GUÍA DE REFERENCIA RÁPIDA A
mikroC

MikroElektronika
SOFTWARE AND HARDWARE SOLUTIONS FOR EMBEDDED WORLD ...making it simple

Visión general de los elementos léxicos

La guía de referencia rápida mikroC proporciona definiciones formales de los elementos léxicos en los que consiste el lenguaje de programación mikroC. Estos elementos son pareciados a las unidades reconocidas por mikroC. Cada programa escrito en mikroC consiste en las secuencias de los caracteres ASCII tales como letras, dígitos y signos especiales. Los signos no imprimibles (por ejemplo: carácter nueva línea, tabulador, etc) se refieren a los signos especiales también. Un conjunto de los elementos básicos en mikroC es organizado y limitado. El programa se escribe en la ventana *mikroC Code Editor*. Durante el proceso de compilación se realiza el análisis sintáctico (*parsing*). El *parser* debe identificar los tokens de forma que el código fuente es tokenizado, es decir, reducido a tokens y espacios en blanco (*whitespace*).

Espacio en blanco

El espacio en blanco (*whitespace*) es el nombre genérico dado a los espacios (en blanco), tabuladores horizontales y verticales y nueva línea. Los espacios en blanco se utilizan como separadores para indicar donde empiezan y terminan los tokens. Por ejemplo, las dos secuencias:

```
char i;
unsigned int j;
```

y

```
char
i ;

    unsigned int j;
```

son léxicamente equivalentes y se analizan idénticamente dando por resultado siete tokens cada uno:

```
char
i
;
unsigned
int
j
;
```

Espacio en blanco en las cadenas literales

Cuando el espacio en blanco está dentro de una cadena literal no se utiliza como separador, sino que se interpreta como un carácter común, esto es, como una parte de una sola cadena. Por ejemplo, la siguiente cadena:

```
some_string = "mikro foo";
```

se descompone en cuatro tokens, con tal de que la cadena literal represente un token:

```
some_string
=
"mikro foo"
;
```

Tokens

Un token es el elemento más pequeño del lenguaje de programación C reconocido por el compilador. El código fuente es escaneado de izquierda a derecha. El analizador sintáctico (*parser*) extrae los tokens, seleccionando el que coincida con la secuencia de caracteres más larga posible dentro de la secuencia analizada.

Palabras clave

Las palabras clave o reservadas son los tokens que tienen el significado fijo y no se pueden utilizar como identificadores. Aparte da las palabras clave estándar en mikroC hay un conjunto de identificadores predefinidos (de constantes y de variables) denominados palabras reservadas. Ellas describen un microcontrolador específico y no pueden ser redefinidas. Lista alfabética de las palabras clave en mikroC:

__asm	far*	static	catch**
_asm	fdata*	struct	cdecl**
absolute	float	switch	inline**
asm	for	typedef	throw**
at	goto	union	true**
auto	idata*	unsigned	try**
bdata*	if	using	class**
bit	ilevel*	void	explicit**
break	int	volatile	friend**
case	io*	while	mutable**
cdata*	large*	xdata*	namespace**
char	long	ydata*	operator**
code*	ndata*	__automated**	private**
compact*	near*	__cdecl**	protected**
const	org	__export**	public**
continue	pascal	__finally**	template**
data*	pdata*	__import**	this**
default	register	__pascal**	typeid**
delete	return	__stdcall**	typename**
dma*	rx*	__try**	virtual**
do	sbit	_cdecl**	
double	sfr*	_export**	* arquitectura
else	short	_import**	dependiente
enum	signed	_pascal**	
extern	sizeof	_stdcall**	** reservado para
false	small*	bool**	la futura mejora

Comentarios

Los comentarios son partes de programa utilizados para aclarar operaciones de programa o para proporcionar más informaciones sobre él. Son para uso exclusivo del programador. Se eliminan del código fuente antes del análisis sintáctico. Los comentarios en mikroC pueden ser unilíneas o multilíneas. Los multilíneas están entre llaves o /* y */ como en el ejemplo a continuación:

```
/* Escriba su comentario aquí. Puede abarcar varias líneas */
```

Los cometarios unilíneas empiezan por '//'. Ejemplo:

```
// Escriba su comentario aquí.
// Puede abarcar una sóla línea.
```

Asimismo, los bloques de instrucciones del ensamblador pueden introducir los comentarios unilíneas al colocar el signo ';' delante del comentario:

```
asm {
  some_asm ; Esta instrucción del ensamblador …
}
```

Identificadores

Los identificadores son los nombres arbitrarios utilizados para identificar los objetos básicos del lenguaje tales como: etiquetas, tipos, símbolos, constantes, variables, procedimientos y funciones. Los identificadores pueden contener todas las letras mayúsculas y minúsculas del abecedario (a a z, y A a Z), el guión bajo "_", y los dígitos 0 a 9. El primer carácter debe ser una letra o el guión bajo. mikroC no distingue mayúsculas y minúsculas por defecto, así que Sum, sum y suM representan identificadores equivalentes. La distinción de mayúsculas y minúsculas se puede habilitar/deshabilitar utilizando la opción apropiada de mikroC IDE. Aunque los nombres de identificadores son arbitrarios (de acuerdo con las reglas fijas), ocurre un error si se utiliza el mismo nombre para más de un identificador dentro del mismo ámbito.
Ejemplos de identificadores válidos:

```
temperature_V1
Pressure
no_hit
dat2string
SUM3
_vtext
```

y de identificadores inválidos:

```
7temp         // NO -- no puede empezar por un numeral
%higher       // NO -- no puede contener los caracteres especiales
if            // NO -- no puede coincidir con las palabras reservadas
j23.07.04     // NO -- no puede contener los caracteres especiales (punto))
```

Literales

Los literales son tokens que representan valores fijos y numéricos o valores de caracteres. El tipo de dato correspondiente a un literal es deducido por el compilador en base a indicios implícitos, como el valor numérico y formato usados en el código fuente.

Literales enteros

Las constantes literales enteras pueden estar expresadas en los formatos hexadecimal (base 16), decimal (base 10), octal (base 8) o binario (base 2).

- Los literales enteros con el prefijo 0x (o 0X) se consideran números hexadecimales. Por ejemplo, 0x8F
- En el formato decimal los literales enteros se representan como una secuencia de dígitos con el prefijo opcional + o - (no están delimitados por coma, espacio en blanco o puntos). Si no se especifica el prefijo, el valor de la constante se considera positivo. Por ejemplo, el número 6258 equivale a +6258.
- Las constantes literales enteras que empiezan con 0 se consideran números octales. Por ejemplo, 0357.
- Las constantes con el prefijo 0b (o 0B) se consideran números binarios. Por ejemplo, 0b10100101.

Ejemplos de las constantes literales:

```
0x11    // literal hexadecimal equivale a literal decimal 17
11      // literal decimal
011     // literal octal equivale a literal decimal 9
0b11    // literal binario equivale a literal decimal 3
```

El rango permitido para los literales enteros es determinado por el tipo *long* para los literales con signo (*signed*) y por el tipo *unsigned long* para los literales sin signo (*unsigned*).

Constantes literales de punto flotante (fraccionarias)

Las constantes literales de punto flotante consisten en:

- Parte entera;
- Punto decimal;
- Parte fraccionaria; y
- e/E y un entero con signo (exponente)

Ejemplos de las constantes de punto flotante:

```
0.          // = 0.0
-1.23       // = -1.23
23.45e6     // = 23.45 * 10^6
2e-5        // = 2.0 * 10^-5
3E+10       // = 3.0 * 10^10
.09E34      // = 0.09 * 10^34
```

Literales carácter

Un literal carácter es una secuencia de caracteres ASCII delimitados por comillas simples.

Cadenas literales (alfanuméricas)

Una cadena literal representa una secuencia de caracteres ASCII encerrados entre comillas dobles. Como hemos mencionado, una cadena literal puede contener espacio en blanco. El *parser* no descompone las cadenas literales en tokens, sino que las interpreta como una totalidad. La longitud de una cadana literal depende del número de caracteres en los que consiste. Al final de cada cadena literal hay un carácter nulo (*null*), llamado el ASCII cero, que no cuenta en la longitud total de la cadena literal. Una cadena literal que no contiene ningún signo entre comillas (cadena nula) se almacena como un solo carácter nulo. Ejemplos de cadenas literales:

```
"Hola mundo!"                    // mensaje de 11 caracteres
"La temperatura es estable"     // mensaje de  25 caracteres
"  "                            // dos espacios de 2 caracteres
"C"                             // letra C de 1 carácter
""                              // cadena nula de 0 caracteres
```

Secuencias de escape

El carácter barra invertida '\' se utiliza para introducir una secuencia de escape, que permite representar visualmente ciertos caracteres que no tienen representación gráfica. Una de las más comunes secuencias de escape es el carácter nueva línea '\n'. Una barra invertida se utiliza con los números octales o hexadecimales para representar su símbolo ASCII. La tabla a la derecha muestra las secuencias de escape válidas.

Secuencia	Valor	Símbolo	Que hace
\a	0x07	BEL	Timbre audible
\b	0x08	BS	Retroceso
\f	0x0C	FF	Formfeed
\n	0x0A	LF	Nueva línea
\r	0x0D	CR	Retorno decarro
\t	0x09	HT	Tabulador horizontal
\v	0x0B	VT	Tabulador vertical
\\	0x5C	\	Barra invertida
\'	0x27	'	Comilla sencilla (Apóstrofo)
\"	0x22	"	Comilla doble
\?	0x3F	?	Interrogación
\O		cualquiera	O = cadena de hasta tres dígitos octales
\xH		cualquiera	H = cadena de dígitos hexadecimales
\XH		cualquiera	H = cadena de dígitos hexadecimales

Disambiguación

Algunas situaciones ambiguas pueden surgir cuando se utilizan secuencias de escape. Ver el ejemplo. La cadena literal se interpreta como '\x09' y '1.0 Intro'. Sin embrago, mikroC la compila como \x091 y '.0 Intro'.

```
Lcd_Out_Cp("\x091.0 Intro");
```

Para evitar estos problemas, podemos escribir el código de la siguiente manera:

```
Lcd_Out_Cp("\x09" "1.0 Intro")
```

Continuación de línea con barra invertida

Una barra invertida '\' se puede utilizar como carácter de continuación de extender una cadena constante a través de fronteras de línea:

```
"Esto es realmente \
   una cadena de una línea."
```

Constantes de enumeración

Las constantes de enumeración son los identificadores definidos en declaraciones de tipo *enum.* Por ejemplo:

```
enum weekdays {  SUN = 0, MON, TUE, WED, THU, FRI, SAT };
```

Los identificadores (enumeradores) utilizados deben ser únicos en el ámbito de la declaración enum. Se permiten los iniciadores negativos.

Puntuadores

Los puntuadores utilizados en mikroC (conocidos como separadores) son:

- [ ] - corchetes;
- ( ) - paréntesis;
- { } - llaves
- , - coma;
- ; - punto y coma ;
- : - dos puntos
- . - punto
- * - asterisco
- = - signo igual
- # - almohadilla

La mayoría de estos puntuadores también funcionan como operadores.

Corchetes

Los corchetes indican índices de matrices uni y multi dimensionales:

```
char ch, str[ ]  = "mikro";
int mat[ 3][ 4] ; /*  matriz  de  3  x  4   */
ch  =  str[ 3] ;    /*  cuarto  elemento  */
```

Paréntesis

Los paréntesis () sirven para agrupar expresiones, aislar expresiones condicionales e indicar llamadas a funciones, y declaraciones de éstas:

```
d = c *  (a + b);          /*  modifica  la  precedencia  normal  */
if  (d  ==  z)  ++x;       /*  expresión  condicional  */
func();                    /*  llamada  a  función,  sin  parámetros  */
void  func2(int  n);       /*  declaración  de  función,  con  parámetros  */
```

Llaves

Las llaves { } indican el inicio y el final de una sentencia compuesta:

```
if (d == z) {
  ++x;
  func();
}
```

Una llave de cierre } se ultiliza como terminador en una sentencia compuesta. Por lo tanto no se necesita el punto y coma después de corchete de cierre, dejando aparte las declaraciones en las estructruras.

Coma

La coma ‘,’ se utiliza para separar los parámetros en las llamadas a función, los identificadores en las declaraciones y los elementos del inicializador:

```
Lcd_Out(1, 1, txt);
char i, j, k;
const char MONTHS[12] = (31,28,31,30,31,30,31,31,30,31,30,31);
```

Punto y coma

Un punto y coma ‘;’ es un terminador de sentencia. Cada expresión válida en C (inluyendo la vacía) seguida por un punto y coma se interpreta como una sentencia, conocida como sentencia de expresión. Un punto y coma se utiliza para indicar el inicio de los comentarios en los bloques en ensamblador.

```
a + b;     /* evalúa a + b, descarta el resultado */
++a;       /* efecto lateral en 'a', se descarta el valor ++a */
;          /* expresión vacía, o una sentencia nula */
```

Dos puntos

Dos puntos ‘:’ se utilizan para indicar una etiqueta:

```
start:  x = 0;
  ...
goto start;
```

Punto

Un punto ‘.’ se utiliza para indicar el acceso a una estructura o a una unión de campo. Asimismo se puede utilizar para acceder a los bits particulares de registros en mikroC. Por ejemplo:

```
nombre.apellido = "Smith";
```

Asterisco (Declaración de punteros)

Un asterisco ‘*’ en una declaración de variable indica la creación de puntero a un tipo. Los punteros con niveles múltiples de indirección pueden ser declarados al indicar un número apropiado de asteriscos:

```
char *char_ptr;  /* declara puntero a carácter */
```

Signo igual

El signo igual '=' se utiliza para separar declaraciones de variables de las listas de inicialización así como los lados izquierdo y derecho en las expresiones de asignación

```
int test[ 5]  = {  1,  2,  3,  4,  5 };
int x = 5;
```

Almohadilla (directiva de preproceso)

La almohadilla (#) señala una directiva de preproceso cuando aparece en el primer carácter (distinto de espacio en blanco) de una línea.

Declaraciones

Una declaración introduce uno o más nombres en un programa e informa al compilador qué representa su nombre, de qué tipo es, cuáles operaciones se permiten, etc. Este capítulo repasa los conceptos relacionados con declaraciones: declaraciones, definiciones, declaraciones de especificadores e inicialización. Las entidades que se pueden declarar incluyen:
Variables;
Constantes;
Funciones;
Tipos;
Estructuras, uniones, y etiquetas de enumeraciones;
Miembros de estructuras;
Miembros de uniones;
Otros tipos de matrices;
Etiquetas de sentencias; y
Macros de preprocesador.

Declaraciones y definiciones

Al definir las declaraciones, también conocidas como definiciones, además de introducir el nombre de una entidad, se establece la creación de la entidad. Puede haber muchas referencias al mismo identificador, especialmente si el programa es multifichero, mientras que sólo se permite una definición para cada identificador. Por ejemplo:

```
/* Definición de variable i: */
int i;

/* La siguiente línea es un error, i ya está definido!*/
int i;
```

Declaraciones y declaradores

Una declaración contiene especificadores seguidos por uno o más identificadores (declaradores). Empieza con los especificadores opcionales de clase de almacenamiento, especificadores de tipo u otros modificadores. Los identificadores se separan por comas. La declaración termina en un punto y coma:

```
clase-de-almacenamiento [ cualificador-de-tipo]  type var1 [ =init1] , var2 =init2] , ... ;
```

Clase de almacenamiento

Una clase de alacenamiento determina el ámbito (visibilidad) y la duración de variables y/o funciones en un programa C. Las clases de almacenamiento que se pueden utilizar en un programa C son: auto, register, static y extern.

Auto

El especificador de clase de almacenamiento *auto* declara una variable automática (una variable con duración local). Una variable auto está visible sólo dentro de bloque en el que está declarada. El especificador de clase de almacenamiento *auto* se refiere sólo a los nombres de variables declaradas en un bloque o a los nombres de parámetros de funciones. Sin embargo, estos nombres tienen almacenamiento automático por defecto. Por esta razón el especificador de clase de almacenamiento *auto* es redundante con frecuencia en declaraciones de datos.

Register

El especificador de clase de almacenamiento *register* se utiliza para definir las variables locales que deben estar almacenadas en un registro en vez de en la RAM. Por el momento, este modificador no tiene un significado especial en mikroC. MikroC simplemente ignora solicitudes para la asignación de registros.

Static

El especificador de clase de almacenamiento *static* permite definir variables o funciones con el enlazado interno, lo que significa que cada parte de un identificador en particular representa la misma varible o función dentro de un solo fichero. Además, las variables declaradas como *static* tienen una duración de almacenamiento estática, lo que quiere decir que tienen asignada una posición de memoria desde el inicio de ejecución de progama hasta el final del mismo. La duración de almacenamiento estática de una variable es diferente del ámbito de fichero o global. Una variable puede tener duración estática y el ámbito local.

Extern

El especificador de clase de almacenamiento *extern* permite declarar objetos que se pueden utilizar en varios ficheros fuente. Una declaración externa permite utilizar una variable declarada en la siguiente parte del fichero fuente actual. La declaración no sustituye la definición. Se utiliza para describir una variable que está definida externamente.

Una declaración externa puede aparecer fuera de una función o en el principio de un bloque. Si la declaración describe una función o aparece fuera de una función y describe un objeto con enlazado externo, la palabra clave *extern* es opcional.

Si una declaración para un identificador ya existe dentro del ámbito de fichero, cualquier declaración externa del mismo identificador encontrada dentro de un bloque se refiere al mismo objeto. Si no hay otra declaración para el identificador dentro del ámbito de fichero, el identificador tiene enlazado externo.

Cualificadores de tipos

Los cualificadores de tipos en C añaden las propiedades especiales a las variables que se están declarando. mikroC proporciona dos palabras clave: const y volatile.

Const

El cualificador *const* se utiliza para indicar que el valor de variable no puede ser cambiado. Su valor se establece en la inicialización.

Volatile

El cualificador *volatile* indica que los valores de variables pueden ser cambiados sin reparar en la interferencia de usuario en el programa. El compilador no debe optimizar esta variable.

Especificador Typedef

La declaración *typedef* introduce un nombre que dentro del ámbito se convierte en un sinónimo para el tipo específicado:

```
typedef <definición_de_tipo> sinónimo;
```

Las declaraciones *typedef* se pueden utilizar para construir los nombres más cortos o con más significado para los tipos que ya están definidos por el lenguaje o por el usuario. Los nombres *typedef* permiten encapsular los detalles de implementación que se pueden cambiar.

A diferencia de las declaraciones *class, struct, union* y *enum*, las declaraciones *typedef* no introducen los tipos nuevos, sino los nombres nuevos para los tipos existentes.

Declaración asm

mikroC permite embeber las instrucciones del ensamblador en el código fuente mediante la declaración ***asm***:

```
asm {
 bloque de instrucciones del ensamblador
}
```

Una sola instrucción del ensamblador puede ser embebida en el código C:

```
asm instrucción del ensamblador;
```

Inicialización

El valor inicial de un objeto declarado se puede establecer en el tiempo de declaración. Este procedimiento es denominado inicialización. Por ejemplo:

```
int i = 1;
char *s = "hello";
struct complex c = {0.1, -0.2};
// donde 'complex' es una estructura que contiene dos objetos de tipo
float
```

Ámbito y visibilidad

Ámbito (scope)

El ámbito de un identificador representa una región de programa en la que se puede utilizar el identificador. Hay varias categorías del ámbito lo que depende de cómo y dónde se declaran los identificadores.

Visibilidad (visibilidad)

Similar al ámbito, la visibilidad representa una región de programa en la que se puede utilizar el identificador dado. Ámbito y visibilidad coinciden generalmente, si bien pueden darse circunstancias en que un objeto puede aparecer invisible temporalmente debido a la presencia de un identificador duplicado (un identificador del mismo nombre, pero del ámbito diferente). El objeto existe, pero el identificador original no puede ser utilizado para accederlo hasta que el identificador duplicado es terminado. La visibilidad no puede exceder al ámbito, pero éste puede exceder a la visibilidad.

Tipos

mikroC es estrictamente un lenguaje de tipo, lo que significa que cada variable o constante son de tipo definido antes de que se inicie el proceso de compilar. Al comprobar el tipo no es posible asignar o acceder incorrectamente a objetos.

mikroC soporta los tipos de datos estándar (predefinidos) tales como los tipos enteros con signo o sin signo de varios tamaños, matrices, punteros, estructuras y uniones etc. Además, el usuario puede definir nuevos tipos de datos al utilizar el especificador *typedef*. Por ejemplo:

```
typedef char MyType1[ 10] ;
typedef int MyType2;
typedef unsigned int * MyType3;
typedef MyType1 * MyType4;

MyType2 mynumber;
```

Tipos aritméticos

Los especificadores de tipos aritméticos incluyen las siguientes palabras clave: *void, char, int, float* y *double*, precedidas por los siguientes prefijos: *short* (corto), *long* (largo), *signed* (con signo) y *unsigned* (sin signo). Hay dos tipos aritméticos, el tipo entero y el de punto flotante.

Tipos enteros

Los tipos *char* e *int,* junto con sus variantes se consideran los tipos de datos enteros.
Sus formas se crean utilizando uno de los siguientes prefijos de los modificadores *short, long, signed* y *unsigned.* La tabla más abajo representa las características principales de tipos enteros. Palabras clave entre paréntesis pueden ser omitidas (con frecuencia).

Tipo	Tamaño en bytes	Rango
(unsigned) char	1	0 .. 255
signed char	1	- 128 .. 127
(signed) short (int)	1	- 128 .. 127
unsigned short (int)	1	0 .. 255
(signed) int	2	-32768 .. 32767
unsigned (int)	2	0 .. 65535
(signed) long (int)	4	-2147483648 .. 2147483647
unsigned long (int)	4	0 .. 4294967295

Tipos de punto flotante (fraccionarios)

Los tipos de punto flotante *float* y *double* junto con el tipo *long double* se consideran tipos de punto flotante. En la tabla más abajo se muestran las características principales de los tipos de punto flotante:

Tipo	Tamaño en bytes	Rango
float	4	$-1.5 * 10^{45} .. +3.4 * 10^{38}$
double	4	$-1.5 * 10^{45} .. +3.4 * 10^{38}$
long double	4	$-1.5 * 10^{45} .. +3.4 * 10^{38}$

Enumeraciones

El tipo de datos de *enumeración* se utiliza para representar un conjunto de valores abstracto y discreto, con los nombres simbólicos apropiados. La enumeración se declara de la siguiente manera:

```
enum colores { negro, rojo, verde, azul, violeta, blanco } c;
```

Tipo Void

El tipo ***Void*** es un tipo especial que indica la ausencia de valor. No hay objetos de ***void***; En vez de eso, el tipo ***void*** se utiliza para derivar los tipos más complejos.

```
void print_temp(char temp) {
  Lcd_Out_Cp("Temperature:");
  Lcd_Out_Cp(temp);
  Lcd_Chr_Cp(223);  // carácter de grado
  Lcd_Chr_Cp('C');
}
```

Punteros

Un puntero es una variable cuyo fin específico es almacenar direcciones de memoria de objetos. Una variable accede directamente a la dirección de memoria mientras que el puntero puede ser pensado como una referencia a la dirección. Los punteros se declaran de la manera similar a cualquier otra variable, con un asterisco '*' delante del identificador. Los punteros deben ser inicializados antes de que se utilicen. Por ejemplo, para declarar un puntero a un objeto de tipo *int* es necesario escribir:

```
int *nombre_del_puntero;    /* Puntero no inicializado */
```

Para acceder al dato almacenado en la locación de memoria a la que apunta un puntero, es necesario añadir el prefijo '*' al nombre del puntero. Por ejemplo, vamos a declarar la variable *p* que apunta a un objeto de tipo *unsigned int,* y luego vamos a asignarle el valor 5 al objeto:

```
unsigned int *p;
...
*p = 5;
```

Un puntero se puede asignar al otro puntero del tipo compatible, así que los dos apuntarán a la misma locación de memoria. Al modificar el objeto al que apunta un puntero, será modificado automáticamente el objeto al que apunta otro puntero, porque comparten la misma locación de memoria.

Punteros nulo

Un puntero nulo tiene un valor reservado que es frecuentemente pero no necesariamente el valor nulo, lo que indica que no se refiere a ningún objeto. Asignar la constante entera 0 a un puntero significa asignarle un valor nulo.

```
int *pn = 0;        /* Aquí está un puntero nulo */

/* Podemos probar el puntero de la siguiente manera */
if ( pn == 0 ) {  ... }
```

El puntero a void se difiere del puntero nulo. La siguiente declaración declara *vp* como un puntero genérico capaz de ser asignado por cualquier valor del puntero, incluyendo valor nulo.

```
void *vp;
```

Punteros a funciones

Como indica su nombre, los punteros a funciones son los punteros que apuntan a funciones.

```
int addC(char x,char y){
   return x+y;
}

int subC(char x,char y){
   return x-y;
}

int mulC(char x,char y){
   return x*y;
}


int divC(char x,char y){
   return x/y;
}


int modC(char x,char y){
   return x%y;
}

// matriz de puntero a funciones que recibe dos tipos chars y devuelve el
tipo int

int   (*arrpf[])(char,char) = {addC, subC, mulC, divC, modC};

int res;
char i;
void main() {

    for (i=0;i<5;i++){
      res = arrpf[i](10,20);
    }
}
```

Estructuras

La estructura es un tipo derivado que generalmente representa un conjunto de los miembros nombrados (o de las componentes) definido por el usuario. Estos miembros pueden ser de cualquier tipo.

Declaración e inicialización de estructuras

Las estructuras se declaran mediante la palabra clave ***struct***:

```
struct tag {lista-de-declaradores-de-miembros};
```

En este ejemplo, la etiqueta *tag* es el nombre de estructura; *lista-de-declaradores-de-miembros* es una lista de miembros de estructura, o sea, una lista de declaraciones de variables. Las variables de tipo estructurado se declaran de la misma forma que las variables de cualquier otro tipo.

Declaraciones incompletas

Las declaraciones incompletas son conocidas también como declaraciones anticipadas o adelantadas. Un puntero a un tipo de estructura puede aparecer regularmente en la declaración de otro tipo de estructura aún antes de que el tipo de estructura apuntada haya sido declarado.

```
struct A;  // declaración incompleta
struct B {struct A *pa;};
struct A {struct B *pb;};
```

La primera aparición de A se denomina incompleta porque todavía no existe definición para ella. Una declaración incompleta sería correcta aquí ya que la definicón de B no requiere conocer el tamaño de A.

Estructuras sin nombres y Typedefs

Si se omite el nombre de estructura, aparece una estructura sin nombre. Las estructuras sin nombres pueden utilizarse para declarar identificadores en la lista-de-declaradores-de-miembros-delimitados por comas para que sean (o deriven) del tipo de estructura dado. Los objetos adicionales no se pueden declarar en ningún otro sitio. Es posible crear un *typedef* al mismo tiempo que se declara una estructura, con o sin nombre:

```
/* Con nombre: */
typedef struct mystruct {  ... } Mystruct;
Mystruct s, *ps, arrs[10];  /* igual que struct mystruct s, etc. */

/* Sin nombre: */
typedef struct {  ... } Mystruct;
Mystruct s, *ps, arrs[10];
```

Tamaño de estructura

El tamaño de estructura en la memoria se puede determinar mediante el operador *sizeof*. No es necesario que el tamaño de la estructura sea igual a la suma de tamaños de sus miembros. A menudo es más grande debido a ciertas limitaciones de almacenamiento de memoria.

Asignación

Las variables del mismo tipo estructurado pueden ser asignadas una a otra mediante el operador de asignación simple (=), lo que copiará todo el contenido de variables a sus destinaciones a pesar de la complejidad interior de la estructura. Fíjese que dos variables son del mismo tipo de estructura si las dos están definidas por la misma declaración o utilizando el mismo identificador de tipos. Por ejemplo:

```
/* a and b are of the same type: */
struct {int m1, m2;} a, b;

/* Pero c y d no _son_ del mismo tipo aunque las descripciones de sus
estructuras son idénticas: */
struct {int m1, m2;} c;
struct {int m1, m2;} d;
```

Acceso a miembros de estructuras

Para acceder a los miembros de estructuras y uniones se utilizan las dos siguientes operadores de selección:
. (punto); y
-> (flecha a la derecha).

El operador '.' se llama selector directo de miembro. Se utiliza para acceder directamente a uno de miembros de estructura.

```
s.m   // direct access to member m
```

El operador -> se llama el selector indirecto (de puntero) de miembro.

```
ps->m     // acceso indirecto a miembro m;
          // igual que (*ps).m
```

Tipos de uniones

Los tipos de uniones son los tipos derivados, con cierta similitud sintáctica y funcional con los tipos de estructura. La diferencia principal es que los miembros de uniones comparten el mismo espacio de memoria.

Declaración de uniones

Las uniones tienen la misma declaración que las estructuras, con la palabra clave *union* utilizada en vez de *struct:*

```
etiqueta de unión {lista-de-declaradores-de-miembros};
```

Campos de bits

Los campos de bits son grupos de un número determinado de bits, que pueden o no tener un identificador asociado. Los campos de bits permiten subdividir las estructuras en las partes nombradas de tamaños definidos por el usuario.

Declaración de campos de bits

Los campos de bits se pueden declarar sólamente en las estructuras y uniones. La estructura se declara normalmente y se le asignan los campos individuales de la siguiente manera (los campos deben ser de tipo *unsigned*):

```
struct tag {
   lista-de-declaradores-de-campos-de-bits sin signo;
}
```

Es este ejemplo, la etiqueta *tag* es el nombre opcional de la estructura mientras que *lista-de-declaradores-de-campos-de-bits* es la lista de campos de bits. Cada identificador de campos de bits requiere dos puntos y su ancho en bits tiene que ser especificado explícitamente.

Acceso a los campos de bits

A los campos de bits se les puede acceder de la misma manera que a los miembros de estructura utilizando el selector directo e indirecto de miembro '.' y '->'.

```
// Este ejemplo escribe la estructura Value_For_PORT0 to P0
struct myPort {
   lo_nibble :4;
   hi_nibble :4;
} Value_For_PORT0;

void main() {
   ...
   Value_For_PORT0.lo_nibble  = 7;
   Value_For_PORT0.hi_nibble  = 0x0C;
   P0 = *(char *) (void *)&Value_For_PORT0;
            // typecasting :
            // 1. dirreción de estructura a puntero a void
            // 2. puntero a void a puntero a char
            // 3. dereferenciar para obtener el valor
}
```

Matrices

La matriz es el tipo estructurado más simple y más utilizado. Una variable de tipo de matriz es realmente una matriz de los objetos del mismo tipo. Estos objetos son los elementos de una matriz identificados por su posición en la misma. Una matriz representa una zona de almacenamiento contiguo, suficiente grande para contener todos sus elementos.

Declaración de matrices

La declaración de matrices es similar a la declaración de variables, con los corchetes añadidos después del identificador:

```
tipo_de_elemento nombre_de_matriz [ constante-expresión] ;
```

Una matriz nombrada como *nombre_de_matriz* y compuesta de los elementos de *tipo_de_elemento* se declara aquí. El *tipo_de_elemento* puede ser de cualquier tipo escalar a excepción de *void,* tipo *definido por el usuario, puntero, enumeración* u otra matriz. El resultado de la *constante - expresión* entre corchetes determina un número de elementos de la matriz. Ejemplos de la declaración de matrices:

```
#define MAX = 50
int    vector_one[ 10] ;          /* declara una matriz de 10 elementos enteros */
float  vector_two[ MAX] ;         /* ídem de 50 elementos flotantes   */
float  vector_three[ MAX - 20] ;  /* ídem de 30 elementos flotantes  */
```

Inicialización de matrices

Una matriz puede ser inicializada en la declaración al asignarle una secuencia de valores delimitados por comas, entre llaves. Al inicializar una matriz en la declaración, se puede omitir el número de elementos puesto que será determinado automáticamente a base del número de los elementos asignados. Por ejemplo:

```
/* Declarar una matriz que contiene el número de días de cada mes */
int days[ 12]  = { 31,28,31,30,31,30,31,31,30,31,30,31} ;
```

Matrices multidimensionales

Una matriz es unidimensional si es de tipo escalar. A veces las matrices unidimensionales se denominan vectores. Las matrices multidimensionales se crean al declarar matrices de tipo matriz. El ejemplo de una matriz bidimensional:

```
char m[ 50][ 20] ;    /* matriz bidimensional de tamaño 50x20 */
```

La variable *m* representa una matriz de 50 elementos. Cada uno representa una matriz de los elementos de tipo *char*. Así es creada una matriz de 50x20 elementos. El primer elemento es m[0][0], mientras que el último es m[49][19].
Una matriz multidimensional se puede inicializar con un conjunto de valores apropiado entre corchetes. Por ejemplo:

```
int a[ 3][ 2]  = {{ 1,2} , { 2,6} , { 3,7}} ;
```

Conversión de tipos

La conversión es un proceso de cambiar el tipo de variable. mikroC soporta las conversiones implícita y explícita.

Conversión implícita

La conversión automática de un valor de un tipo de datos al otro por un lenguaje de programación, sin hacerlo específicamente por el programador, es denominada conversión de tipos implícita. El compilador realiza la conversión implícita en los siguientes casos:

- Un operando está preparado para una operación aritmética o lógica;
- Una asignación se ha hecho a un valor de un tipo diferente del valor asignado;
- Una función está proporcionada con el valor de argumento que es de un tipo diferente del parámetro;
- El valor devuelto de la función es de tipo diferente del tipo de valor que debe ser devuelto (tipo de valor devuelto predefinido).

Promoción

Cuando los operandos son de tipos diferentes, mediante la conversión implícita se realiza la promoción de tipo más bajo a tipo más alto, de la siguiente manera:

short → int, long, float, or double
char → int, long, float, or double
int → long, float, or double
long → float or double

Los bytes más altos del operando sin signo extendido se llenan de ceros. Los bytes más altos del operando con signo extendido se llenan del signo de bit. Si el número es negativo, los bytes más altos se llenan de unos, de lo contrario se llenan de ceros. Por ejemplo:

```
char a;
unsigned int b;
...
a = 0xFF;
b = a;  // promoción de a a unsigned int, b equivale a 0x00FF
```

Recorte

En las sentencias de asignación y en las sentencias que requieren una expresión de tipo particular, el valor correcto será almacenado en el destino sólo si el resultado de expresión no excede al rango del destino. Por el contrario, los datos que sobran, o sea, los bytes más altos se pierden.

```
char i;
unsigned int j;
...
j = 0xFF0F;
i = j;        // i obtiene el valor 0x0F, byte más alto 0xFF se pierde
```

Conversión explícita

La conversión explícita se puede realizar sobre cualquier expresión al utilizar el operador unitario de conversión de tipos. Un caso especial es conversión entre los tipos con signo y sin signo.

```
short b;
char a;
...
b = -1;
a = (char)b;   // a es 255, no 1
               //Es porque el dato se queda en la representación binaria
               // 11111111; pero el compilador la interpreta de la manera diferente
```

Tal conversión explícita no cambia la representación binaria de los datos. Por ejemplo:
La conversión explícita no se puede realizar sobre el operando que está a la izquierda del

```
(int)b = a;    // Compilador informa acerca de un error
```

operador de asignación:

```
char a, b, c;
unsigned int cc;
...
a = 241;
b = 128;

c  = a + b;                    // es igual a  113
c  = (unsigned int)(a + b);    // es igual a  369
cc = a + b;                    // es igual a  369
```

Ejemplo de la conversión explícita:

Funciones

Las funciones son los subprogramas que realizan ciertas tareas a base de varios parámetros de entrada. Las funciones pueden devolver un valor después de la ejecución. La función devuelve un valor después de la ejecución. El valor devuleto de la función se puede utilizar en las expresiones. Una llamada a función se considera una expresión como cualquier otra. Cada programa debe tener una sola función llamada ***main*** que marca el punto de entrada del programa.
Las funciones se pueden declarar en los ficheros de cabecera estándar o proporcionados por el usuario, o dentro de ficheros de programa.

Declaración de función

Las funciones se declaran en los ficheros fuentes del usuario o se ponen disponibles al enlazar las librerías precompiladas. La sintaxis de declaración de función es:

```
tipo_de_resultado nombre_de_funcion name(lista-de-declaradores-de-
parámetros));
```

nombre_de_función representa el nombre de función y debe ser un identificador válido. *tipo_de_resultado* representa el tipo del resultado de función que puede ser cualquier tipo estándar o definido por el usuario.

Prototipos de función

Una función puede ser definida sólo una vez en el programa, pero puede ser declarada varias veces, en el supuesto de que las declaraciones sean compatibles. Los parámetros pueden tener los nombres diferentes en las declaraciones diferentes de la misma función:

```
/* Aquí están dos prototipos de la misma función: */

int test(const char*)    /* declara la pruba de función */
int test(const char*p)   /* declara la misma prueba de función */
```

Definición de función

La definición de función consiste en su declaración y su cuerpo. El cuerpo de función es realmente un bloque - una secuencia de las definiciones locales y sentencias encerradas entre corchetes {}. Todas las variables declaradas dentro del cuerpo de función son locales a la función, o sea, tienen el ámbito de función. Ejemplo de una definición de función:

```
/* la función max devuelve el mayor argumento: */

int max(int x, int y) {
  return (x>=y) ? x : y;
}
```

Llamada a función

Una llamada a función se realiza al especificar su nombre seguido por los parámetros actuales colocados en el mismo orden que los parámetros formales correspondientes. Si hay una llamada a función en una expresión, el valor devuelto de función se utilizará como un operando en dicha expresión.

```
nombre_de_función (expresión_1, ... , expresión_n);
```

Cada expresión en la llamada a función es un argumento actual.

Conversión de argumentos

El compilador es capaz de forzar a los argumentos cambiar su tipo en otro adecuado de acuerdo con las reglas de conversión implícita.

```
int limit = 32;
char ch = 'A';
long res;

// prototipo
extern long func(long par1, long par2);

main() {
   ...
   res = func(limit, ch);  // llamada a función
}
```

Ya que el programa tiene el prototipo de función *func*, convierte *limit y ch* en *long,* utilizando las reglas estándar de asignación antes de pasarlos al parámetro formal apropiado.

Operador Ellipsis

El operador ellipsis '...' consiste en tres puntos sucesivos sin intervención de espacios en blanco. Un ellipsis se puede utilizar en las listas de argumentos formales de prototipos de funciones para indicar un número variable de argumentos o de argumentos con tipos distintos. Por ejemplo:

```
void func (int n, char ch, ...);
```

Esta declaración indica que *func* se define de tal manera que las llamadas tengan obligatoriamente por lo menos dos argumentos (*int* y *char)*, pero también pueden tener cualquier número de argumentos adicionales.

Operadores

Los operadores son un tipo de tokens que indican las operaciones que se realizan sobre los operandos en una expresión. Si el orden de ejecución de operaciones no es determinado explícitamente madiante paréntesis, lo determinará el operador de precedencia.

- Operadores aritméticos
- Operador de asignación
- Operador de manejo de bits
- Operadores lógicos
- Operadores de indirección/referencia
- Operadores relacionales
- Selectores de miembros de estructura
- Operador coma,
- Operador condicional ? :
- Operador de subíndice de matriz []
- Operador de llamada a función ()
- Operador Sizeof
- Operadores de preproceso # y ##

Presedencia y asociatividad de operadores

Hay 15 categorías de presedencia en mikroC. Los operadores de la misma categoría tienen igual presedencia. Cuando aparecen operadores duplicados en la tabla, el operador de la precedencia más alta es unitario. Cada categoría tiene una regla de asociatividad: de izquierda a derecha o de derecha a izquierda. En ausencia de paréntesis, estas reglas se aplican al agrupar las expresiones con operadores de igual presedencia.

Precedencia	Operandos	Operadores	Asociatividad
15	2	() [] . ->	→
14	1	! ~ ++ -- + - * & (tipo) sizeof	←
13	2	* / %	→
12	2	+ -	→
11	2	<< >>	→
10	2	< <= > >=	→
9	2	== !=	→
8	2	&	→
7	2	^	→
6	2	|	→
5	2	&&	→
4	2	||	→
3	3	?:	←
2	2	= *= /= %= += -= &= ^= |= <<= >>=	←
1	2	,	→

Operadores aritméticos

Los operadores aritméricos se utilizan para realizar las operaciones matemáticas. Todos los operadores aritméticos se asocian de izquierda a derecha.

Operador	Operación	Precedencia
Operadores binarios		
+	suma	12
-	resta	12
*	multiplicación	13
/	división	13
%	Operador de módulo devuelve el resto de la división de enteros (no puede ser utilizado con números fraccionarios)	13
Operadores unitarios		
+	Más unitarios no afecta al operando	14
-	Menos unitario cambia el signo del operando	14
++	El incremento añade uno al valor del operando. El postincremento añade uno al operando después de que haya sido evaluado, mientras que el preicremento añade uno antes de la evaluación del operando.	14
--	El decremento resta uno del valor del operando. El postdecremento resta uno del valor del operando después de que haya sido evaluado, mientras que el predecremento resta uno antes de la evaluación del operando.	14

Operadores relaciones

Los operadores relacionales se utilizan para comprobar la veracidad o falsedad de expresiones. Si una expresión es cierta, devuleve 1, al contrario devuleve 0. Todos los operadores relacionales se asocian de izquierda a derecha.

Operador	Operación	Precedencia
==	igual que	9
!=	desigual que	9
>	mayor que	10
<	menor que	10
>=	mayor o igual que	10
<=	menor o igual que	10

Opedarodes de manejo de bits

Los operadores de manejo de bits se utilizan para modificar los bits individuales de un operando. Los operadores de manejo de bits realizan desplazamiento de izquierda a derecha. La única excepción es el complemento a uno '~' que realiza desplazamiento de derecha a izquierda.

Operador	Operación	Precedencia
&	operador AND; compara pares de bits y devuleve 1 si los ambos bits están a 1, en caso contrario devuelve 0.	8
|	operador inclusivo OR; compara pares de bits y devuelve 1 si uno o los ambos bits están a 1, en caso contrario devuelve 0.	6
^	operador exclusivo OR (XOR); compara pares de bits y devuelve 1 si los ambos bits son complementarios, en caso contrario devuelve 0.	7
~	complemento a uno (unitario); invierte cada bit	14
<<	desplazamiento a izquierda; desplaza los bits a izquierda, el bit más a la izquierda se pierde y se le asigna un 0 al bit más a la derecha.	11
>>	Desplazamiento a derecha; desplaza los bits a derecha, el bit más a la derecha se pierde y se le asigna un 0 al bit más a la izquierda, en caso contrario el signo de byte no cambia.	11

Operadores lógicos

Los operandos de las operaciones lógicas se consideran ciertos o falsos, según su valor sea 0 o distinto de 0. Los operadores lógicos siempre devuleven 1 o 0. Los operadore lógicos && y || se asocian de izquierda a derecha. Operador de negación lógica ! se asocia de derecha a izquierda.

Operador	Operación	Precedencia
&&	Y lógico (AND)	5
||	O lógico (OR)	4
!	negación lógica	14

Operador condicional

El operador condicional '?:'es el único operador ternario en mikroC. La sintaxis del operador condicional es:

```
expresión1 ? expresión2 : expresión3;
```

expresión1 se evalúa primero. Si el resultado es cierto, entonces se evalúa la *expresión2*, mientras que la *expresión3* se ignora. Si *expresión1* resulta falsa se evalúa la *expresión2* mientras que la *expresión2* se ignora. El resultado será el valor de la *expresión2* o la *expresión3*, dependiendo de cuál se evalúa.

Operador de asignación simple

El operador se asignación simple '=' se utiliza para la asignación de valor común:

```
expresión1 = expresión2;
```

expression1 es un objeto (locación de memoria) al que se le asigna el valor de la *expresión2.*

Operador de asignación compuesta

mikroC permite la asignación más compleja por medio del operador de asignación compuesta. La sintaxis de asignación compuesta es:

```
expresión1 op= expresión2;
```

donde *op* puede ser uno de los operadores binarios +, -, *, /, %, &, |, ^, <<, or >>.

La asignación compuesta más arriba es una forma corta de la siguiente expresión:

```
expresión1 = expresión1 op expresión2;
```

Operadores de desplazamiento de bits

Hay dos operadores de desplazamiento de bits en mikroC. Son el operador << que realiza un desplazamiento de bits a la izquierda y el operador >> que realiza un desplazamiento de bits a la derecha. Cada uno de los operadores tiene dos operandos. El operando izquierdo es un objeto que se desplaza, mientras que el derecho indica el número de desplazamientos que se realizarán. Los dos operandos deben ser de tipo *integral*. El operando derecho debe ser el valor positivo. Al desplazar a la izquierda los bits que salen por la izquierda se pierden, mientras que los 'nuevos' bits a la derecha se rellenan con ceros. Por lo tanto, el desplazamiento del operando sin signo a la izquierda por n posiciones equivale a multiplicarlo por 2^n si todos los bits descartados son ceros. Lo mismo se puede aplicar a los operandos con signo si todos los bits descartados son iguales que el signo de bit. Al desplazar a la derecha los bits que salen por la derecha se pierden, mientras que los 'nuevos' bits a la izquierda se rellenan con ceros (en caso del operando sin signo) o con el signo de bit (en caso del operando con signo). El desplazamiento del operando a la derecha por n posiciones equivale a dividirlo por 2^n.

Operador	Operación	Precedencia
<<	Desplazamiento a izquierda	11
>>	Desplazamiento a derecha	11

Expresiones

Una expresión es una secuencia de operadores, operandos y puntuadores que devuelven un valor. Las expresiones primarias son: literales, constantes, variables y llamadas a función. Se pueden utilizar para crear las expresiones mas complejas por medio de operadores. La forma de agrupación de los operandos y de las subexpresiones no representa obligatoriamente el orden en el que se evalúan en mikroC.

Expresiones con coma

Una característica especial en mikroC es que permite utilizar coma como operador de secuencias para formar así llamadas expresiones con coma o secuencias. En la siguiente secuencia:

```
expresión_1, expresión_2, ... expresión_n;
```

cada expresión se evalúa de izquierda a derecha. El valor y el tipo de la expresión con coma equivale al valor y al tipo de *expresión_n*.

Sentencias

Las sentencias especifican y controlan el flujo de ejecución del programa. En ausencia de las sentencias de salto y de selección, las sentencias se ejecutan en el orden de su aparición en el código de programa. En líneas generales, las sentencias se pueden dividir en:

- Sentencias de etiqueta;
- Sentencias de expresión;
- Sentencias de selección;
- Sentencias de iteración (Bucles);
- Sentencias de salto; y
- Sentencias compuestas (Bloques).

Sentencias de etiqueta

Cada sentencia en programa puede ser etiquetada. Una etiqueta es un identificador colocado delante de la sentencia de la siguiente manera:

```
identificador_de_etiqueta: sentencia;
```

La misma etiqueta no se puede redefinir dentro de la misma función. Las etiquetas tienen sus propios espacios de nombres y por eso el identificador de etiqueta puede corresponder a cualquier otro identificador en el programa.

Sentencias de expresión

Cualquier expresión seguida por punto y coma forma una sentencia de expresión:

```
expresión;
```

Un caso especial es la sentencia nula; consiste en un solo punto y coma (;). Una sentencia nula se utiliza con frecuencia en los bucles 'vacíos':

```
for (; *q++ = *p++ ;)
  ;  /* el cuerpo de este bucle es una sentencia nula */
```

Sentencias condicionales

Las sentencias condicionales o las sentencias de selección permiten decidir entre distintos cursos de acción en función de ciertos valores.

Sentencia If

La sentencia **If** es una sentencia condicional. La sintaxis de la sentencia **if** es la siguiente:

```
if some_expression statement1; [ else statement2;]
```

Si *some_expression* se evalúa como cierta, se ejecuta *statement1*. Por el contrario, se ejecuta *statement2*. La palabra clave *else* con la sentencia alternativa *(statement2)* es opcional.

Sentencias If anidadas

Sentencias *If* anidadas requieren atención especial. La regla general es que se descomponen empezando por la sentencia *If* más interior (más anidada), mientras que cada sentencia *else* se relaciona con la *If* más cercana y disponible a su izquierda:

```
if (expression1) statement1;
else if (expression2)
if (expression3) statement2;
else statement3;         /* esto pertenece a: if (expression3) */
else statement4;         /* esto pertenece a: if (expression2) */
```

Sentencia Switch

La sentencia *switch* es una sentencia condicional de ramificación múltiple, basada en una cierta condición. Consiste en una sentencia de control (selector) y una lista de los valores posibles de la expresión. La sintaxis de la sentencia *switch* es la siguiente:

```
switch (expression) {
  case constant-expression_1 : statement_1;
    ...
  case constant-expression_n : statement_n;
  [default : statement;]
}
```

Primero se evalúa la *expression* (condición). Entonces la sentencia *switch* la compara con todas las constantes-expresiones disponibles que siguen la palabra clave *case*. Si hay una coincidencia, *switch* pasa el control a la constante-expresión correspondiente después de que se ejecuta la siguiente sentencia.

Sentencias de iteración

Las sentencias de iteración permiten repetir un conjunto de sentencias.

Sentencia For

La sentencia *for* se utiliza para implementación del bucle iterativo cuando el número de iteraciones está especificado. La sintaxis de la sentencia *for* es la siguiente:

```
for ([init-expression]; [condition-expression]; [increment-expression]) statement;
```

Antes de la primera interación del bucle, *init-expression* pone las variables a los valores iniciales. En *init-expression* no es posible declarar. *Condition-expression* se comprueba antes de la primera ejecución del bloque. La sentencia *for* se ejecuta repetidamente hasta que el valor de *condition-expression* sea falso. Después de cada iteración del bucle, *increment-expression* incrementa el contador de bucle.
Ejemplo de calcular el producto escalar de dos vectores, utilizando la sentencia *for:*

```
for ( s=0, i=0; i<n; i++ ) s+= a[i] * b[i];
```

Sentencia While

La sentencia *while* se utiliza para implementación del bucle iterativo cuando el número de iteraciones no está especificado. Es necesario comprobar la condición de iteración antes de la ejecución del bucle. La sintaxis de la sentencia *while* es la siguiente:

```
while (cond_exp) some_stat;
```

La sentencia *some_stat* se ejecuta repetidamente siempre que el valor de la expresión *cond_exp* sea cierto. El valor de la expresión se comprueba antes de que se ejecute la siguiente iteración. Por eso, si el valor de la expresión es falso antes de entrar el bucle, no se ejecutará ninguna iteración, Probablemente la forma más simple de crear un bucle infinito sea la siguiente:

```
while (1) ...;
```

Sentencia Do

La sentencia *do* se utiliza para implementación del bucle iterativo cuando el número de iteraciones no es especificado. La sentencia se ejecuta repetidamente hasta que la expresión sea cierta La sintaxis de la sentencia *do* es la siguiente:

```
do some_stat; while (cond_exp);
```

La sentencia *some_stat* se ejecuta repetidamente hasta que el valor de la expresión *cond_exp* llegue a ser cierto. La expresión se evalúa después de cada iteración así que el bucle ejecutará la sentencia por lo menos una vez.

Sentencias de salto

mikroC soporta las siguientes sentenicas de salto: *break, continue, return* and *goto*.

Sentencia Break

A veces es necesario detener el bucle dentro del cuerpo. La sentencia *break* dentro de bucles se utiliza para pasar el control a la primera sentencia después del bucle respectivo. La sentencia *break* se usa con frecuencia en las sentencias *switch* para detener su ejecución después de que ocurra la primera coincidencia positiva. Por ejemplo:

```
switch (state) {
  case 0: Lo();  break;
  case 1: Mid(); break;
  case 2: Hi();  break;
  default: Mensaje("estado inválido!");
}
```

Sentencia Continue

La sentencia *continue* dentro del bucle se utiliza para iniciar una nueva iteración del bucle. Las sentencia que siguen la sentencia *continue* no se ejecutarán.

```
while (..) {
...
if (val>0) continue;
...
// continue salta aquí
}
```

```
do {
...
if (val>0) continue;
...
// continue salta aquí
while (..);
```

```
for (..;..;..) {
...
if (val>0) continue;
...
// continue salta aquí
}
```

Sentencia Return

La sentencia *return* se utiliza para salir de la función actual devolviendo opcionalmente un valor. La sintaxis es:

```
return [expresión];
```

La expresión se evalúa y devuelve el resultado. El valor devuelto se convierte automáticamente en el tipo de función esperado, si es necesario. La expresión es opcional.

Sentencia Goto

La sentencia *goto* se utiliza para saltar de forma incondicional a la parte apropiada de programa. La sintaxis de la sentencia *goto* es:

```
goto identificador_de_etiqueta;
```

Esta sentencia ejecuta un salto a la etiqueta *identificador_de_etiqueta*. La sentencia *goto* puede estar delante o detrás de la declaración de la etiqueta. La declaración de etiqueta y la sentencia *goto* deben pertenecer a la misma rutina. Por consiguiente, no es posible saltar a un procedimiento o una función o de ellos. La sentencia *goto* se puede utilizar para salir de cualquier nivel de las estructuras anidadas. No es recomendable saltar a bucles u otras sentencias estructuradas, ya que se pueden producir resultados inesperados. En general no es recomendable utilizar la sentencia *goto* dado que prácticamente cada algoritmo se puede realizar sin ella dando los programas estructurados legibles.
No obstante, la sentencia *goto* es útil para salir de las estructuras de control profundamente anidadas.

```
for (...) {
  for (...) {
    ...
    if (disaster)
      goto Error;
    ...
  }
}
 .
 .
 .
Error: // error handling code
```

Preprocesador

El preprocesador es un procesador de texto integrado que prepara el código fuente para la compilación. El preprocesador permite:

- insertar un texto de un fichero específico en un punto determinado en el código. (Inclusión de ficheros)
- sustituir los símbolos léxicos específicos por otros símbolos (Macros); y
- compilación condicional que incluye u omite condicionalmente las partes del código (compilación condicional)

Directivas de preproceso

Cualquier línea en el código fuente precedida por el símbolo # se considera directiva de preproceso (o una línea de control), a menos que # esté incluido en una cadena literal, una constante carácter o integrado en un comentario. mikroC soporta las directivas de preproceso estándar:

# (null directive)	#error	#ifndef
#define	#endif	#include
#elif	#if	#line
#else	#ifdef	#undef

Inclusión de ficheros

La directiva *#include* copia un fichero especificado en el código fuente. La sintaxis de la directiva *#include* tiene uno de dos formas:

```
#include <nombre_de_fichero>
#include "nombre_de_fichero"
```

El preprocesador sustituye la línea *#include* por el contenido de un fichero específicado en el código fuente. Por eso, la colocación de *#include* puede afectar al ámbito y a la duración de todos los identificadores en el fichero incluido. La diferencia entre estas dos formas yace en encontrar algoritmo utilizado para intentar a localizar el fichero a incluir.

Ruta explícita

Colocar una ruta explícita en *nombre_de_fichero,* significa que se buscará sólo dicha carpeta. Por ejemplo:

```
#include "C:\my_files\test.h"
```

Macros

Las macros proporcionan un macanismo de reemplazo de tokens, antes de la compilación.

Definir Macros y expansiones de macros

La directiva # *define* define una macro:

```
#define identificador_de_macro <secuencia_de_tokens>
```

Cada ocurrencia de *identificador_de_macro* en el código fuente precedida por esta línea de control será reemplezada por una *secuencia_de_tokens* (probablemente vacía).

Macros con Parámetros

La siguiente sintaxis se utiliza para definir una macro con parámetros:

```
#define identificador_de_macro (<arg_list>)<secuencia_de_tokens>
```

arg_list opcional es una secuencia de los identificadores separados por comas, como la lista de argumentos de una función en C. Cada identificador separado por comas tiene la función de un argumento formal o un marcador de posición. Un simple ejemplo:

```
/* Una simple macro que devuelve el mayor argumento: */
#define _MAX(A, B) ((A) > (B)) ? (A) : (B)
```

Compilación condicional

Las directivas de compilación condicional se utilizan generalmente para facilitar las modificaciones y compilaciones de programas fuente en diferentes entornos de ejecución.

Directivas #if, #elif, #else, y #endif

Las directivas condicionales *#if, #elif, #else*, y *#endif* se comportan igual que las sentencias las sentencias condicionales comunes en mikroC. Si la expresión escrita después de *#if* es cierta *(*tiene un valor distinto de cero), las líneas de código `<sección_1>` que siguen inmediatamente a la directiva *#if* se retienen en la unidad de compilación. La sintaxis es:

```
#if constant_expression_1
<sección_1>

[ #elif constant_expression_2
<section_2>]
   ...
[ #elif constant_expression_n
<section_n>]

[ #else
<final_section>]

#endif
```

Cada directiva *#if* en el código fuente debe terminar por una directiva de cierre *#endif*. Puede haber cualquier número de *#elif* entre las directivas *#if* y *#endif,* pero sólo se permite una directiva *#else*. La directiva *#else*, si está presente, precede a *#endif.*

Directivas #ifdef y #ifndef

Las directivas *#ifdef* y *#ifndef* son condicionales especializadas para comprobar si un identificador está definido o no. La siguiente línea:

```
#ifdef identifier
```

tiene exactamente el mismo efecto que *#if 1* si el identificador está actualmente definido, o *#if 0* si el identificador está actualmente indefinido. La directiva *#ifndef* comprueba la condición opuesta comprobada por *#ifdef.*

MikroElektronika
DEVELOPMENT TOOLS I COMPILERS I BOOKS

Si quiere saber más de nuestros productos, por favor visite nuestra página web www.mikroe.com
Si tiene problemas con cualquiera de nuestros productos o sólo necesita información adicional, deje un ticket en www.mikroe.com/en/support
Si tiene alguna pregunta, comentario o propuesta de negocio, póngase en contacto con nosotros en office@mikroe.com