Como hacer un programa separando el código fuente en múltiples archivos.

Un programa puede desarollarse en su totalidad ingresando todo el código en un único archivo fuente *.c y luego ser compilado para generar un único ejecutable.

Por ejemplo: un único archivo main.c es compilado para generar un archivo main.exe (normalmente sin extensión en el mundo Unix, o con extensión **.com** o **.exe** en el mundo MS-DOS/Windows o con extensión **.hex** en el mundo de los microcontroladores).

Pero también puede crearse dicho único programa ejecutable separando el desarrollo del código en diferentes archivos fuentes ***.c**.

Dicho proceso de compilación se divide en dos etapas, la etapa de **compilación** propiamente dicha y la de **enlazado.** Aunque hay que tener presente que las IDE de desarrollo como MPLAB X realizan estas etapas de un modo transparente al usuario como si fuere un único proceso.

Etapa de compilación: En una primer etapa nuestro código en **C** se transforma en código objeto (generando archivos .o) es decir, código máquina, (instrucciones que el ordenador puede ejecutar).

Etapa de enlazado: En la segunda etapa, estos ficheros objeto son unidos entre sí para formar el archivo ejecutable (normalmente sin extensión en el mundo Unix, o con extensión **.com** o **.exe** en el mundo MS-DOS/Windows) o con extensión **.hex** para el mundo de los microcontroladores.

Si optamos por separar el código en diferentes archivos fuente cada archivo ***.c** que creemos será compilado por separado generando cada uno su archivo objeto o módulo **.o** correspondiente.

Luego todos los archivos de código objeto o módulos **.o** se unirán para generar un único archivo ejecutable mediante el enlazador o linker.

Estos pasos muchas veces suceden de manera casi invisible al usuario porque las IDE como MPLAB X lo hacen de manera automática. Pero es muy importante entender que el proceso completo de generación de un programa se da pasando en primer término por la compilación propiamente dicha y luego realizando la etapa de enlazado hasta llegar al ejecutable.

PROGRAMA PARA CALCULAR LA SUPERFICIE Y EL PERÍMETRO DE UN RECTÁNGULO Y TRÍANGULO.

Para ilustrar lo expuesto hasta aquí crearemos un programa para calcular la superficie y el perímetro de un rectángulo y un tríangulo utilizando las dos variantes. Es decir, escribiendo todo el código del programa en un único archivo *.c y luego separando el desarrollo del código en varios archivos *.c.

Para implementar el programa crearemos 4 funciones, 2 que hagan los cálculos de las superficies de las figuras y 2 de sus perímetros.

EJEMPLO 1A - Un único archivo main.c

```
/*
ejemplo de programa creado con un único archivos fuente.

Compilador: gcc.exe

*/

#include <stdio.h>
#include <stdlib.h>

float superficieRectangulo(float base, float altura); // prototipo, declaración o reserva de función
float perimetroRectangulo(float base, float altura); // prototipo, declaración o reserva de función

float superficieTriangulo(float a, float b, float c);     // prototipo, declaración o reserva de función
float perimetroTriangulo(float a, float b, float c);     // prototipo, declaración o reserva de función


// programa principal
int main(int argc, char *argv[])
{



        float ladoA = 0, ladoB = 0, ladoC = 0; // declaración y definición de variables locales a main para el triangulo


        float baserectangulo=0,alturarectangulo=0; // declaración y definición de variables locales a main para el rectangulo



  printf("EJEMPLO PROYECTO CON  UN SOLO ARCHIVO\n\n");

  printf( "Escriba el valor del lado A del triangulo: " );
  scanf( "%f", &ladoA);
  printf( "Escriba el valor del lado B del triangulo: " );
  scanf( "%f", &ladoB);
  printf( "Escriba el valor del lado C del triangulo: " );
  scanf( "%f", &ladoC);
  printf( "\n" );

  printf( "Escriba el valor de la base del rectangulo: " );
  scanf( "%f", &baserectangulo);
  printf( "Escriba el valor de la altura del rectangulo: " );
  scanf( "%f", &alturarectangulo);
  printf( "\n" );

  printf("superficie del triangulo: %f\n\n", superficieTriangulo(ladoA,ladoB,ladoC));
  printf("perimetro del triangulo: %f\n\n", perimetroTriangulo(ladoA,ladoB,ladoC));
  printf( "\n" );

  printf("superficie del rectangulo: %f\n\n", superficieRectangulo(baserectangulo,alturarectangulo));
  printf("perimetro del rectangulo: %f\n\n", perimetroRectangulo(baserectangulo,alturarectangulo));

  system("PAUSE");
  return 0;
}
```

```
float superficieRectangulo(float base, float altura) // desarrollo o definición de función
{
	return base * altura;
}

float perimetroRectangulo(float base, float altura) // desarrollo o definición de función
{
	return base + altura;
}


float superficieTriangulo(float a, float b, float c) // desarrollo o definición de función
{
	float Area;
	float Semisuma;
	Semisuma=(a+b+c)/2;
	Area=sqrt(Semisuma*(Semisuma-a)*(Semisuma-b)*(Semisuma-c));
	return Area;
}

float perimetroTriangulo(float a, float b, float c) // desarrollo o definición de función
{
	return a+b+c;
}
```

EJEMPLO1B

En esta nueva versión separaremos el código en 3 archivos fuentes: main.c, rectangulo.c y triangulo.c

main.c

```
/*
archivo: main.c

ejemplo1b:
ejemplo de programa creado con múltiples archivos fuente.
Las funciones son declaradas en main.c y desarrolladas en los archivos rectangulo.c y triangulo.c

Compilador: gcc.exe

*/

#include <stdio.h>
#include <stdlib.h>

float superficieRectangulo(float base, float altura);    // prototipo, declaración o reserva de función
float perimetroRectangulo(float base, float altura);     // prototipo, declaración o reserva de función

float superficieTriangulo(float a, float b, float c);    // prototipo, declaración o reserva de función
float perimetroTriangulo(float a, float b, float c);     // prototipo, declaración o reserva de función


// programa principal
int main(int argc, char *argv[])
{



 float ladoA = 0, ladoB = 0, ladoC = 0; // declaración y definición de variables locales a main para el triangulo

 float baserectangulo=0,alturarectangulo=0; // declaración y definición de variables locales a main para el rectangulo



   printf("EJEMPLO PROYECTO CON MULTIPLES ARCHIVOS FUENTE\n\n");

   printf( "Escriba el valor del lado A del triangulo: " );
   scanf( "%f", &ladoA);
   printf( "Escriba el valor del lado B del triangulo: " );
   scanf( "%f", &ladoB);
   printf( "Escriba el valor del lado C del triangulo: " );
   scanf( "%f", &ladoC);
   printf( "\n" );
```

```
   printf( "Escriba el valor de la base del rectangulo: " );
   scanf( "%f", &baserectangulo);
   printf( "Escriba el valor de la altura del rectangulo: " );
   scanf( "%f", &alturarectangulo);
   printf( "\n" );

   printf("superficie del triangulo: %f\n\n", superficieTriangulo(ladoA,ladoB,ladoC));
   printf("perimetro del triangulo: %f\n\n", perimetroTriangulo(ladoA,ladoB,ladoC));
   printf( "\n" );

   printf("superficie del rectangulo: %f\n\n", superficieRectangulo(baserectangulo,alturarectangulo));
   printf("perimetro del rectangulo: %f\n\n", perimetroRectangulo(baserectangulo,alturarectangulo));

   system("PAUSE");
   return 0;
}
```

rectangulo.c

```
/*
archivo: rectangulo.c

Desarrollo de las funciones que calculan la superficie y el perímetro del rectángulo

*/

float superficieRectangulo(float base, float altura) // desarrollo o definición de función
{
	return base * altura;
}

float perimetroRectangulo(float base, float altura) // desarrollo o definición de función
{
	return base + altura;
}
```

triangulo.c

```
/*
archivo:  triangulo.c

Desarrollo de las funciones que calculan la superficie y el perímetro del triángulo
*/

#include <math.h>


/*
TEOREMA DE HERON Para hallar el area de un triangulo
La fórmula de Herón se distingue de otras fórmulas para
hallar el área de un triángulo, como la de la mitad de la base por la altura
o la de la mitad del módulo de un producto cruz de dos lados,
al no requerir ninguna elección arbitraria de un lado
como base o un vértice como origen.
*/

float superficieTriangulo(float a, float b, float c) // desarrollo o definición de función
{
	float Area;
	float Semisuma;
	Semisuma=(a+b+c)/2;
	Area=sqrt(Semisuma*(Semisuma-a)*(Semisuma-b)*(Semisuma-c));
	return Area;
}


float perimetroTriangulo(float a, float b, float c) // desarrollo o definición de función
{
	return a+b+c;
}
```

Cada uno de los archivos fuentes ***.c** será compilado por separado y se generará por cada uno de ellos un archivo objeto o módulo ***.o**.

Luego el enlazador o linker unirá todos los archivos objeto ***.o** para crear un único archivo ejecutable ***.exe *.com o *.hex**.

Para hacer más fácil la tarea de estudio de los ejemplos utilicé el compilador de C gcc que se puede instalar con el programa de desarrollo devc++ de blodsheed.

Compilando cada archivo fuente con el compilador **gcc** con la opción **-c** generaremos el paso intermedio de crear los archivos objetos o módulos.

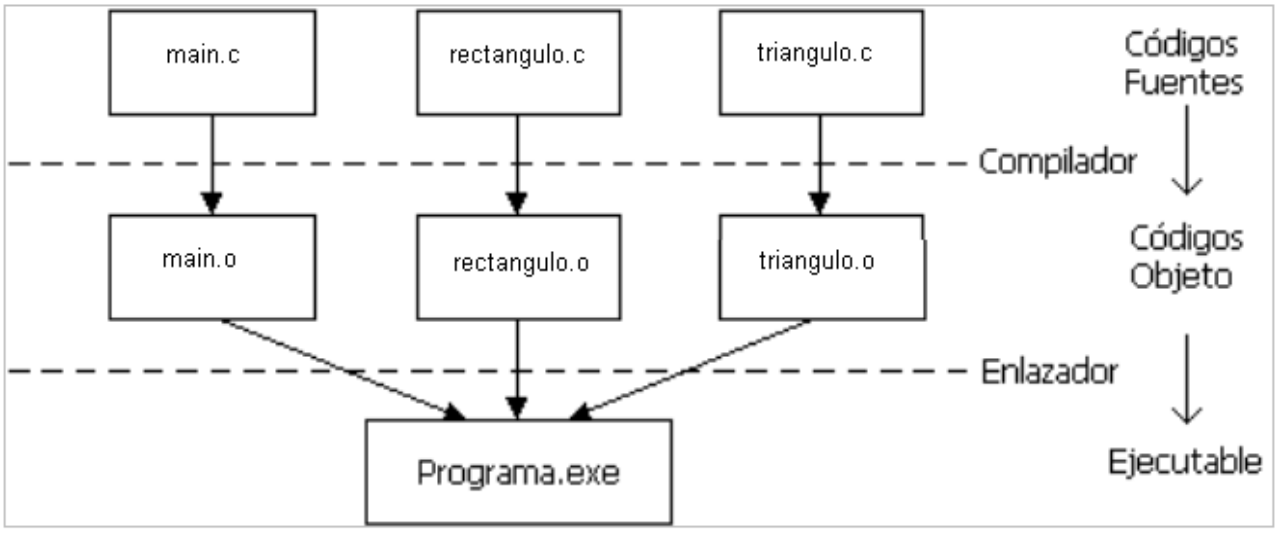
gcc -c main.c
gcc -c rectangulo.c
gcc -c triangulo.c

con estas 3 líneas habremos creado 3 archivos.

main.o
rectangulo.o
triangulo.o

luego linkearemos todos los archivos objects con la opción -g y -o generando un único ejecutable **programa.exe**

gcc -g -o programa.exe main.o rectangulo.o triangulo.o



Si quisiéramos podríamos modificar únicamente el archivo **main.c** y recompilarlo para generar un nuevo archivo **main.o** y luego linkearlo con los otros archivos object **rectangulo.o** y **triangulo.o** que fueron generados con los archivos que no fueron modificados **rectangulo.c** y **triangulo.c**.

De esta manera, si no hemos modificado los archivos rectangulo.c y triangulo.c que se compilaron a sus correspondientes archivos objeto rectangulo.o y triangulo.o podemos ahorrarnos esa parte de la compilación linkeando el nuevo archivo main.o junto a los viejos archivos rectangulo.o y triangulo.o que ya teníamos generados.

Esto nos permite reutilizar código, a modo de librerías. Podríamos darle a alguien

los dos archivos **rectangulo.o** y **triangulo.o**, e indicarle que funciones contienen y como se utilizan (parámetros, valor de retorno, etc.) para que pueda hacer uso del código que hemos creado.

Las ventajas son evidentes. Una mejor organización del código, una mejor modularidad y, sobre todo, mayor velocidad a la hora de compilar, ya que los archivos que no hayan sido modificados no deberán ser recompilados. También nos permite seleccionar que archivos podemos ocultar frente a terceras personas para proteger información sobre como hemos programado el contenido de los archivos fuentes.

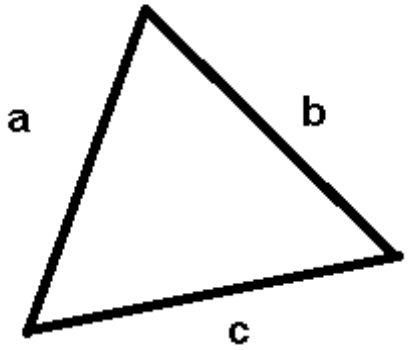
Bastaría con entregar sólo el archivo fuente **main.c** y los dos archivos **rectangulo.o** y **triangulo.o** para que puedan recompilar el programa completo sin saber como fueron codificadas las funciones dentro de rectangulo.c y triangulo.c.

Con solo indicarle al usuario cuales son las funciones disponibles y cuales son sus parámetros de entrada y su tipo y valor de retorno alcanzará para que éstas puedan ser utilizadas sin tener que exhibir su desarrollo.

Ejemplo:

float superficieTriangulo(float a, float b, float c)

float perimetroTriangulo(float a, float b, float c)



No importa como se hace el cálculo de la superficie o del perímetro del triangulo. Solo utilizamos la función.

```
printf("superficie del triangulo: %f\n\n", superficieTriangulo(ladoA,ladoB,ladoC));
printf("perimetro del triangulo: %f\n\n", perimetroTriangulo(ladoA,ladoB,ladoC));
```

Fuente:
http://es.wikibooks.org/wiki/Programaci%C3%B3n_en_C/El_proceso_de_compilaci%C3%B3n

http://gzaloprgm.com.ar/tut_compilando/

http://www.zator.com/Cpp/E3_2_2.htm

falta ampliar aquí el tema de los archivos headers.h y las directivas de preprocesado

#include

Comportamiento de las variables y funciones en el desarrollo de programas separados en múltiples archivos fuentes.

Para comprender el comportamiento de las variables y funciones en proyectos con múltiples archivos primero debemos tener muy en claro que es una variable y cuales son sus atributos, todas sus características y las clasificaciones.

VARIABLES: representan un espacio de memoria en el que podemos almacenar temporalmente nuestros datos. Por cada variable que definimos en nuestro programa se reservará una cantidad finita de bytes de memoria en una dirección física.

Podemos crear variables simplemente indicando el **tipo** de dato y el **nombre** (identificador).

Sintaxis:

[signed|unsigned] **int** <identificador> ;

ejemplo:

signed int suma; // en la variable suma podremos almacenar numeros enteros.

ELEMENTOS Y CARACTERÍSTICAS DE LAS VARIABLES:

Identificador o nombre: Los identificadores pueden contener las letras **a** a **z** y **A** a **Z**, el guión bajo "**_**" ("Underscore") y los dígitos **0** a **9**.

Caracteres permitidos: a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Guión bajo: _

Dígitos permitidos: 0 1 2 3 4 5 6 7 8 9

Restricciones: Solo hay dos restricciones en cuanto a la composición: El primer carácter debe ser una letra o el guión bajo. Sin embargo el estándar establece que los identificadores que comienzan con guión bajo y mayúscula no deben ser utilizados. Este tipo de nombres se reserva para los compiladores y las librerías estándar. Tampoco se permite la utilización de nombres que contengan dos guiones bajos seguidos.

El estándar ANSI establece que como mínimo serán significativos los 31 primeros caracteres, aunque pueden ser más, según la implementación. Es decir, para que un compilador se adhiera al estándar ANSI, debe considerar como significativos, al menos, los 31 primeros caracteres.

Los identificadores distinguen mayúsculas y minúsculas, así que Sum, sum y suM son distintos para el compilador.

Un identificador no puede coincidir con una palabra clave o reservada (for, while, int, switch, etc) o con el nombre de una función.

Palabras reservadas: No se pueden utilizar como identificadores de variables las siguientes palabras:

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

Además los compiladores en particular pueden reservar otras palabras, como es el caso de la palabra **`asm`.**

Tipo: Los datos que podemos almacenar en las variables pueden ser de dos variedades bien marcadas.

- **Números enteros:** bit, signed char, unsigned char, signed short, unsigned short, signed int , unsigned int, signed short long, unsigned short long, signed long, unsigned long, signed long long y unsigned long long.
- **Números fraccionarios:** float, double y long double.

Tamaño: dependiendo del tipo de dato elegido se reservará una cantidad finita de bytes. 1, 2, 3 o 4 bytes.

Rangos de valores que se pueden almacenar en los diferentes tipos de datos **enteros**.

TIPO de DATO	Tamaño (bits)	NOMBRES GENERICOS	MINIMO	MAXIMO
bit (no es standard de Ansi C)	1	BIT	0	1
signed **char**	8	INT8	0	255
unsigned **char**	8	UINT8	-128	127
signed **short**	16	INT16	-32.768	32.767
unsigned **short**	16	UINT16	0	65.535
signed **int**	16	INT16	-32.768	32.767
unsigned **int**	16	UINT16	0	65.535
signed **short long** No es standard de Ansi C	24	INT24	-8.388.608	8.388.607
unsigned **short long.** No es standard de Ansi C	24	UINT24	0	16.777.215
signed **long**	32	INT32	-2.147.483.648	2.147.483.647
unsigned **long**	32	UINT32	0	4.294.967.295
signed **long long**	32	INT32	-2.147.483.648	2.147.483.647
unsigned **long long**	32	UINT32	0	4.294.967.295

TIPO de DATO	Tamaño (bits)
`Float`	3 o 4
`double`	3 o 4
`long double`	3 o 4

Symbol	Meaning	24-bit Value	32-bit Value
XXX_RADIX	Base de la representación del exponente (Radix of exponent representation)	2	2
XXX_ROUNDS	Modo de redondeo para la suma (Rounding mode for addition)	0	0
XXX_MIN_EXP	Valor mínimo exponente en **base 2** para que sea un valor float normalizado (Min. n such that FLT_RADIX n^{-1} is a normalized float value)	-125	-125
XXX_MIN_10_EXP	Valor mínimo de exponente en **base 10** para que sea un valor float normalizado (Min. n such that 10^n is a normalized float value)	-37	-37
XXX_MAX_EXP	Valor máximo de exponente en **base 2** para que sea un valor float normalizado (Max. n such that FLT_RADIX n^{-1} is a normalized float value)	128	128
XXX_MAX_10_EXP	Valor máximo de exponente en **base 10** para que sea un valor float normalizado (Max. n such that 10^n is a normalized float value)	38	38
XXX_MANT_DIG	Número de dígitos que componen la mantisa (Number of FLT_RADIX mantissa digits)	16	24
XXX_EPSILON The smallest number which added to 1.0 does not yield 1.0	El número de dígitos binarios en la mantisa es acotado. Por eso las sumas de la forma **1+x**, donde x es muy pequeño, se redondean al número 1. Por ello el menor número que sumado a 1.0 no alcanza a incrementar a 1.0 (The smallest number which added to 1.0 does not yield 1.0) es: 3.05176e-05 o 1.19209e-07	3.05176e-05	1.19209e-07

Valor: valor propiamente dicho almacenado en la variable. Debe ser un dato válido de acuerdo al rango permitido por el tipo elegido (bit, char, int, etc.).

Dirección de memoria: todas las variables ocupan un espacio físico de memoria. Dicho espacio físico de memoria tiene una ubicación o dirección como si fuere el domicilio de una casa. Para conocer la dirección de memoria de una variable debemos utilizar el operador &.
Ejemplo: printf("direccion de memoria de la Variable A: %p",&A);

Modificadores de almacenamiento: C brinda modificadores que afectan al modo en que se almacenan las variables y a su ámbito temporal, es decir, la zona de programa desde donde las variables serán accesibles o donde pueden ser referenciadas (**Alcance o ámbito (scope))** y su vida útil o cuanto tiempo permanece la variable en memoria (**Duración de almacenamiento**).

Valor de inicialización: dependiendo del calificador que se utilice al comenzar el programa las variables podrán tener un valor aleatorio o serán inicializadas a cero. Las variables **globales** y las creadas con el calificador **static** serán inicializadas a cero si no se les asigna un valor.

```
/*
ejemplo valores de iniciliazación de variables.

Compilador: gcc.exe

*/

#include <stdio.h>
#include <stdlib.h>

// variables globales

int miGlobal; //Todas las variables declaradas fuera de un bloque tendrán siempre duración estática o fija.
              //Aún cuando no se las defina con el modificador static. Son inicializadas a cero a pesar de que no se le indique un valor.

// programa principal
int main(int argc, char *argv[])
{
   int miLocal; // variable auto
   static int miStaticLocal; // variable static

   printf( "Valor de variable miGlobal: %d \n",miGlobal);
   printf( "Valor de variable miLocal: %d \n",miLocal);
   printf( "Valor de variable miStaticLocal: %d \n",miStaticLocal);
   system("PAUSE");
   return 0;
}


/*
salida:
Valor de variable miGlobal: 0
Valor de variable miLocal: 1973198388
Valor de variable miStaticLocal: 0
Presione una tecla para continuar . . .
*/
```

Conexión o vinculación: Cuando se desarrolla un programa con múltiples archivos fuente la conexión o vinculación determina si todos los archivos pueden tener acceso a las variables de otros archivos o sólo acceso limitado al archivo donde fueron creadas.

Estos conceptos son tratados en detalle en los párrafos subsiguientes.

Alcance o ámbito (scope) de las variables:

El alcance o ámbito de una variable define el lugar desde donde puede ser utilizada, es decir donde puede esar está activa, disponible o visible.

El ámbito o alcance de una variable determina cuáles son las funciones que pueden reconocerla.

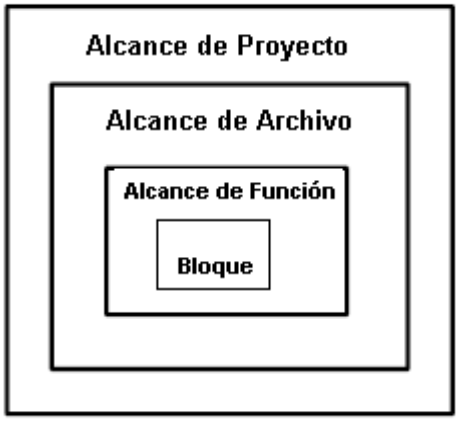
Si una función reconoce una variable la misma será visible en dicha función.

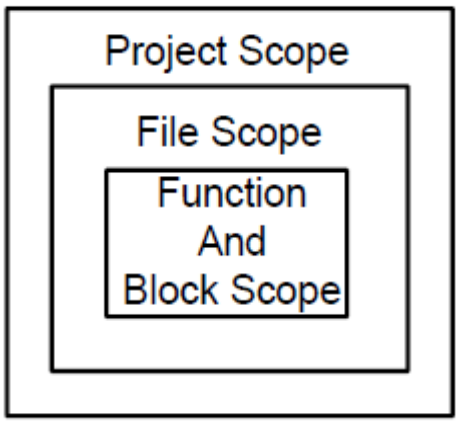
El ámbito es la zona de un programa en la que es visible una variable.

La posición de la sentencia que crea la variable en el programa determina el ámbito.

```
char i;
void main(void)
{
     // i podrá ser alcanzada por la función main
    // z no podrá ser alcanzada por la función main.
};
char z;
```

- Las variables no pueden ser accedidas fuera de su ámbito o scope.
- Hay cuatro tipos de ámbitos o scope:
    - de proyecto,
    - de archivo,
    - de función,
    - y de bloque.





Variable globales con alcance de PROYECTO: Si una variable tiene alcance de proyecto (project scope) entonces significa que la variable podrá ser utilizada o visible desde cualquier archivo fuente que pertenezca al proyecto.

Las Variables que tienen alcance de proyecto (project scope) también son llamadas como **variables globales de proyecto** o variables **super-globales**.

Para declarar una variable global de proyecto o super-global debemos hacerlo declarando la variable al principio de un archivo fuente y fuera de cualquier bloque y sin agregar el calificador static.

Por ejemplo:

```
char i; // i es una Variable global con alcance de Proyecto o Project scope.
static char j; // j es una Variable Global con alcance de archivo o File scope. (la palabra static lo determina)

Int miFuncion1(char k) // las funciones tienen por defecto alcance de proyecto o Project scope
    { // comienzo de bloque o Block start
                    // Los parámetros de las funciones como char k
                     // tienen alcance de función o bloque (block scope).

     static char m; // m es una Variable Local estática con alcance de función o bloque
      char n; //  n es una variable local Automática con alcance de función o bloque.
    } // final de bloque o Block end

static Int miFuncion2(char k) // las funciones con calificador static tienen alcance de
     {                        //   Archivo o File scope.
       while(1)
          {
           int q; // q tiene alcance de bloque. Sólo puede ser utilizada dentro del while.
          }
            // aquí q ya no puede ser usada porque q tiene alcance de bloque y estamos fuera
             // del bloque del while.
     }
```

Variable con alcance de ARCHIVO: Si una variable tiene alcance de archivo (File scope), entonces significa que la variable podrá ser utilizada o visible únicamente desde dentro del archivo fuente al que pertenezca y podrá ser alcanzada únicamente por el código subsiguiente al de su declaración dentro del archivo.

Las variables que tienen alcance de archivo (File scope) también son llamadas como **variables globales con alcance de archivo** o simplemente **variables globales**.

Para declarar una variable global de archivo o variable-global debemos hacerlo declarando la variable al principio de un archivo fuente y fuera de cualquier bloque y anteponiendo el calificador static.

Ejemplo variables globales con alcance de archivo.

```
static char j; // solo podrá ser utilizada dentro del presente archivo. Otros archivos fuentes no  podrán utilizarla.

void main(void)
{
     // j podrá ser alcanzada por la función main
    // z no podrá ser alcanzada por la función main porque aún no fue declarada y está siendo utilizada antes
     // de su declaración. Si podrá ser utilizada por miFuncion();
};

static char z;

void miFuncion(void)
{
 printf("valor de Z %c",z); // z puede ser alcanzada por miFunción porque está siendo
                            // utilizada luego o debajo de su declaración.
}
```

Variables locales con alcance de FUNCIÓN: Si una variable tiene alcance de función (function scope) entonces significa que la variable podrá ser utilizada o visible únicamente desde dentro de la función en donde sea declarada. Dicha variable podrá ser vista únicamente desde cualquier parte del código dentro de la función hasta el final de la misma.

Las variables que tienen alcance de función (Function scope) son llamadas **variables locales**.

```
void main(void)
{
    int q; // q tiene alcance local. Sólo puede ser utilizada dentro de main.
}

void mifuncion(void)
{
   int q; // q tiene alcance local. Sólo puede ser utilizada dentro de mifuncion.
          // nótese que se puede usar el mismo nombre para variables locales diferentes y no se produce conflicto.
}
```

Variables locales con alcance de BLOQUE: Si una variable tiene alcance de bloque (Block scope) entonces significa que la variable podrá ser utilizada o visible únicamente desde dentro del bloque en donde sea declarada. Dicha variable podrá ser vista desde cualquier parte del código que forma parte del bloque desde el comienzo del bloque hasta el final del mismo.

Las variables que tienen alcance de bloque (Block scope) también son llamadas **variables locales**.

```
void main(void)
{

while(1)
    {
    int q; // la variable q tiene alcance de bloque. Sólo puede ser utilizada dentro del while.

    }
         // aquí la variable q ya no puede ser usada porque q tiene alcance de bloque y estamos
        // fuera del bloque del while.
}
```

Duración de almacenamiento:

La duración hace referencia a la cantidad de tiempo que la variable vive en una determinada locación de memoria. La duración puede ser **automática** (dinámica) o **estática** (fija).

La duración automática significa que puede crearse y destruirse repetidamente como en el caso de la variable **auto**.

LA duración estática refiere a que puede durar toda la vida del programa como en el caso de la variable **static**.

• **Variables de duración auto:** Una variable **x** con duración **auto** reservará espacio en memoria cuando el ámbito de utilización sea alcanzado. Por ejemplo en una variable con alcance de función la variable será creada cuando se instancie y se entre a la función. Y cuando se termine la ejecución de la función y se salga de la misma el espacio de memoria ocupado por dicha variable será liberado y podrá ser utilizado por otra variable. Y cuando se vuelva a entrar a la función un nuevo espacio de memoria será utilizado para alojar a la variable **x**.

Para declarar una variable con duración automática se utiliza el calificador **auto.**

Sintaxis:

[auto] <tipo> <nombre_variable>;

El calificador **auto** sirve para declarar únicamente variables automáticas o locales. Es el modificador por defecto cuando se declaran variables u objetos locales, es decir, si no se especifica ningún modificador al crear una variable dentro de una función se creará una variable automática.

No es posible crear variables automáticas globales.

Debido a que estos objetos serán creados y destruidos cada vez que sea necesario, usándose, en general, diferentes posiciones de memoria, su valor se perderá cada vez que sean nuevamente creadas, perdiéndose el valor previo en cada caso.

```
void main(void)
{
    int z; // z tiene alcance local y duración auto.
    auto int q; // q también tiene alcance local y duración auto.
}
```

Estas variables se crean durante la ejecución, y se elige el tipo de memoria a utilizar en función del ámbito temporal de la variable. Una vez cumplido el ámbito, la variable es destruida. Es decir, una variable automática local de una función se creará cuando sea declarada, y se destruirá al terminar la función. Una variable local automática de un bucle será destruida cuando el bucle termine.

• **Variables de duración fija o static:** las variables de duración fija o static reservarán espacio de memoria al comienzo de la ejecución del programa y dicho espacio en memoria nunca cambiará durante toda la vida de ejecución del programa.

```
void main(void)
{
   static int z; // z tiene alcance local y duración static.
    auto int q; // q tiene alcance local y duración auto.
    int x; // x también tiene alcance local y duración auto. Nótese el calificado auto por defecto
}
```

DISTINTOS SIGNIFICADOS DEL MODIFICADOR STATIC (SEGÚN SE LO UTILICE DENTRO O FUERA DE UNA FUNCIÓN): El modificador **static** tiene un comportamiento diferente cuando se lo utiliza fuera de un bloque de una función. Todas las variables declaradas fuera de un bloque tendrán siempre duración estática o fija. Aún cuando no se las defina con el modificador static.

El modificador static en las definiciones de variables declaradas fuera de un bloque sirve para indicar si la variable que se crea es una variable global con alcance limitado al archivo fuente o por el contrario con alcance global para todo el proyecto si no se lo utiliza.

Con el calificador static la variable sólo tendrá alcance en el archivo en el que se la crea y **sin el calificador static** la variable tendrá alcance global a nivel de proyecto y podrá utilizarse en todos los demás archivos fuente.

Valor de inicialización: dependiendo del calificador que se utilice en la declaración de las variables al comenzar el programa podrán tener un valor aleatorio o serán inicializadas a cero.

Las variables globales y las creadas con el calificador static **siempre** serán inicializadas a cero o en su caso al valor que se les establezca.

Las variables auto serán inicializadas con un valor aleatorio si no se les asigna un valor.

```
/*
ejemplo valores de iniciliazación de variables.

Compilador: gcc.exe

*/

#include <stdio.h>
#include <stdlib.h>

// variables globales

int miGlobal; // tiene alcance global de proyecto y duración static
              // (todas las variables globales son static).
              // Su valor de inicialización será 0.

// declaración de funciones

void mifuncion(void);

// programa principal
int main(int argc, char *argv[])
{
   int miLocal; // tiene alcance local y duración auto. Su valor de inicialización será aleatorio.
                // (nótese que todas las variables locales son auto por defecto)
```

```
   int q=0; // variable para el bucle for

   static int miStaticLocal;

   printf( "Valor de variable miGlobal: %d \n",miGlobal);
   printf( "Valor de variable miLocal: %d \n",miLocal);
   printf( "Valor de variable miStaticLocal: %d \n",miStaticLocal);

   for (q=0;q<10;q++)
      {
       mifuncion();
      }

   system("PAUSE");
   return 0;
}


// desarrollo de funciones
void mifuncion(void)
{
    static int miStatic=10; // la variable w tiene duración static. Su valor de inicialización será 10
                        // y luego en los próximos llamados de la función conservará el valor que se le haya dado.
   printf( "Valor de variable miStatic: %d \n",miStatic);
    miStatic++;
            // la próxima vez que se invoque la función miStatic conservará el valor
            // y valdrá 11.

    int a; // a tiene duración auto. Cada vez que se invoque la función a tendrá un valor aleatorio.
    a=10; // a tiene duración auto. Cada vez que se invoque la función a tendrá valor 10.
printf( "Valor de variable a: %d \n",a);
    a++;
     // la próxima vez que se invoque la función a será inicializada nuevamente y valdrá 10 y no 11.
}



/*
salida:
Valor de variable miGlobal: 0
Valor de variable miLocal: 1973198388
Valor de variable miStaticLocal: 0
Valor de variable miStatic: 10
Valor de variable a: 10
Valor de variable miStatic: 11
Valor de variable a: 10
Valor de variable miStatic: 12
Valor de variable a: 10
Valor de variable miStatic: 13
Valor de variable a: 10
Valor de variable miStatic: 14
Valor de variable a: 10
Valor de variable miStatic: 15
Valor de variable a: 10
Valor de variable miStatic: 16
Valor de variable a: 10
Valor de variable miStatic: 17
Valor de variable a: 10
Valor de variable miStatic: 18
Valor de variable a: 10
Valor de variable miStatic: 19
Valor de variable a: 10
Presione una tecla para continuar . . .*/
```

VARIABLES GLOBALES VS LOCALES

Las variables globales: son las variables que se crean fuera de cualquier función. Son visibles por cualquier función, incluida la función main().

Una variable global es visible desde su punto de definición en el archivo fuente en adelante. Es decir, si se define una variable global, cualquier línea del resto del programa que le siga podrá utilizar o referenciar esa variable. Es decir, se podrán referenciar desde el punto del programa en que están definidas hasta el final del archivo fuente. Si un archivo fuente tiene más de una función, todas las funciones que siguen a la definición de la variable podrán referenciarla.

Características:

Incrementan el rendimiento debido a que se puede acceder a ellas directamente desde cualquier función y se elimina la sobrecarga de pasar datos a funciones.

Las variables globales son accesibles por cualquier función definida en el mismo archivo después de la declaración de la variable.

Las variables globales (con alcance de proyecto) también son accesibles desde funciones creadas en otros archivos. Pero para poder ser utilizadas deben ser declaradas previamente con el calificador extern en cada archivo en donde se utilizan.

Ejemplo: definimos en un archivo main.c la variable global bandera y para poder utilizarla desde el archivo funciones.c debemos previamente declarar su existencia en dicho archivo con el calificador **extern**.

Archivo main.c

```
int bandera; // definición de variable global con alcance de proyecto (sin calificador static)

int main(void)
{
}
```

Archivo funciones.c

```
 extern int bandera; // declaro que la variable bandera está definida en otro lugar.

mifuncion(int a)
{
bandera=1;
}
```

Características de las variables globales:

Las variables globales sólo pueden tener duración estática. No pueden ser definidas como auto. Si se las declara como auto se produce un error.

Son visibles globalmente en el archivo fuente a continuación del punto de definición.

Las variables globales están disponibles a otros archivos fuente (enlace externo). Salvo que se las declare con el calificador static que las hace disponibles únicamente para el archivo que las contiene.

Las variables locales: Son las variables definidas dentro de una función y sólo son visibles dentro de esa función específica.

Además de tener un ámbito restringido, son especiales por otra razón: existen en memoria sólo cuando la función está activa. Es decir, mientras se ejecutan las sentencias de la función. Cuando la función no se está ejecutando, sus variables locales no ocupan espacio en memoria, ya que no existen.

No se puede cambiar una variable local por ninguna sentencia externa a la función.

Los nombres de las variables locales no son únicos. Dos o más funciones pueden definir la misma variable con el mismo nombre. Cada variable será distinta y pertenecerá a su función específica.

Las variables locales de las funciones no existen en memoria hasta que se ejecute la función. Por esta razón, múltiples funciones pueden compartir la misma memoria para sus variables locales. Pero no al mismo tiempo.

Características de las variables locales: no pueden ser modificadas por ninguna sentencia externa a la función en las que fueron creadas. Sólo son visibles dentro de la función donde se las creó.

Los nombres de las variables locales pueden ser usados para crear otras variables locales con el mismo nombre en otras funciones. Cada variable será distinta y pertenecerá a la función en la que fueron creadas.

Por defecto son "auto": las variables creadas dentro de una función se crean con el calificador auto por defecto. Se crean automáticamente cuando se entra a la función y se liberan también automáticamente cuando se termina la ejecución de la función. El valor se pierde al salir de la función.

Se les asigna espacio en memoria automáticamente a la entrada de la función y se les libera el espacio tan pronto se sale de dicha función.

Calificador static para las variables locales: Se pueden crear variables locales con el calificador static, y en este caso la variable, y por consiguiente su valor, no se extinguirán al salir de la función. El valor de la variable se conservará durante toda la ejecución del programa.

Las variables estáticas son opuestas en su significado a las variables automáticas. Las variables estáticas no se borran al finalizar la función. No se pierde su valor. Cuando la función termina su ejecución las variables retienen sus valores entre llamadas sucesivas a dicha función.

Se las utiliza para acumular valores a través de sucesivas llamadas a la función. Ejemplo: una variable que lleve la cuenta de cuantas veces fue invocada una función.

CONEXIÓN O VINCULACIÓN: determina si la variable o función solo puede utilizarse en el archivo actual (vinculación interna) o desde cualquier otro archivo fuente dentro de un proyecto con múltiples archivos (vinculación externa).

> **Vinculación interna:** variable o función global con el calificador static. Sólo pueden ser utilizadas desde el archivo fuente que las contiene.
>
> **Vinculación externa:** son las **variables con alcance de proyecto:** variable o función global sin el calificador static. Pueden ser usadas desde cualquier archivo.

El calificador Extern (Especificador de clase de almacenamiento): Indica que la variable que estamos declarando se define más adelante en el mismo archivo o en un archivo diferente.

Para poder utilizar una variable definida en otro archivo será necesario utilizar el calificador **extern**. Cuando una variable se declara externa, se indica al compilador que el espacio de la variable está definido en otro lugar.

Archivo main.c

```
int bandera; // definición de variable global con alcance de proyecto (sin calificador static)

int main(void)
{
extern int sumatoria; // declaro que la variable contador será definida en otro lugar.
                      // tal declaración es necesaria porque la variable aún no ha sido definida.

                    // La declaración le informa al compilador que la variable existe y está definida en otro lugar.
                    // Le dice al compilador quedate tranquilo que en algún lado está definida. Pero no le dice en donde.
                    // El encargado de encontrar en donde ha sido definida la variable es el linker.
                    // si el linker no encuentra en donde se definió se producirá un error y no se creará el ejecutable.

extern int miGlobal; // informo al compilador que la variable se define en otro lado.
}

int miGlobal =10; // esta variable es global pero main sólo podrá verla si se indica que ha sido definida externamente
          //mediante el calificador extern. Porque fue definida fuera de main y después de la función main.

static int sumatoria; // definición de variable global con alcance de archivo (con calificador static)
                      // solo podrá ser utilizada globalmente dentro de este archivo.
```

Archivo funciones.c

```
 extern int bandera; // declaro que la variable bandera está definida en otro lugar.

int contador; // definición de variable global con alcance de proyecto.
        // Esta variable se puede utilizar desde cualquier función. Inclusive desde funciones de otros
        // archivos fuente. Al no ponerle el calificador static se la puede alcanzar desde otros archivos.
        // Si se le pusiera el calificador static sólo serviría para las funciones del archivo fuente  funciones.c
        // Y si se la intentara daríá como error un undefined reference tp 'contador'
mifuncion(int a)
{
variableGlobal=1;
}
```

De esta forma el compilador le informa al enlazador o linker que dentro del archivo funciones.c aparecen referencias no resueltas hacia la variable bandera.

El compilador no sabe en dónde está definida la variable bandera pero al hacer la declaración con el calificador extern se permite que el enlazador intente encontrar la

definición de la variable bandera en otro archivo o más adelante en el mismo archivo.

Si el enlazador encuentra una definición global apropiada resuelve la referencia indicando en donde se localiza a la variable bandera. Si el enlazador no puede localizar una definición para la variable bandera lanzará un mensaje de error y no se producirá el ejecutable.

Las funciones tienen vinculación externa de forma predeterminada: no necesitan el especificador extern. Para extender el alcance de la función a otros archivos solo hay que incluir el prototipo de la función en cada archivo en el que se invoque a la función. Luego el enlazador será el encargado de encontrar en dónde está definida o desarrollada dicha función. Si no la encuentra se produce un error y no se generará el ejecutable.

El calificador **extern** está implícito **por defecto** para los **prototipos de funciones**. En otras palabras, un prototipo de función tiene vinculación externa de forma predeterminada.

Para las funciones el calificador 'extern' es redundante. Pero en ciertas ocasiones se puede utilizar para aclarar que la función ha sido definida en otro archivo (a modo de aclaración para el programador).

Calificador static para las funciones: Si fuere necesario limitar el uso de una función al archivo que la contiene debemos agregar el calificador static a la definición de la función.