
#elif <expresión-constante-2>
<sección-2>
…#e
lif <expresión-constante-n>
<sección-n>
<#else>
<sección-final>
#endif
```

Todas las directivas condicionales deben completarse dentro del mismo fichero. Sólo se compilarán las líneas que estén dentro de las secciones que cumplan la condición de la expresión constante correspondiente.

Estas directivas funcionan de modo similar a los operadores condicionales C++. Si el resultado de evaluar la expresión-constante-1, que puede ser una macro, es distinto de cero (true), las líneas representadas por sección-1, ya sean líneas de comandos, macros o incluso nada, serán compiladas. En caso contrario, si el resultado de la evaluación de la expresión-constante-1, es cero (false), la sección-1 será ignorada, no se expandirán macros ni se compilará.

En el caso de ser distinto de cero, después de que la sección-1 sea preprocesada, el control pasa al **#endif** correspondiente, con lo que termina la secuencia condicional. En el caso de ser cero, el control pasa al siguiente línea **#elif**, si existe, donde se evaluará la expresión-constante-2. Si el resultado es distinto de cero, se procesará la sección-2, y después el control pasa al correspondiente **#endif**. Por el contrario, si el resultado de la expresión-constante-2 es cero, el control pasa al siguiente **#elif**, y así sucesivamente, hasta que se encuentre un **#else** o un **#endif**. El **#else**, que es opcional, se usa como una condición alternativa para el caso en que todas la condiciones anteriores resulten falsas. El **#endif** termina la secuencia condicional.

Cada sección procesada puede contener a su vez directivas condicionales, anidadas hasta cualquier nivel, cada **#if** debe corresponderse con el **#endif** más cercano.

El objetivo de una red de este tipo es que sólo una sección, aunque se trate de una sección vacía, sea compilada. Las secciones ignoradas sólo son relevantes para evaluar las condiciones anidadas, es decir asociar cada **#if** con su **#endif**.

Las expresiones constantes deben poder ser evaluadas como valores enteros.

6. TIPOS DE DATOS, VARIABLES Y CONSTANTES.

Una característica de C++, es la necesidad de declarar las variables que se usarán en un programa. Esto resulta chocante para los que se aproximan al C++ desde otros lenguajes de programación en los que las variables de crean automáticamente la primera vez que se usan, como en Visual Basic o Visual Foxpro. Se trata, es cierto, de una característica de bajo nivel, más cercana al ensamblador que a lenguajes de alto nivel, pero en realidad una característica muy importante y útil de C++, ya que ayuda a conseguir códigos más compactos y eficaces, y contribuye a facilitar la depuración y la detección y corrección de errores y a mantener un estilo de programación elegante.

Uno de los errores más comunes en lenguajes en los que las variables se crean de forma automática se produce al cometer errores ortográficos. Por ejemplo, en un programa usamos una variable llamada *prueba*, y en un punto determinado le asignamos un nuevo valor, pero nos equivocamos y escribimos *prubea*. El compilador o intérprete no detecta el error, simplemente crea una nueva variable, y continúa como si todo estuviese bien.

En C++ esto no puede pasar, ya que antes de usar cualquier variable es necesario declararla, y si por error usamos una variable que no ha sido declarada, se producirá un error de compilación.

6.1. TIPOS DE DATOS.

C++ tiene los siguientes tipos fundamentales:

1. Caracteres: `char` (también es un entero)
2. Enteros: short, int, long, long long
3. Números en coma flotante: float, double, long double
4. Booleanos: bool
5. Vacío: void

El modificador `unsigned` se puede aplicar a enteros para obtener números sin signo (por omisión los enteros contienen signo), con lo que se consigue un rango mayor de números naturales.

Según la máquina y el compilador que se utilice los tipos primitivos pueden ocupar un determinado tamaño en memoria. La siguiente lista ilustra el número de bits que ocupan los distintos tipos primitivos en la arquitectura x86.

- char: 8 bits (1 Byte)
- short: 16 bits (2 Bytes)
- int: 32 bits (4 Bytes)
- float: 32 bits (4 Bytes)
- double: 64 bits (8 Bytes)

Otras arquitecturas pueden requerir distintos tamaños de tipos de datos primitivos. C++ no dice nada acerca de cuál es el número de bits en un byte, ni del tamaño de estos tipos; más bien, ofrece solamente las siguientes "garantías de tipos":

- Un tipo `char` tiene el tamaño mínimo en *bytes* asignable por la máquina, y todos los bits de este espacio deben ser "accesibles".
- El tamaño reconocido de `char` es de 1. Es decir, `sizeof(char)` siempre devuelve 1.
- Un tipo `short` tiene *al menos el mismo* tamaño que un tipo `char`.
- Un tipo `long` tiene *al menos el doble* tamaño en bytes que un tipo `short`.
- Un tipo `int` tiene un tamaño entre el de `short` y el de `long`, ambos inclusive, preferentemente el tamaño de un apuntador de memoria de la máquina.
- Un tipo `unsigned` tiene el mismo tamaño que su versión `signed`.

No se pueden declarar variables de tipo **void**, ya que tal cosa no tiene sentido. Void es un tipo especial que indica la ausencia de tipo. Se usa para indicar el tipo del valor de retorno en funciones que no devuelven ningún valor, y también para indicar la ausencia de parámetros en funciones que no los requieren, (aunque este uso sólo es obligatorio en C, y opcional en C++), también se usará en la declaración de punteros genéricos.

Las funciones que no devuelven valores parecen una contradicción. En lenguajes como Pascal, estas funciones se llaman procedimientos. Simplemente hacen su trabajo, y no revuelven valores. Por ejemplo, una función que se encargue de borrar la pantalla, no tienen nada que devolver, hace su trabajo y regresa. Lo mismo se aplica a las funciones sin parámetros de entrada, el mismo ejemplo de la función para borrar la pantalla no requiere ninguna entrada para poder realizar su cometido.

6.2. DECLARACIÓN DE VARIABLES.

Para declarar una variable, se emplea la siguiente sintaxis:

```
[signed|unsigned] <Tipo> <Variable(s)>
```

Ejemplo:

```
unsigned int edad;
int Cantidad, Cont, Hijos;
int a = 1234;
bool seguir = true, encontrado;
```

Como se puede apreciar, las variables se pueden declarar y simultáneamente asignar un valor predeterminado o de inicio.

El modificador signed/unsigned permite establecer el rango de valores posibles a almacenar en la variable, ya sea incluyendo negativos o solamente aceptado positivos.

La omisión de la palabra unsigned, implica el rango de valores incluye los negativos.

6.3. ALCANCE.

Dependiendo de dónde se declaran las variables, estas tendrán un alcance Global o Local.

Si las variables son declaradas en el área de Encabezados, serán consideradas como globales. En cambio, si se declaran dentro de una función, serán consideradas como locales.

6.4. CONSTANTES.

Existen 2 formas de declarar constantes:

1) Mediante la definición de una constante simbólica como directiva de preprocesador, haciendo uso de #define.
2) Mediante la declaración explícita de una constante almacenada haciendo uso de la sentencia const. Por ejemplo:

```
const double PI = 3,1415;
```

7. OPERADORES.

Operadores Aritméticos	
Nombre del operador	**Sintaxis**
Más unitario	`+a`
Suma	`a + b`
Preincremento	`++a`
Postincremento	`a++`
Asignación con suma	`a += b`
Menos unitario (negación)	`-a`
Resta	`a - b`
Predecremento	`--a`
Postdecremento	`a--`
Asignación con resta	`a -= b`
Multiplicación	`a * b`
Asignación con multiplicación	`a *= b`
División	`a / b`
Asignación con división	`a /= b`
Módulo (Resto)	`a % b`
Asignación con módulo	`a %= b`

Operadores de Comparación	
Nombre del operador	**Sintaxis**
Menor que	`a < b`
Menor o igual que	`a <= b`
Mayor que	`a > b`
Mayor o igual que	`a >= b`
No igual que	`a != b`
Igual que	`a == b`
Negación lógica	`!a`
AND lógico	`a && b`
OR lógico	`a || b`

Otros Operadores	
Nombre del operador	**Sintaxis**
Asignación básica	`a = b`
Llamada a función	`a()`
Índice de Array	`a[b]`
Indirección (Desreferencia)	`*a`
Dirección de (Referencia)	`&a`
Miembro de puntero	`a->b`
Miembro	`a.b`
Desreferencia a miembro por puntero	`a->*b`
Desreferencia a miembro por objeto	`a.*b`
Conversión de tipo	`(tipo) a`
Coma	`a , b`
Condicional ternario	`a ? b : c`
Resolución de ámbito	`a::b`
Puntero a función miembro	`a::*b`

8. ESTRUCTURAS DE CONTROL.

En principio, las sentencias de un programa en C se ejecutan *secuencialmente*, esto es, cada una a continuación de la anterior empezando por la primera y acabando por la última. El lenguaje C dispone de varias sentencias para modificar este flujo secuencial de la ejecución.

Las más utilizadas se agrupan en dos familias: las ***bifurcaciones***, que permiten elegir entre dos o más opciones según ciertas condiciones, y los ***bucles***, que permiten ejecutar repetidamente un conjunto de instrucciones tantas veces como se desee, cambiando o actualizando ciertos valores.

8.1. OPERADOR CONDICIONAL

El operador condicional es un operador con tres operandos que tiene la siguiente forma general:

```
expresion_1 ? expresion_2 : expresion_3;
```

Explicación: Se evalúa **expresion_1**. Si el resultado de dicha evaluación es ***true*** (#0), se ejecuta **expresion_2**; si el resultado es ***false*** (=0), se ejecuta **expresion_3**.

8.1.1. SENTENCIA IF

Esta sentencia de control permite ejecutar o no una sentencia simple o compuesta según se cumpla o no una determinada condición. Esta sentencia tiene la siguiente forma general:

```
if (expresion)
   sentencia;
```

Explicación: Se evalúa **expresión**. Si el resultado es ***true*** (#0), se ejecuta **sentencia**; si el resultado es ***false*** (=0), se salta **sentencia** y se prosigue en la línea siguiente. Hay que recordar que **sentencia** puede ser una sentencia simple o compuesta (*bloque* { ... }).

8.1.2. SENTENCIA IF ... ELSE

Esta sentencia permite realizar una *bifurcación*, ejecutando una parte u otra del programa según se cumpla o no una cierta condición. La forma general es la siguiente:

```
if (expresion)
   sentencia_1;
else
   sentencia_2;
```

Explicación: Se evalúa **expresion**. Si el resultado es ***true*** (#0), se ejecuta **sentencia_1** y se prosigue en la línea siguiente a **sentencia_2**; si el resultado es ***false*** (=0), se salta **sentencia_1**, se ejecuta **sentencia_2** y se prosigue en la línea siguiente. Hay que indicar aquí también que **sentencia_1** y **sentencia_2** pueden ser sentencias simples o compuestas (*bloques* { ... }).

8.1.2.1. SENTENCIA IF ... ELSE MÚLTIPLE

Esta sentencia permite realizar una ramificación múltiple, ejecutando *una* entre varias partes del programa según se cumpla *una* entre *n* condiciones. La forma general es la siguiente:

```
if (expresion_1)
   sentencia_1;
else if (expresion_2)
        sentencia_2;
     else if (expresion_3)
             sentencia_3;
          else if (...)
                  ...
               [else
                   sentencia_n;]
```

Explicación: Se evalúa **expresion_1**. Si el resultado es ***true***, se ejecuta **sentencia_1**. Si el resultado es ***false***, se salta **sentencia_1** y se evalúa **expresion_2**. Si el resultado es ***true*** se ejecuta **sentencia_2**, mientras que si es ***false*** se evalúa **expresion_3** y así sucesivamente. Si ninguna de las

expresiones o condiciones es ***true*** se ejecuta **expresion_n** que es la opción por defecto (puede ser la sentencia vacía, y en ese caso puede eliminarse junto con la palabra **else**). Todas las sentencias pueden ser simples o compuestas.

8.1.3. SENTENCIA SWITCH

La sentencia que se va a describir a continuación desarrolla una función similar a la de la sentencia ***if ... else*** con múltiples ramificaciones, aunque como se puede ver presenta también importantes diferencias. La forma general de la sentencia ***switch*** es la siguiente:

```
switch (expresion)
{
       case expresion_cte_1:
            sentencia_1;
       case expresion_cte_2:
            sentencia_2;
            ...
       case expresion_cte_n:
            sentencia_n;
      [default:
            sentencia;]
}
```

Explicación: Se evalúa **expresion** y se considera el resultado de dicha evaluación. Si dicho resultado coincide con el valor constante **expresion_cte_1**, se ejecuta **sentencia_1** seguida de **sentencia_2**, **sentencia_3**, ..., **sentencia**. Si el resultado coincide con el valor constante **expresion_cte_2**, se ejecuta **sentencia_2** seguida de **sentencia_3**, ..., **sentencia**. En general, se ejecutan todas aquellas sentencias que están a continuación de la **expresion_cte** cuyo valor coincide con el resultado calculado al principio. Si ninguna **expresion_cte** coincide se ejecuta la **sentencia** que está a continuación de **default**. Si se desea ejecutar únicamente una **sentencia_i** (y no todo un conjunto de ellas), basta poner una sentencia **break** a continuación (en algunos casos puede utilizarse la sentencia **return** o la función **exit()**). El efecto de la sentencia **break** es dar por terminada la ejecución de la sentencia **switch**. Existe también la posibilidad de ejecutar la misma **sentencia_i** para varios valores del resultado de **expresion**, poniendo varios **case expresion_cte** seguidos.

El siguiente ejemplo ilustra las posibilidades citadas:

```
switch (expresion)
{
   case expresion_cte_1:
        sentencia_1;
        break;
   case expresion_cte_2:
   case expresion_cte_3:
        sentencia_2;
        break;
   default:
        sentencia_3;
}
```

8.1.4. SENTENCIAS IF ANIDADAS

Una sentencia ***if*** puede incluir otros ***if*** dentro de la parte correspondiente a su **sentencia**, A estas sentencias se les llama ***sentencias anidadas*** (una dentro de otra), por ejemplo,

```
if (a >= b)
   if (b != 0.0)
       c = a/b;
```

En ocasiones pueden aparecer dificultades de interpretación con sentencias ***if...else*** anidadas, como en el caso siguiente:

```
if (a >= b)
if (b != 0.0)
c = a/b;
else
c = 0.0;
```

Al aplicar sangría tendríamos 2 posibilidades:

```
if (a >= b)
   if (b != 0.0)
       c = a/b;
   else
       c = 0.0;
```

```
if (a >= b)
   if (b != 0.0)
       c = a/b;
else
   c = 0.0;
```

¿Cuál de las dos sería correcta?

En principio se podría plantear la duda de a cuál de los dos ***if*** corresponde la parte ***else*** del programa. Los espacios en blanco –las *indentaciones* de las líneas– parecen indicar que la sentencia que sigue a ***else*** corresponde al segundo de los ***if***, y así es en realidad, pues la regla es que el ***else*** pertenece al ***if*** más cercano. Sin embargo, no se olvide que el compilador de C no considera los espacios en blanco (aunque sea muy conveniente introducirlos para hacer más claro y legible el programa), y que si se quisiera que el ***else*** perteneciera al primero de los ***if*** no bastaría cambiar los espacios en blanco, sino que habría que utilizar *llaves*, en la forma:

```
if (a >= b)
   {
```

```
     if (b != 0.0)
         c = a/b;
   }
else
   c = 0.0;
```

Recuérdese que todas las sentencias ***if*** e ***if...else***, equivalen a una única sentencia por la posición que ocupan en el programa.

8.2. BUCLES

Además de ***bifurcaciones***, en el lenguaje C existen también varias sentencias que permiten repetir una serie de veces la ejecución de unas líneas de código. Esta repetición se realiza, bien un número determinado de veces, bien hasta que se cumpla una determinada condición de tipo lógico o aritmético. De modo genérico, a estas sentencias se les denomina ***bucles***. Las tres construcciones del lenguaje C para realizar bucles son el ***while***, el ***for*** y el ***do...while***.

8.2.1. SENTENCIA WHILE

Esta sentencia permite ejecutar repetidamente, *mientras se cumpla una determinada condición*, una sentencia o bloque de sentencias. La forma general es como sigue:

```
while (expresion_de_control)
       sentencia;
```

Explicación: Se evalúa expresion_de_control y si el resultado es *false* se salta sentencia y se prosigue la ejecución. Si el resultado es *true* se ejecuta sentencia y se vuelve a evaluar expresion_de_control (evidentemente alguna variable de las que intervienen en expresion_de_control habrá tenido que ser modificada, pues si no el *bucle* continuaría indefinidamente). La ejecución de sentencia prosigue hasta que expresion_de_control se hace *false*, en cuyo caso la ejecución continúa en la línea siguiente a sentencia. En otras palabras, sentencia se ejecuta repetidamente mientras expresion_de_control sea *true*, y se deja de ejecutar cuando expresion_de_control se hace *false*. Obsérvese que en este caso el *control* para decidir si se sale o no del *bucle* está antes de sentencia, por lo que es posible que sentencia no se llegue a ejecutar ni una sola vez.

8.2.2. SENTENCIA FOR

For es quizás el tipo de bucle más versátil y utilizado del lenguaje C. Su forma general es la siguiente:

```
for (inicializacion; expresion_de_control; actualizacion)
    sentencia(s);
```

Explicación: Posiblemente la forma más sencilla de explicar la sentencia *for* sea utilizando la construcción *while* que sería equivalente. Dicha construcción es la siguiente:

```
inicializacion;
while (expresion_de_control)
{
  sentencia;
  actualizacion;
}
```

donde **sentencia** puede ser una única sentencia terminada con (;), otra sentencia de control ocupando varias líneas (***if***, ***while***, ***for***, ...), o una sentencia compuesta o un bloque encerrado entre llaves {...}. Antes de iniciarse el bucle se ejecuta ***inicialización***, que es una o más sentencias que asignan valores iniciales a ciertas variables o contadores. A continuación se evalúa **expresión_de_control** y si es ***false*** se prosigue en la sentencia siguiente a la construcción ***for***; si es ***true*** se ejecutan **sentencia** y ***actualización***, y se vuelve a evaluar **expresión_de_control**. El proceso prosigue hasta que **expresión_de_control** sea ***false***. La parte de ***actualización*** sirve para actualizar variables o incrementar contadores. Un ejemplo típico puede ser el producto escalar de dos vectores ***a*** y ***b*** de dimensión ***n***:

```
for (pe =0.0, i=1; i<=n; i++)
{
  pe += a[i]*b[i];
}
```

Primeramente se inicializa la variable **pe** a cero y la variable **i** a 1; el ciclo se repetirá mientras que i sea menor o igual que **n**, y al final de cada ciclo el valor de **i** se incrementará en una unidad. En total, el bucle se repetirá **n** veces. La ventaja de la construcción ***for*** sobre la construcción ***while*** equivalente está en que en la cabecera de la construcción ***for*** se tiene toda la información sobre cómo se inicializan, controlan y actualizan las variables del bucle. Obsérvese que la ***inicialización*** consta de dos sentencias separadas por el operador (,).

8.2.3. SENTENCIA DO ... WHILE

Esta sentencia funciona de modo análogo a ***while***, con la diferencia de que la evaluación de **expresión_de_control** se realiza al final del bucle, después de haber ejecutado al menos una vez las sentencias entre llaves; éstas se vuelven a ejecutar mientras **expresión_de_control** sea ***true***. La forma general de esta sentencia es:

```
do
   sentencia;
while(expresion_de_control);
```

donde **sentencia** puede ser una única sentencia o un bloque, y en la que debe observarse que *hay que poner (;) a continuación del paréntesis* que encierra a **expresión_de_control**, entre otros motivos para que esa línea se distinga de una sentencia ***while*** ordinaria.

8.2.4. Sentencias break, continue, goto

La instrucción ***break*** interrumpe la ejecución del bucle donde se ha incluido, haciendo al programa salir de él aunque la **expresión_de_control** correspondiente a ese bucle sea verdadera.

La sentencia ***continue*** hace que el programa comience el siguiente ciclo del bucle donde se halla, aunque no haya llegado al final de las sentencia compuesta o bloque.

La sentencia ***goto*** *etiqueta* hace saltar al programa a la sentencia donde se haya escrito la *etiqueta* correspondiente. Por ejemplo:

```
sentencias ...
...
if (condicion)
   goto otro_lugar; // salto al lugar indicado por la etiqueta
   sentencia_1;
   sentencia_2;
   ...
otro_lugar: // esta es la sentencia a la que se salta
   sentencia_3;
   ...
```

Obsérvese que la *etiqueta* termina con el carácter (:). La sentencia ***goto*** no es una sentencia muy prestigiada en el mundo de los programadores de C, pues disminuye la claridad y legibilidad del código. Fue introducida en el lenguaje por motivos de compatibilidad con antiguos hábitos de programación, y siempre puede ser sustituida por otras construcciones más claras y estructuradas.

9. FUNCIONES.

9.1. FUNCIONES DE ENTRADA/SALIDA

A diferencia de otros lenguajes, *C no dispone de sentencias de entrada/salida*. En su lugar se utilizan funciones contenidas en la librería estándar y que forman parte integrante del lenguaje.

Las funciones de entrada/salida (Input/Output) son un conjunto de funciones, incluidas con el compilador, que permiten a un programa recibir y enviar datos al exterior. Para su utilización es necesario incluir, al comienzo del programa, el archivo **stdio.h** en el que están definidos sus prototipos:

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

donde stdio proviene de standard-input-output.

9.1.1. Función printf()

La función **printf()** imprime en la unidad de salida (el monitor, por defecto), el texto, y las constantes y variables que se indiquen. La forma general de esta función se puede estudiar viendo su *prototipo*:

```
int printf("cadena_de_control", tipo arg1, tipo arg2, ...)
```

Explicación: La función **printf()** imprime el texto contenido en **cadena_de_control** junto con el valor de los otros argumentos, de acuerdo con los *formatos* incluidos en **cadena_de_control**. Los puntos suspensivos (...) indican que puede haber un número variable de argumentos. Cada formato comienza con el carácter (%) y termina con un *carácter de conversión*.

Considérese el ejemplo siguiente,

```
int i;
double tiempo;
float masa;
printf("Resultado n°: %d. En el instante %lf la masa vale %f\n", i, tiempo, masa);
```

en el que se imprimen 3 variables (**i**, **tiempo** y **masa**) con los formatos (**%d**, **%lf** y **%f**), correspondientes a los tipos (***int***, ***double*** y ***float***), respectivamente. La cadena de control se imprime con el valor de cada variable intercalado en el lugar del formato correspondiente.

Caracteres de conversión para la función printf().

Carácter	Tipo de argumento	Carácter	Tipo de argumento
d, i	int decimal	o	octal unsigned
u	int unsigned	x, X	hex. unsigned
c	char	s	cadena de char
f	float notación decimal	e, g	float not. científ. o breve
p	puntero (void *)		

9.1.2. Función scanf()

La función **scanf()** es análoga en muchos aspectos a **printf()**, y se utiliza para leer datos de la entrada estándar (que por defecto es el teclado). La forma general de esta función es la siguiente:

```
int scanf("%x1%x2...", &arg1, &arg2, ...);
```

donde x1, x2, ... son los caracteres de conversión, mostrados en la Tabla 8.2, que representan los formatos con los que se espera encontrar los datos. La función scanf() devuelve como valor de retorno el número de conversiones de formato realizadas con éxito. La cadena de control de scanf() puede contener caracteres además de formatos. Dichos caracteres se utilizan para tratar de detectar la presencia de caracteres idénticos en la entrada por teclado. Si lo que se desea es leer variables numéricas, esta posibilidad tiene escaso interés. A veces hay que comenzar la cadena de control con un espacio en blanco para que la conversión de formatos se realice correctamente.

En la función **scanf()** los argumentos que siguen a la **cadena_de_control** deben ser **pasados por referencia**, ya que la función los lee y tiene que trasmitirlos al programa que la ha llamado. Para ello, dichos argumentos deben estar constituidos por las direcciones de las variables en las que hay que depositar los datos, y no por las propias variables. Una excepción son las cadenas de caracteres, cuyo nombre es ya de por sí una dirección (un puntero), y por tanto no debe ir precedido por el operador (&) en la llamada.

carácter	caracteres leídos	argumento
c	cualquier carácter	char *
d, i	entero decimal con signo	int *
u	entero decimal sin signo	unsigned int
o	entero octal	unsigned int
x, X	entero hexadecimal	unsigned int
e, E, f, g, G	número de punto flotante	float
s	cadena de caracteres sin ''	char
h, l	para short, long y double	
L	modificador para long double	

Por ejemplo, para leer los valores de dos variables *int* y *double* y de una cadena de caracteres, se utilizarían la sentencia:

```
int n;
double distancia;
char nombre[20];
scanf("%d%lf%s", &n, &distancia, nombre);
```

en la que se establece una correspondencia entre **n** y **%d**, entre **distancia** y **%lf**, y entre **nombre** y **%s**. Obsérvese que **nombre** no va precedido por el operador (&). La lectura de cadenas de caracteres se detiene en cuanto se encuentra un espacio en blanco, por lo que para leer una línea completa con varias palabras hay que utilizar otras técnicas diferentes.

En los formatos de la cadena de control de **scanf()** pueden introducirse *corchetes* [...], que se utilizan como sigue. La sentencia...

```
scanf("%[AB \n\t]", s); // se leen solo los caracteres indicados
```

lee caracteres hasta que encuentra uno diferente de (`'A','B',' ','\n','\t'`). En otras palabras, se leen sólo los caracteres que aparecen en el corchete. Cuando se encuentra un carácter distinto de éstos se detiene la lectura y se devuelve el control al programa que llamó a **scanf()**. Si los corchetes contienen un carácter (^), se leen todos los caracteres distintos de los caracteres que se encuentran dentro de los corchetes a continuación del (^). Por ejemplo, la sentencia,

```
scanf(" %[^\n]", s);
```

lee todos los caracteres que encuentra hasta que llega al carácter ***nueva línea*** '\n'. Esta sentencia puede utilizarse por tanto para leer líneas completas, con blancos incluidos.

Recuérdese que con el formato ***%s*** la le lectura se detiene al llegar al primer delimitador (carácter blanco, tabulador o nueva línea).