# Especificación del lenguaje ccZ80++

http://ccz80pp.webcindario.com

# Índice

# Introducción

El lenguaje ccZ80++ permite escribir programas para ordenadores basados en el microprocesador Intel Z80, como Amstrad CPC, Amstrad PCW, Spectrum y MSX entre los más populares.

Debe utilizarse en un sistema Windows o también desde Linux utilizando WINE o Mono o desde Mac utilizando Mono. Necesita que se encuentre instalado Microsoft .NET 4 Client Profile u otra versión superior, que se puede descargar en http://www.microsoft.com/en-us/download/details.aspx?id=24872.

Se puede considerar una evolución del lenguaje ccZ80, pero ccZ80++ añade definición de clases, uso de objetos y una sintaxis más cercana al lenguaje C++ para permitir un más rápido aprendizaje.

Se diferencian mayúsculas y minúsculas en las palabras claves y en los identificadores definidos en el programa. No se diferencian para los nombres de ficheros dentro del entorno Windows.

Aunque se recomienda escribir el código con un formato estructurado, los saltos de línea, líneas en blanco y espacios no afectan a la generación del código objeto.

Para tener una primera impresión de este lenguaje a continuación se da su versión de "Hola mundo":

```
include Text.ccz80++

class ClasePrincipal
{
  static void main()
  {
    Text.PrintString("Hola Mundo");
  }
}
```

Este primer programa hace uso de una clase Text y de su miembro estático printString, incluida en el fichero Text.ccz80++, de la cual se proporcionan versiones para Amstrad CPC 464, Amstrad CPC 6128, Spectrum y MSX en la sección de descargas de la página web.

# Sintaxis del compilador

El compilador para ccZ80++ se utiliza desde la línea de comando, produciendo un fichero objeto resultado de la compilación del tipo indicado en el parámetro correspondiente. Este fichero se crea, sobrescribiendose si existiese previamente, con el mismo nombre que el fichero fuente principal y con una de las siguientes extensiones, según el tipo de resultado solicitado al compilador:

- **.cc80** si se ha indicado una salida de código ccz80.
- **.asm** si se ha indicado una salida de código ensamblador.
- **.bin** si se ha indicado una salida de código máquina.

Al finalizar la compilación se devuelve en la variable ERRORLEVEL un valor 0 si ha sido correcta, o un valor 1 si ha ocurrido algún error que impida la compilación.

Toda la salida de información producida por la compilación se muestra en la misma ventana de línea de comandos desde la que se invoca al compilador.

El compilador está completamente integrado en el fichero ccz80++.exe que se puede obtener en la sección de descargas de la página web.

La sintaxis del compilador es:

```
ccz80++ fichero /org=dirección /out=tipo /msg=nivel /path=rutas
```

El orden de los parámetros dados al compilador y si se indican en mayúsculas o minúsculas no es significativo.

Además de ccz80++ que es el nombre del fichero correspondiente al compilador, el significado de cada elemento es:

**fichero**

Obligatorio. Es el nombre del fichero fuente principal del programa, recomendado que tenga extensión .ccz80++. No es necesario que éste sea el que contenga la clase contenedora de la función main, puede incluir otros ficheros, uno de los cuales contener la clase con la función main.

**/org=dirección**

Opcional. Permite especificar una dirección de inicio para la generación del código binario resultante de la compilación, donde dirección es la dirección de inicio que puede especificarse en decimal o en hexadecimal con el prefijo #. Si no se indica este parámetro se genera el código a partir de la dirección 0.

**/out=tipo**

Opcional. Permite especificar el tipo de resultado dado por el compilador, donde tipo puede ser:

- **ccz80** para generar código ccz80.
- **assembler** para generar código ensamblador.
- **binary** para generar código máquina.

Si no se indica este parámetro se genera código máquina.

**/msg=nivel**

Permite especificar el nivel de mensajes que mostrará el compilador. Existen tres niveles, de menor a mayor importancia:

- notas, que informan sobre circunstancias que no impiden la compilación ni perjudican la velocidad o tamaño del código máquina resultante.
- avisos, que informan sobre circunstancias que no impiden la compilación, pero pueden perjudicar la velocidad o tamaño del código máquina resultante.
- errores, que impiden la generación del código máquina.

Como nivel se puede indicar:

- **all**: muestra los mensajes de todos los niveles, notas, avisos y errores.
- **warning**: muestra sólo los mensajes de nivel aviso y error.
- **error**: muestra sólo los mensajes de nivel error.

Si no se indica este parámetro se muestran los mensajes de todos los niveles.

**/path=rutas**

Permite indicar una lista de rutas donde el compilador debe buscar los ficheros fuente, tanto el indicado al compilador como fichero fuente principal como los ficheros indicados en las sentencias include dentro del programa.

Se pueden dar varias rutas, separadas por punto y coma.

Si el fichero fuente principal o un ficheros especificado con include se indica con una ruta completa no se hace uso de las rutas determinadas por esta opción; si se indican con una ruta relativa, el fichero con su ruta relativa se busca en principio a partir de la ruta actual de la línea de comandos y después en todas las rutas dadas por esta opción, en el orden en que se han escrito.

Ejemplos de uso del compilador:

```
ccz80++ C:\Fuente\Juegos\asteroid.ccz80++ /org=#4000 /msg=warning
ccz80++ "Ficheros fuente\programa.ccz80++" /org=5000 /out=assembler
ccz80++ ..\prueba.ccz80++ "/path=C:\Source;C:\ccz80++\Fichero include"
```

Hay que recordar que si algún parámetro contiene espacios, como el nombre del fichero fuente principal, o las rutas del parámetro path, el parámetro hay que indicarlo encerrado entre comillas dobles.

## Gestión de errores de compilación

En los errores que detecta el compilador se informa de la línea, posición y fichero donde se encuentra el error. Hay que tener en cuenta que la posición se informa sobre el número de carácter en la línea donde se ha producido el error, por lo que si la línea contiene tabulaciones puede ser que no coincida la posición indicada en el error con la columna en el editor de texto que se esté utilizando, ya que una tabulación es una posición, pero puede corresponder a varios espacios en el editor de texto.

En algunas ocasiones, si se trata de un error no detectable por el compilador de ccZ80++ y se ha producido en la compilación que hace ccz80 o en el ensamblado posterior, la información del error se hace en referencia al fichero ccz80 o al fichero ensamblador que se van generando durante la compilación del programa. En esos casos conviene compilar con la opción /out=ccz80 o /out=assembler para generar esos ficheros y estudiar la causa del error sobre ellos.

# Tipos de datos en ccZ80++

En un programa se pueden tratar los siguientes tipos de dato:

- **short**: valor de 8 bits sin signo, con un rango de 0 a 255.
- **int**: valor de 16 bits sin signo, con un rango de 0 a 65535.
- **objeto**: instancia de una clase definida en el programa. En él se pueden encontrar otros datos short, int y objetos de otras clases.
- **cadena de caracteres**: conjunto de valores short con el código ASCII de los diferentes caracteres que componen la cadena, finalizado con un valor más con el código 0.

Los valores short e int pueden usarse de forma equivalente, pero teniendo en cuenta que si se asigna o pasa como parámetro un elemento de tipo int para un elemento tipo short sólo se guardará el byte bajo del valor int.

Cuando se produce un sobrepasamiento en una operación para un valor short o int el valor resultante queda truncado a los últimos 8 ó 16 bits respectivamente del resultado.

## Tablas

Se pueden definir tablas de una o varias dimensiones de los tipos de datos short, int y objeto. Se puede hacer de dos formas:

- Definiendo las dimensiones, indicando tras el nombre de la tabla entre corchetes el número de elementos de cada dimensión. Ejemplo:

```
short tabla[3][4]; // Tabla con 12 elementos short
```

- Para tablas de una dimensión, declarando los valores de los elementos, sin necesidad de especificar el número de elementos. Esta forma sólo es válida para tablas estáticas. Ejemplos:

```
static short textoEjemplo[] = "Ejemplo"; // Tabla con 8 elementos short, incluyendo el
       carácter indicador de fin de cadena
static int valores[] = { 3, 8, 25, 2, -5, 3 }; // Tabla con 6 elementos int
```

Para acceder a los elementos de una tabla y obtener o asignar su valor se puede hacer de dos formas:

- Indicando el nombre de la tabla y los índices que identifican al elemento que se quiere acceder. Ejemplo:

```
short pantalla[32][25];
short i = 3;
n = pantalla[i][6]; // Asigna a n el valor del elemento de índices 3, 6
```

- Declarando una variable tipo int para usarla como puntero y utilizándolo con los operadores de referencia a memoria. Cuando el puntero debe señalar al siguiente o anterior elemento hay que incrementar o decrementar respectivamente la variable usada como puntero el número de bytes que ocupa un elemento de la tabla, que se puede obtener con el operador @ si no se conoce. Ejemplo:

```
int tabla[10];
int puntero = &tabla;
n = **puntero; // Asigna a n el primer elemento de tabla
n = **( puntero + (5 * 2)); // Asigna a n el elemento de índice 5 de tabla, sin cambiar el
       valor del puntero
puntero += @tabla[0]; // Puntero señala el segundo elemento de tabla; ya que el tamaño del
       elemento es conocido, mejor hacer puntero += 2
```

Si se quiere dar a una función como parámetro una tabla la función debe declarar el argumento correspondiente de forma idéntica a la declaración de la tabla que se da como parámetro. Si se quiere pasar la dirección de inicio de la tabla y utilizar en la función ese parámetro como puntero a la tabla debe pasarse su dirección con el operador &. Ejemplos:

```
void funcion(short tabla[5]) { … }
void funcion 2(int punteroTabla) { … }
…
short tabla[5];
funcion(tabla); // Envío de parámetro como tabla
funcion2(&tabla); // Envío de parámetro como puntero a tabla
```

Las tablas son pasadas como parámetro por referencia, por lo que cualquier cambio en la tabla dentro de la función que la recibe se hará sobre la tabla original.

## Cadenas de caracteres

El tipo cadena de caracteres se implementa en ccZ80++ como la lista de todos los caracteres que la forman más un carácter al final con código 0, indicador de fin de cadena.

Una cadena de caracteres se especifica en el programa como los caracteres que la forman entre comillas dobles, sin carácter indicador de fin de cadena, que se añade automáticamente. Cuando se usa una cadena de caracteres en una expresión se está utilizando el valor numérico de su dirección en memoria donde se encuentra almacenada.

Hay que tener en cuenta que si se especifica una misma cadena de caracteres en varios lugares de un programa sólo se almacena una única vez en memoria, por lo que si se realiza un cambio sobre algún carácter se está cambiando la cadena para todas apariciones que tenga en el programa. Se debe hacer una copia de la cadena de caracteres en otra posición de memoria para evitar esto.

Para almacenar una cadena de caracteres la forma más adecuada es declarar una tabla unidimensional de valores short con un número de elementos igual o mayor que el número de caracteres que pueda tener como máximo la cadena, más uno para el carácter de final de cadena.

Si se quiere implementar una tabla de cadenas de caracteres se podía hacer declarando una tabla bidimensional de valores short, con una primera dimensión igual al número de cadenas a almacenar y una segunda dimensión igual al máximo de caracteres que pueda tener la cadena más extensa de las que se van a almacenar, más uno para el carácter de final de cadena.

También se puede declarar la tabla de elementos short e inicializarla con los códigos de los caracteres de la cadena de caracteres que se quiere almacenar en ella.

Ejemplos:

```
short nombre[11]; // Para cadenas de hasta 10 caracteres de longitud
short nombres[5][21]; // Para 5 cadenas de hasta 20 caracteres de longitud
short nombre = "Ana"; // Tabla con 4 elementos short
short nombre = { 'A', 'n', 'a', '\0' }; // Misma definición que en ejemplo previo
```

Para gestionar las cadenas de caracteres tampoco existen sentencias de forma nativa en ccZ80++. Es necesario hacer uso de funciones para realizar la gestión de este tipo de datos. En la sección de descargas se puede obtener la clase String que contiene un conjunto de funciones para la utilización de cadenas de caracteres.

Para obtener la dirección de memoria de una cadena de caracteres almacenada en una tabla de elementos short se puede hacer con el operador & aplicado a la tabla de valores short que la almacena. Ejemplo: &nombre para una declaración short nombre[11].

Si se trata de una lista de cadena de caracteres se tendría que obtener la dirección del primer carácter de la cadena de la que se quiera saber su dirección en memoria. Ejemplo: &nombres[4][0] para una declaración short nombres[5][21], obteniendo así la dirección de la última cadena de la lista.

En una expresión se pueden tratar las constantes cadenas de caracteres, entre comillas dobles, con los siguientes operadores: ==, !=, >, >=, <, <= y +. Las comparaciones se realizan según el orden alfabético de las cadenas.

# Expresiones

Existen dos tipos de expresiones:

- **Expresiones constantes**, que deben producir un valor constante, numérico o cadena de caracteres. Sólo pueden usar valores constantes y un subconjunto de operadores de los admitidos por el lenguaje. Una expresión constante puede estar formado por un único valor constante, numérico o cadena de caracteres.

  Estas expresiones se pueden utilizar en puntos del programa donde es necesario un valor constante.

  Los operadores válidos para una expresión constante son:

  ( ) Paréntesis para modificar la prioridad de las operaciones en la expresión
  & And a nivel de bit
  && And lógico para condiciones
  * Producto
  ^ Xor a nivel de bit
  : Separador parte falsa de operador ternario
  . Especificador de miembro de clase (sólo para miembros públicos y estáticos)
  == Comparación de igualdad para número y cadena de caracteres
  ! Not lógico para condiciones
  != Comparación de no igualdad para número y cadena de caracteres
  > Comparación de valor mayor para número y cadena de caracteres
  >= Comparación de valor mayor o igual para número y cadena de caracteres

>> Desplazamiento a la derecha a nivel de bit
< Comparación de valor menor para número y cadena de caracteres
<= Comparación de valor menor o igual para número y cadena de caracteres
<< Desplazamiento a la izquierda a nivel de bit
- Resta y especificación de valor negativo
% Módulo
+ Suma, especificación de valor positivo y concatenación de cadena de caracteres
? Separador parte verdadera de operador ternario
/ División
~ Not a nivel de bit
| Or a nivel de bit
|| Or lógico para condiciones

La prioridad de los operadores, de mayor a menor, es la siguiente:

1. Paréntesis: ( ).
2. Especificador de miembro de clase: .
3. Operadores unarios: -, !, ~, +
4. Producto, división y módulo: *, /, %
5. Suma, concatenación y resta: +, -
6. Desplazamiento: <<, >>
7. Comparación: <, <=, >, >=
8. Comparación de igualdad: ==, !=
9. And a nivel bit: &
10. Xor a nivel bit: ^
11. Or a nivel bit: |
12. And lógico para condiciones: &&
13. Or lógico para condiciones: ||
14. Operador ternario: ?, :

Ejemplos:

```
(3 + 5) / 4 // Expresión que produce valor 2
"Hola " + nombre + "." // nombre es una constante definida con una valor de cadena de
       caracteres
25 * Constantes.ancho // ancho es una constante numérica pública en la clase Constantes
10 // Expresión que produce el valor 10
```

- **Expresiones generales**, que deben producir un valor numérico. Pueden usar valores constantes, argumentos, variables y resultados devueltos por funciones. Una expresión pude estar formada por un único operando constante, argumento, variable o llamada a una función.

  Estas expresiones se pueden usar dentro del código de las funciones, junto con las sentencias, donde no sea necesario estrictamente un valor constante.

  Los operadores válidos son:

  ( ) Paréntesis para modificar la prioridad de las operaciones en la expresión
  & And a nivel de bit y dirección de un elemento
  && And lógico para condiciones
  &= Auto and a nivel de bit
  * Producto y referencia a valores de memoria tipo short
  ** Referencia a valores de memoria tipo int
  *= Auto producto
  @ Tamaño
  ^ Xor a nivel de bit
  ^= Auto xor a nivel bits
  : Separador parte falsa de operador ternario
  . Especificador de miembro de clase u objeto

| | |
|---|---|
| = | Asignación |
| == | Comparación de igualdad para número y cadena de caracteres |
| ! | Not lógico para condiciones |
| != | Comparación de no igualdad para número y cadena de caracteres |
| > | Comparación de valor mayor para número y cadena de caracteres |
| >= | Comparación de valor mayor o igual para número y cadena de caracteres |
| >> | Desplazamiento a la derecha a nivel de bit |
| >>= | Auto desplazamiento a la derecha a nivel de bit |
| < | Comparación de valor menor para número y cadena de caracteres |
| <= | Comparación de valor menor o igual para número y cadena de caracteres |
| << | Desplazamiento a la izquierda a nivel de bit |
| <<= | Auto desplazamiento a la izquierda a nivel de bit |
| - | Resta y especificación de valor negativo |
| -= | Auto resta |
| -- | Auto decremento |
| % | Módulo |
| %= | Auto módulo |
| + | Suma, especificación de valor positivo y concatenación de cadena de caracteres |
| += | Auto suma |
| ++ | Auto incremento |
| ? | Separador parte verdadera de operador ternario |
| / | División |
| /= | Auto división |
| ~ | Not a nivel de bit |
| \| | Or a nivel de bit |
| \|= | Auto or a nivel de bit |
| \|\| | Or lógico para condiciones |

La prioridad de los operadores, de mayor a menor, es la siguiente:

1. Paréntesis: ( )
2. Especificador de miembro de clase u objeto: .
3. Autoincremento, autodecremento: ++, --
4. Operadores unarios: -, !, &, *, **, @, ~, +
5. Producto, división y módulo: *, /, %
6. Suma, concatenación y resta: +, -
7. Desplazamiento: <<, >>
8. Comparación: <, <=, >, >=
9. Comparación de igualdad: ==, !=
10. And a nivel bit: &
11. Xor a nivel bit: ^
12. Or a nivel bit: |
13. And lógico para condiciones: &&
14. Or lógico para condiciones: ||
15. Operador ternario: ?, :
16. Autoasignaciones: %=, &=, *=, /=, ^=, |=, +=, <<=, =, -=, >>=

Ejemplos:

```
a = b / 3;
funcion(a + 3, funcion2(3, 5));
n = (v++) + objetoNave.coordenadaX();
```

## Operadores

La mayoría de los operadores tienen la misma función, sintaxis y prioridad que en otros lenguajes como el C++. Existen algunos operadores nuevos o con diferencias con respecto a otros lenguajes:

- Referencia a valores de memoria tipo short: *. Este operador en su forma unaria permite acceder a un valor tipo short en memoria que se indica con su operando. Con ello se puede obtener y asignar un valor de tipo short para una posición de memoria.

```
*dirección
```

Expresiones de ejemplo:

```
*1000 = 10; // Almacena el valor 10 en la dirección de memoria 1000
*n = 10; // Almacena el valor 10 en la dirección de memoria n
v = *n; // Asigna a v el valor de memoria de la dirección n
```

- Referencia a valores de memoria tipo int: **. Este operador de tipo unario permite acceder a un valor de tipo int en memoria que se indica con su operando. El valor se almacena en la dirección indicada por el operando, el byte bajo del valor, y en la siguiente, el byte alto del valor. Con ello se puede obtener y asignar un valor de tipo int para una posición de memoria.

```
**dirección
```

Expresiones de ejemplo:

```
**1000 = #ABCD; // Almacena el valor hexadecimal #ABCD en memoria en las direcciones 1000
       (byte bajo) y 1001 (byte alto)
**n = 2014; // Almacena el valor 2014 en memoria en las direcciones n (byte bajo) y n + 1
       (byte alto)
v = **n; // Asigna a v el valor en memoria de las direcciones n (byte bajo) y n + 1 (byte
       alto)
```

- Tamaño: @. Este operador de tipo unario permite obtener el tamaño en bytes en memoria del parámetro o variable dado como operando. Puede ser útil para incrementar variables int que actúen como punteros para pasar a señalar al siguiente o anterior elemento de una tabla.

```
@elemento
```

Expresiones de ejemplo:

```
p += @tabla[i]; // Incrementa p el número de bytes del elemento de índice i de tabla
pantalla.valor(@tabla); // Llama a la función miembro valor del objeto pantalla pasando
       como parámetro el tamaño en bytes de tabla
```

- Dirección de un elemento: &. Este operador unario permite obtener la dirección de memoria de un argumento o variable de clase o de función, de un elemento de una tabla y también de una función.

  En el caso de una función hay que simular su llamada, que realmente no se realiza, con parámetros ficticios con cualquier valor, pero del tipo que necesitaría recibir la función, para que el compilador conozca exactamente de qué función se quiere obtener su dirección por si se encuentra sobrecargada. Ejemplos:

```
n = &proceso(0, 0); // Asigna a n la dirección de la función proceso que está declarada
       con dos argumentos numéricos, short o int indiferentemente
n = &proceso(); // Asigna a n la dirección de la función proceso que está declarada sin
       argumentos
```

- Especificador de miembro de clase u objeto: . (operador punto). Con este operador de tipo binario se puede acceder a miembros públicos de una clase u objeto. En el código de una función se puede acceder directamente a las constantes, variables y funciones miembro de su clase contenedora, pero si se quiere acceder a miembros de otra clase se debe utilizar el operador punto, con las siguientes normas:

  - No se puede acceder a un miembro no público.

- Si el miembro es una constante o es estático se puede acceder a través del nombre de la clase o a través de un objeto instanciado de la clase.
- Si el miembro es una constante y no es estático se debe acceder a través de un objeto instanciado de la clase.

```
nombreClaseONombreObjeto.miembro
```

Expresiones de ejemplo:

```
Juego.maxPuntos; // Juego es una clase y maxPuntos es un miembro público y estático
juego.vidas; // juego es un objeto y vidas un miembro público
juego.inicio(5); // juego es un objeto e inicio una función miembro pública
```

- Operador ternario: ? :. Evalúa la expresión correspondiente al primer operando, y si resulta distinto de 0 se considera cierta y el operador devuelve el segundo operando, o si resulta 0 se considera falsa y el operador devuelve el tercer operando.

  En ccZ80++ se evalúan siempre los tres operandos antes de evaluar si el primer operando resulta cierto o falso, por lo que las expresiones del segundo y tercer operando se evalúan, modificando elementos si contienen operadores de asignación o modificación, y llamando a funciones si contienen llamada a alguna.

Cualquier operador, además de realizar la operación correspondiente, produce un resultado. Este resultado puede tenerse en cuenta para utilizarlo en la expresión y construir expresiones con múltiples operaciones. Por ejemplo el operador + para valores numéricos realiza la suma y produce un resultado de tipo numérico, que se puede asignar a una variable, quedando la expresión como resultado = valor1 + valor2. Otro ejemplo menos trivial puede ser if ((n = v) == 5) … en el que se asigna a n el valor de v y se utiliza el resultado producido, que es el del primer operando n, comparándolo con 5.

Para conocer qué resultado produce cada operador se puede tener en cuenta lo siguiente:

- Los operadores que realizan una operación aritmética, a nivel de bit o de desplazamiento producen el resultado de esa operación.
- Los operadores que realizan una operación lógica para condiciones o una operación de comparación producen un valor 0 si la operación resulta falsa o 1 si la operación resulta verdadera.
- Los operadores de asignación producen como resultado el operando que recibe la asignación.
- Los operadores de auto incremento y auto decremento producen el valor del operando al que afectan, antes o después del incremento o decremento, dependiendo de si el operador se indica antes o después del operando.
- Los operadores de referencia a valores de memoria producen el valor de la posición de memoria dada por el operando.
- El operador de tamaño produce el valor correspondiente al tamaño en memoria del operando.
- El operador ternario produce el resultado de la expresión de la parte verdadera o de la expresión de la parte falsa.
- El operador especificador miembro de clase u objeto produce el valor correspondiente al miembro indicado como segundo operando.

En ccZ80++ todos los operadores son asociativos por la izquierda. Esto tiene importancia para los operadores de asignación porque se evalúa primero la asignación más a la izquierda. Por ejemplo la expresión n1 = n2 = n3 produce error de compilación porque realiza primero n1 = n2 y a ese resultado le asigna n3; para permitir esta expresión había que escribirla así: n1 = (n2 = n3).

## Valores constantes para expresiones

Cuando en una expresión se necesita indicar un valor constantes, se admiten los siguientes tipos:

- Números decimales: expresados con los dígitos 0 a 9. Ejemplo: 123.
- Números hexadecimales: expresados con el prefijo # y los dígitos 0 a 9 y las letras A a F, mayúsculas o minúsculas indiferentemente. Ejemplo: #1A.

- Caracteres: representan el valor numérico de su código ASCII, y se expresan entre comillas simples. Se pueden indicar secuencias de escape que representen un carácter no imprimible. Ejemplos: ‘a’, ‘\n’.
- Cadenas de caracteres: representan una lista de caracteres almacenados en memoria, finalizada con un carácter de código ASCII 0. Se expresan entre comillas dobles. Se pueden indicar secuencias de escape dentro de la lista que representen caracteres no imprimibles. Ejemplo: "hola\n\radios".

### Secuencias de escape

Para expresar un carácter especial en un valor constante de carácter o dentro de una cadena de caracteres se pueden usar las siguientes secuencias de escape:

- \0: representa el carácter de código 0.
- \a: representa el carácter de código 7.
- \b: representa el carácter de código 8.
- \e: representa el carácter de código 27.
- \f: representa el carácter de código 12.
- \n: representa el carácter de código 10.
- \r: representa el carácter de código 13.
- \t: representa el carácter de código 9.
- \v: representa el carácter de código 0.
- \xNN: representa el carácter de código NN, donde NN se expresa base hexadecimal.

## Palabras clave

Las palabras clave del lenguaje son:

- **asm**: permite insertar código ensamblador directamente en el programa.
- **break**: hace saltar el proceso a la instrucción siguiente tras el bucle donde se encuentre o finaliza la realización de sentencias y expresiones de una sentencia switch.
- **case**: indica un posible valor para la expresión de una sentencia switch para situar a continuación las sentencias y expresiones correspondientes.
- **class**: permite definir una clase.
- **const**: permite definir una constante como miembro de una clase o local de una función.
- **continue**: hace salta el proceso al final de bucle donde se encuentre.
- **default**: indica el inicio de las sentencias y expresiones a realizar si el valor de la expresión de una sentencia switch no coincide con ningún valor de los casos indicados.
- **do**: permite definir un bucle con la comprobación de finalización al final de cada repetición.
- **else**: permite especificar las sentencias y expresiones a realizar cuando la expresión de una sentencia if resulta falsa.
- **for**: permite definir un bucle con inicialización de valores, comprobación de finalización al principio de cada repetición y actualización de valores al final de cada iteración.
- **if**: permite definir una condición para realizar unas sentencias y expresiones o no dependiendo del resultado de la expresión correspondiente a la condición.
- **include**: permite incluir en el programa un fichero fuente en el punto donde se encuentre esta sentencia durante el proceso de compilación.
- **inline**: permite indicar que el código de una función de tipo ensamblador se inserta en el punto donde se llame en lugar de llamarse y retornar a ese punto de llamada.
- **int**: permite declarar argumentos y variables de clase y de función, simples y de tabla, de elementos tipo int. También declarar que una función devuelve un valor de tipo int.
- **public**: permite indicar que una constante, variable o función miembros de clase pueden ser utilizados desde fuera de la clase.
- **register**: permite indicar que el parámetro de una función tipo ensamblador se envía en un registro en lugar de insertarse en la pila.
- **repeat**: permite definir un bucle que se repite un número fijo de veces.
- **return**: permite salir de una función, devolviendo un valor como resultado de la función si es necesario.

- **short**: permite definir argumentos y variables de clase y de función, simples y de tabla, de elementos tipo int. También definir que una función devuelve un valor de tipo short.
- **static**: permite indicar que una variable o función miembros de clase o una variable local de función son comunes a todas las instancias de objeto que se creen para la clase contenedora.
- **switch**: permite realizar un grupo entre varios de sentencias y expresiones dependiendo del valor que produzca una expresión.
- **void**: permite definir que una función no devuelve ningún valor.
- **while**: permite definir un bucle con la comprobación de finalización al principio de cada repetición y también finalizar un bucle con la comprobación de finalización al final de cada repetición.

No se puede declara ninguna clase, constante, variable o función con nombre igual a alguna de las palabras clave.

# Identificadores

Los identificadores permiten nombrar clases, constantes, variables, funciones y argumentos.

Los identificadores del programa deben comenzar por una letra y contener letras mayúsculas, letras minúsculas, dígitos 0 a 9 y el carácter de subrayado _. Las letras válidas son sólo las 26 del alfabeto inglés y no se admiten letras acentuadas.

Pueden tener una longitud indefinida y son sensibles al uso de mayúsculas y minúsculas.

Un identificador no se puede repetir con otros de su mismo nivel ni con los de nivel superior. Esta norma se puede desglosar del siguiente modo:

- Un identificador de una clase no puede repetirse con el de otra clase, ya que están al mismo nivel
- Un identificador de constante miembro de clase no puede repetirse con el de otro miembro de la misma clase como otra constante de clase, variable de clase o función, porque son del mismo nivel; ni con el identificador de una clase, ya que es de un nivel superior
- Un identificador de argumento, constante local de función o variable local de función no puede repetirse con el de otro argumento, constante local o variable local de la misma función, porque son del mismo nivel; ni con identificadores de miembros de su clase contenedora ni con identificadores de clases, porque todos ellos son de nivel superior.
- Un identificador de constante o variable definido dentro de un bloque de una función, que puede ser un bloque de sentencia o un bloque anidado, no puede repetirse con ningún identificador dentro del mismo bloque ni con ningún otro de bloques que contengan al presente, hasta el bloque principal de la función contenedora, ni con identificadores de la clase contenedora de la función, ni con identificadores de clase, pues todos ellos son de nivel superior en la jerarquía del programa.
- Un identificador puede repetirse si se encuentra definido en un bloque que no se encuentra en línea ascendente en la jerarquía del programa, como por ejemplo en una función diferente, o en un bloque de una sentencia diferente a la actual que no se encuentre anidada.
- Como excepción a la norma, un identificador puede repetirse con otro de un bloque de nivel superior del presente si se define una vez cerrado el bloque presente, y no antes de comenzar el bloque presente.

# Uso de objetos

Para crear un objeto en el programa hay que definir primero la clase a la que pertenecerá el objeto mediante la sentencia class.

Un objeto se puede crear como miembro de clase o como miembro local de una función. En ambos casos, se declara como una variable de tipo short o int pero indicando como tipo el nombre de la clase. También se pueden declarar tablas de objetos del mismo modo que tablas de elementos short o int. Ejemplos:

```
Juego juego; // Declara un objeto juego de clase Juego
Real numeros[10]; // Declara una tabla numeros con 10 objetos de la clase Real
```

Se puede acceder a los miembros públicos del objeto con el operador punto, no a los miembros privados. Si un miembro es estático se puede acceder también a través del nombre de la clase además de a través del nombre del objeto.

Desde el código de las funciones miembro de la clase correspondiente al objeto se puede acceder a cualquier miembro público o privado. Si una función declara un objeto de su misma clase contenedora también puede acceder a cualquier miembro público o privado de ese objeto.

Una función no puede devolver un objeto como resultado, pero puede recibir como parámetro un objeto o una tabla de objetos, y para ello debe declarar el argumento correspondiente de forma idéntica a la declaración del objeto o tabla de objetos. Ejemplo:

```
void funcion(Puntuacion argumento) …
…
Puntuacion s;
funcion(s);
```

Los objetos y tablas de objetos son pasados como parámetro por referencia, por lo que cualquier cambio en el objeto o algún elemento de la tabla dentro de la función que lo recibe, se hará sobre el objeto original.

## Constructores

En ccZ80++ no existen los constructores y destructores tal como se conocen en otros lenguajes. Para solventar esta carencia existe posibilidad de la declaración en la clase de la siguiente función:

```
public void init()
{
  …
}
```

Esta función no es invocable explícitamente desde el código.

Si se declara un objeto local no estático en una función, no es válido para objetos declarados como miembros de clase, la función init se llamará automáticamente en el punto de la declaración y tantas veces como el flujo del programa pase por la declaración.

Así, en esta función se pueden asignar y realizar las operaciones necesarias para dar valores consistentes a las variables miembros de la clase o realizar otras acciones de inicialización del objeto.

Si la clase es derivada de otra se crea una cadena de llamadas a las funciones init para las clases en las que se haya definido, desde la clase base más profunda hasta la clase base del objeto que se declara.

Si la declaración es de una tabla de objetos en lugar de un único objeto, se realiza la llamada de la misma forma para cada uno de los objetos de la tabla.

Si el objeto o tabla declarados son estáticos no se realiza la llamada a la función init. Un objeto estático sólo puede acceder a las variables de clase declaradas como estáticas, por lo que la función init no es necesaria, ya que los valores de las variables estáticas pueden ser inicializados en su declaración.

## Objeto this

Todas las funciones declaradas como no estáticas reciben de forma implícita un parámetro this que corresponde al objeto contenedor de la función cuando es llamada. Mediante este parámetro se puede, por ejemplo, pasar el objeto

contenedor de la función actual a otra función; también opcionalmente acceder a los miembros de clase del objeto, aunque no es necesario usar this ya que únicamente con el identificador de los miembros se puede referenciar dichos miembros.

# Estructura de un programa ccZ80++

Un programa ccZ80++ se compone de uno o varios ficheros. En cada fichero se pueden definir una o varias clases, de forma no anidada. Dentro de cada clase se declaran sus constantes, variables y funciones miembro. En cada función se definen sus argumentos, variables locales y el código que especifica qué operaciones realiza la función.

En cualquier punto de un fichero, clase o función se pueden indicar comentarios, precedidos por doble barra // y hasta el final de línea, para documentar las definiciones o el proceso.

## Estructura de un fichero

```
include fichero

class nombre …
{
  …
}
```

La sentencia include puede repetirse las veces que sea necesario, una por cada fichero a incluir en el programa. La sentencia class pude repetirse las veces que sea necesario, una por cada clase que se quiera definir. Estas sentencias pueden ir en el orden que se necesite.

## Estructura de una clase

```
class nombreClase [ : claseBase]
{
  definiciones de constantes miembro
  definiciones de variables miembro
  definiciones de funciones miembro
}
```

Una clase puede heredar opcionalmente otra claseBase que debe estar definida previamente. El efecto es que todos los miembros de la clase base pasan a formar parte de la clase derivada, con el mismo tipo de acceso privado o público que en la clase base. Equivale a copiar el contenido de la clase base al principio del contenido de la clase derivada.

Con la herencia, además de una mejor estructuración del programa y de un ahorro la escritura del código, se consigue un menor tamaño del código máquina final, ya que se genera una sola instancia del código de las funciones miembro de la clase base para todas las clases derivadas que la hereden.

En el interior de una clase las constantes, variables y funciones pueden declararse en el orden que se necesite.

### Declaración de constantes en una clase

Una constante permite definir un identificador que tendrá asociado un valor numérico o de cadena de caracteres y que se puede utilizar posteriormente en cualquier expresión tantas veces como se quiera del mismo modo que se utilizaría directamente el valor numérico o de cadena de caracteres y con la misma forma de uso dentro de una expresión.

```
[public] const nombre = expresiónConstante, nombre2 = expresiónConstante2, …;
```

Se pueden declarar tantas constantes como se quiera con una sentencia const. Cada expresiónConstante es un valor numérico o de cadena de caracteres o una expresión constante que produzca uno de estos tipos de resultado.

El modificador opcional public hace que la constante miembro pueda ser referenciada desde objetos que no pertenecen a la clase, a través de un objeto del tipo de la clase que se está definiendo y también a través del nombre de la clase si también tiene el modificador static. Si no se indica public la constante se considera privada.

Los nombres de las constantes no deben coincidir con el de ninguna otra constante, variable o función dentro de la clase ni con el nombre de ninguna clase definida en el programa.

Ejemplos:

```
const maximo = 100, titulo = "Puntos";
public const valorInicial = 0;
```

## Declaración de variables en una clase

Una variable puede ser de tipo numérico o de tipo objeto de una clase definida en el programa. Como miembro de una clase no se pueden declarar objetos de la propia clase o de una clase que no se haya definido todavía en el orden de compilación del programa. Esto está motivado porque en el momento de esa declaración no se conoce aún el tamaño del objeto declarado de la propia clase o de una clase aún no definida.

```
[public] [static] tipo nombre, nombre2, …;
```

Se pueden declarar tantas variables como se quiera del mismo tipo con la misma sentencia. El tipo puede ser:

- **short**: para declarar una variable de tipo short.
- **int**: para declarar una variable de tipo int.
- **clase**: para declarar una variable objeto de la clase indicada.

Si la variable es de tipo short o int y se indica el modificador static se puede especificar tras el nombre de la variable el signo igual y una expresión constante cuyo valor se asignará a la variable como valor inicial. Ejemplo:

```
static int varInt = 3 * 15;
```

Cada variable puede ser de tipo simple si no se indica nada, o de tipo tabla si se indican cada una de sus dimensiones a continuación como una expresión constante entre corchetes. Ejemplo:

```
short tabla[5][10 * 2]; // Tabla con 100 elementos short, 5 x 20
```

Si se indica el modificador static puede declararse una variable de tipo tabla de una única dimensión de elementos short o int indicando únicamente los corchetes, sin especificar el número de elementos, y a continuación el signo igual y entre llaves un conjunto de expresiones constantes separadas por comas. Esta tabla queda inicializada con los valores dados. Ejemplo:

```
static int tabla[] = { 1, 17 * 3, 45 }; // Tabla con 3 elementos int
```

También con el modificador static puede declararse una tabla de tipo short de una única dimensión indicando únicamente los corchetes, sin especificar el número de elementos, y a continuación el signo igual y una cadena de caracteres. Con ello se declara una tabla de tipo short con un número de elementos igual al número de caracteres de la cadena de caracteres especificada más uno, para el carácter de código 0 que la finaliza que se añade automáticamente. Esta tabla queda inicializada con los códigos ASCII de los caracteres de la cadena de caracteres más un valor 0.

```
static short texto[] = "hola"; // Tabla con 6 elementos short
```

El modificador public hace que la variable miembro pueda ser referenciada desde objetos que no pertenecen a la clase, a través de un objeto del tipo de la clase que se está definiendo y también a través del nombre de la clase si también tiene el modificador static. Si no se indica public la variable se considera privada.

El modificador static hace que la variable sea común en todos los objetos del tipo de la clase que se está definiendo. Además permite que la variable sea accesible desde el exterior de la clase mediante el nombre de la clase si también tiene el modificador public.

Los nombres de las variables no deben coincidir con el de ninguna otra constante, variable o función dentro de la clase ni con el nombre de ninguna clase definida en el programa.

Otros ejemplos:

```
short n1, n2, tablaN[10]; // Variables tipo short n1, n2 y tabla tableN con 10 elementos
       short
public int tabla2[4][3][2]; // Tabla tabla2 con 24 elementos int
Real reales[5], aux; // Tabla reales con 5 elementos objetos de clase Real y variable
       objeto aux de clase Real
```

## Definición de funciones

Dentro de una función es donde se definen las operaciones que realiza un programa.

```
[public] [static] tipoResultado nombre(tipoArgumento nombreArgumento, tipoArgumento2
       nombreArgumento 2, …)
{
  sentencias y bloques de sentencia
  expresiones
  bloques anidados
}
```

El valor de retorno tipoResultado puede ser short o int.

Los tipos de parámetros tipoArgumento pueden ser short, int, objeto, tabla de short, tabla de int y tabla de objetos. El nombre de los argumentos nombreArgumento es un identificador para acceder al argumento dentro de la función.

Los nombres de los argumentos de la función deben ser únicos en la lista de argumentos de la función que se está definiendo y no coincidir con el nombre de ninguna constante, variable o función miembro de la clase contenedora, ni con el de ninguna constante ni variable local de la función, ni con el de ninguna clase del programa.

Se puede sobrecargar una función si se declara con un conjunto de argumentos diferente en número, tipo u orden. Se considera que es equivalente un argumento de tipo short o int. Se considera diferente un argumento de tipo tabla si tiene dimensiones diferentes. Ejemplo:

```
void funcion(short a, int b) { … }
int funcion(int x, int y) { … } // Error, esta función es equivalente a la anterior
void funcion(short a, int b, int c) { … } // Esta función no es equivalente
```

Cuando se realiza la llamada a una función hay que pasar como parámetros valores que coincidan con los argumentos de su declaración con la siguiente correspondencia:

- Argumento **short** o **int**: puede darse como parámetro una variable short, una variable int, un valor constante numérico en base decimal o hexadecimal, una cadena de caracteres o una expresión que produzca un valor numérico. En el caso de argumentos short, si se pasa un valor que sobrepase la capacidad del tipo short se pierde precisión tomándose sólo el byte bajo del valor dado.
- Argumento **objeto**: debe pasarse un objeto del mismo tipo.

- Argumento **tabla de short**: debe pasarse una tabla de elementos short con las mismas dimensiones que en la declaración del argumento.
- Argumento **tabla de int**: debe pasarse una tabla de elementos int con las mismas dimensiones que en la declaración del argumento.
- Argumento **tabla de objetos**: debe pasarse una tabla de objetos del mismo tipo y con las mismas dimensiones que en la declaración del argumento.

Un argumento short o int es recibido como copia del valor original, mientras que un argumento objeto o de tipo tabla de cualquier tipo de elemento es recibido por referencia al elemento original especificado como parámetro.

El modificador public hace que la función miembro pueda ser referenciada desde objetos que no pertenecen a la clase, a través de un objeto del tipo de la clase que se está definiendo y también a través del nombre de la clase si también tiene el modificador static. Si no se indica public la función se considera privada.

El modificador static hace que la función sea accesible desde el exterior de la clase mediante el nombre de la clase si también tiene el modificador public. Una función estática no puede acceder a miembros de la clase a la que pertenece, y no se genera para ella el parámetro this, sólo puede usar los parámetros que reciba y sus constantes y variables locales. Si una función no va a utilizar miembros de su clase contenedora se puede declarar como estática para mejorar el rendimiento del programa.

Dentro una función pueden existir los siguientes tipos de bloques:

- Bloque principal de la función, delimitado por las llaves de apertura y cierre de la función.
- Bloques correspondiente a alguna sentencia, delimitado por las llaves de apertura y cierre del bloque tras una sentencia if, repeat, do, while o for.
- Bloques anidados en cualquier punto de la función.

Ejemplos de bloques:

```
void funcion()
{ // Inicio de bloque principal de la función
  …
  if (a == 0)
  { // Inicio del bloque de la sentencia if
    …
    { // Inicio de bloque anidado dentro del bloque de la sentencia if
      …
    } // Final de bloque anidado dentro del bloque de la sentencia if
    …
  } // Final del bloque de la sentencia if
  …
  { // Inicio de bloque anidado dentro del bloque principal
    …
  } // Final de bloque anidado dentro del bloque principal
} // Final del bloque principal de la función
```

Dentro de cada bloque se pueden especificar sentencias y declarar constantes y variables, cuya visibilidad será local dentro del bloque, pero cuyo nombre no debe coincidir con el de ninguna constante, variable, argumento de la función, ni con el de ninguna función de la clase, ni con el de ninguna clase del programa. La visibilidad de estas constantes y variables será sólo dentro del bloque donde se declaren y en otros bloques que se encuentren en su interior.

**Definición de constantes y variables en una función**

Se aplican las mismas reglas que para la definición de constantes y variables en una clase, con las siguientes diferencias:

- Las variables de tipo short o int simple, no tablas, se pueden inicializar con una expresión si tras su nombre se indica el signo igual y la expresión. Esta expresión se evaluará en el punto de la declaración de la variable y tantas veces como el flujo del programa pase por este punto. Ejemplo:

```
short a = 0;
int b = 16 + funcion(a) >> 2;
```

- Los nombres de constantes y variables locales de una función no deben repetirse con el nombre de ningún argumento de la función, ni con el de ninguna otra constante o variable de la misma función, además de no repetirse con ninguna constante, variable o función definida en la clase contenedora, ni con el de ninguna clase del programa.
- Excepción al punto previo es que se puede declarar otra constante o variable con el mismo nombre de una constante o variable si una de ellas se ha definido dentro de un bloque de la función y la otra se define en otro bloque de la función o en el bloque contenedor pero posteriormente al cierre del primer bloque. Ejemplo:

```
...
short a = 0;
if (a != 0)
{
  …
  short a = 1; // Error: nombre duplicado
  short b;
  …
}
else
{
  …
  short b; // Correcto: nombre duplicado pero en otro bloque
  …
}

short b; // Correcto: nombre duplicado pero tras cerrar el bloque donde está la primera
       declaración
…
```

- El modificador public no es válido para la definición de constantes y variables locales de una función.

- Una variable debe declararse antes de ser utilizada dentro del código de la función, aunque puede ser en cualquier punto de la función, no necesariamente al principio.

- En una función sí se puede declarar una variable de tipo objeto de la clase contenedora propia o de otra clase aún no definida.

**Sentencias**

Además de definiciones de constantes y variables una función puede contener sentencias que realicen las operaciones propias de la función. Las sentencias pueden aparecer en el bloque principal de la función, en un bloque de sentencia o en un bloque anidado.

Las sentencias que admite ccZ80++ son:

- **asm**: permite insertar código ensamblador directamente.

```
asm sentencia ensamblador
```

o

```
asm
{
  sentencias ensamblador
}
```

Para crear un código fuente más estructurado se recomienda encapsular el código ensamblador en funciones tipo ensamblador. Esto no penaliza demasiado la velocidad y espacio del código máquina generado, e incluso si la función se declara de tipo inline si es posible, no existe ninguna diferencia respecto a si el código ensamblador se inserta con la sentencia asm.

Ejemplos:

```
asm call 5000

asm
{
  imprimirCaracter: equ 1000
  ld a,10
  call imprimirCaracter
}
```

- **if**: realiza el código indicado si se cumple la condición indicada por la expresión dada. Si la expresión produce un valor 0 se considera que no se cumple, si produce cualquier otro valor se considera que se cumple.

  La parte else se realiza si la condición no se cumple, y es opcional.

```
if (expresion)
  sentencia, expresion o bloque
[else
  sentencia, expresion o bloque]
```

  Ejemplo:

```
if (n > 5)
  n = 0;
else
{
  m = n;
  break;
}
```

- **switch**: evalúa un valor y realiza las sentencias y expresiones correspondientes al valor.

```
switch (expression)
{
  case expresiónConstante:
    sentencias y expresiones
    [break;]
  case expresiónConstante2:
    sentencias y expresiones
    [break;]
  …
  [default:]
    sentencias y expresiones
}
```

  Se realizan las sentencias y expresiones que siguen a la palabra clave case con el valor de la expresión constante que coincida con el de la expresión. Si ningún valor de todas las expresiones constantes indicadas con case coincide se realizan las sentencias y expresiones que siguen a default, si se ha especificado, o en caso contrario no se realiza ninguna sentencia o expresión en la sentencia switch.

  Si no se especifica la sentencia break se siguen realizando las sentencias y expresiones del siguiente bloque case.

  Para las expresiones constantes que siguen a case no se permite el operador ternario, pero sí cualquier otro tipo de operador admitido en las expresiones constantes.

En las sentencias y expresiones que siguen a cada palabra clave case o a default no se admiten declaraciones de constantes o variables, a menos que se abra un bloque mediante anidado para incluir en él las declaraciones necesarias y las sentencias y expresiones que corresponden. Sin embargo, no es posible utilizar break dentro de este bloque, sino que hay que cerrarlo y fuera de él indicar la sentencia break.

Ejemplo:

```
switch (n + 1)
{
  case 10:
    n = 2;
  case 20:
    z = 3;
    break;
  case 30:
    return n;
  default:
    --n;
}
```

- **repeat**: repite una o varias sentencias o expresiones el número de veces que se indique.

```
repeat (expresion)
  [sentencia, expresion o bloque
    …
    [continue;]
    …
    [break;]
  …]
```

Las sentencias, expresiones o bloque son opcionales y se pueden omitir indicando sólo un punto y coma en su lugar.

La sentencia continue hace que el control del programa salte todas las sentencias de la iteración actual. La sentencia break sale del bucle y pasa la siguiente instrucción tras el bucle.

Ejemplo:

```
repeat (25 * n) Text.imprimeCaracter('A');
```

- **do**: repite una o varias sentencias o expresiones mientras se cumpla la condición que se evalúa al final. Si la expresión produce un valor 0 se considera que no se cumple, si produce cualquier otro valor se considera que se cumple.

```
do
  [sentencia, expresion o bloque
    …
    [continue;]
    …
    [break;]
  …]
while (expresion);
```

Las sentencias, expresiones o bloque son opcionales y se pueden omitir.

La sentencia continue hace que el control del programa salte todas las sentencias hasta la evaluación de la expresión. La sentencia break sale del bucle y pasa la siguiente instrucción tras el bucle.

Ejemplo:

```
do
{
  ++n;
  if (n % 2) continue;
}
while (n < 10);
```

- **while**: repite una o varias sentencias o expresiones mientras se cumpla la condición que se evalúa al principio. Si la expresión produce un valor 0 se considera que no se cumple, si produce cualquier otro valor se considera que se cumple.

```
while (expresion)
  [sentencia, expresion o bloque
    …
    [continue;]
    …
    [break;]
    …]
```

Las sentencias, expresiones o bloque son opcionales y se pueden omitir indicando sólo un punto y coma en su lugar.

La sentencia continue hace que el control del programa salte todas las sentencias y vuelva al principio a la evaluación de la expresión. La sentencia break sale del bucle y pasa la siguiente instrucción tras el bucle.

Ejemplo:

```
while (n < 100) resultado += n++;
```

- **for**: inicializa valores al iniciar, repite una o varias sentencias o expresiones mientras se cumpla una condición que se evalúa al principio y actualiza valores al final.

```
for ([expresiones inicialización; [expresión condicion]; [expresiones actualización])
  [sentencia, expresion o bloque
  …
  [continue;]
  …
  [break;]
  …]
```

Los tres grupos de expresiones son opcionales y se puede omitir cualquiera de ellas indicando sólo el punto y coma que las separa del resto.

Las sentencias, expresiones o bloque son opcionales y se pueden omitir indicando sólo un punto y coma en su lugar.

Como expresiones de inicialización se puede indicar una o varias expresiones separadas por coma que se realizarán antes de comenzar las repeticiones del bucle.

Si la expresión de condición produce un valor 0 se considera que no se cumple, si produce cualquier otro valor se considera que se cumple.

Como expresiones de actualización se pueden indicar una o varias expresiones separadas por coma que se realizarán al final de las sentencias y expresiones del bucle antes de evaluar de nuevo la expresión de condición y una nueva repetición si se cumple ésta.

La sentencia continue hace que el control del programa salte todas las sentencias y pase al final para la realizar las expresiones de actualización. La sentencia break sale del bucle y pasa la siguiente instrucción tras el bucle.

Ejemplo:

```
for (i = 1, divisor = 128; numero > 0 && i <= 8; ++i, divisor >> 2)
{
  c = numero / divisor;
  numero %= divisor;
}
```

- **continue**: lleva el control al final del bucle actual.

```
continue;
```

- **break**: sale del bucle actual o finaliza la realización de la sentencia switch.

```
break;
```

- **return**: sale de una función y devuelve el valor correspondiente a su resultado si la función no es tipo void.

```
return [expresion];
```

  Si la función donde se encuentra la sentencia return devuelve short o int la expresión es obligatoria. Si la función es de tipo void no debe indicarse ninguna expresión.

  Ejemplos:

```
return; // Para función void
return 0; // Para función short o int
return funtion(3, 2); // Para función short o int
```

Además de estas sentencias, se pueden indicar junto con ellas expresiones que realicen asignaciones y llamadas a funciones. Las expresiones deben finalizar con un punto y coma, tanto si se encuentran de forma independiente como si son parte de la acción de una sentencia. Ejemplos:

```
a = funcion(3, 5) + Pantalla.direccionPosicionActual() * 2;
++n;
Utilidades.Evaluar(a == 5, "correcto", "error");
puntuacion.CambiaValor(juego.obtenerPuntuacion());
```

# Uso de ensamblador

Dentro del código de una función se puede usar la sentencia asm para incluir directamente código ensamblador.

Sin embargo, el código fuente queda más estructurado, y no menos eficiente, si para indicar operaciones directamente en ensamblador se crean funciones de tipo ensamblador que sean llamadas desde el código de una función general no de tipo ensamblador.

Para definir una función de tipo ensamblador se hace dentro de la clase donde se necesite con la siguiente sintaxis:

```
[public] [static] tipoResultado nombre (tipoArgumento nombreArgumento, tipoArgumento2
       nombreArgumento 2, …) asm [register|inline] [(funciónSoporte, funcionSoporte2, …)]
{
  sentencias ensamblador
}
```

El valor de retorno tipoResultado y los tipos de argumento tipoArgumento pueden ser del mismo tipo y con las mismas restricciones que en una función general. El nombre de los argumentos no es significativo, ya que dentro de la función no se referenciarán mediante su nombre, sino mediante su desplazamiento dentro de la pila.

Una función tipo ensamblador se puede sobrecargar con otras también de tipo ensamblador o con otras de tipo general.

Los parámetros que se pasan a una función de tipo ensamblador respecto al argumento indicado en su declaración deben cumplir las mismas normas que para una función general.

Los parámetros en una función tipo ensamblador son recibidos por valor o por referencia del mismo modo que en una función general.

Los modificadores public y static tienen el mismo efecto que sobre una función general.

Se puede especificar uno de los dos modificadores register o inline, con el siguiente efecto:

- **register**: sólo válido para funciones que reciban un parámetro. Hace que el valor del parámetro llegue a la función en el registro A si se trata de un argumento de tipo short o en el registro HL si se trata de un argumento tipo int o un objeto o tabla, que en estos dos últimos casos se recibirá su dirección en memoria.
- **inline**: sólo válido para funciones que reciban uno o ningún parámetro y que no declaren etiquetas. El valor del parámetro, si existe, se recibe igual que en una función con el modificador register. El modificador inline hace que el código de la función se inserte directamente en el punto donde es llamada, en lugar de realizarse una llamada y un retorno al punto de llamada; es decir, es equivalente a insertar su código con la sentencia asm. Una función con este modificador no debe tener una instrucción ret para finalizar y volver al punto desde donde se llamó, simplemente cuando se haya realizado su última instrucción se continuará con el programa.

Tras la palabra clave asm que se debe indicar tras los argumentos se puede especificar una lista de funciones de soporte funciónSoporte, que deben ser de tipo ensamblador.

Estas funciones pueden ser de la propia clase o de otras clases, y en este segundo caso hay que referirse a ellas mediante el nombre de clase y el nombre de función mediante el operador punto, por lo que deben ser públicas y estáticas y estar contenidas en una clase que se haya definido previamente en el orden de compilación del programa.

Para especificar una función de soporte, de la propia clase o de otra, tras su nombre hay que indicar parámetros simulados, sin importar su valor pero sí el tipo, ya que no se realiza realmente la llamada a la función, para que el compilador pueda determinar de qué versión de la función se trata por si está sobrecargada. En el siguiente ejemplo se declara una función tipo ensamblador que hace uso de otras tres como funciones de soporte, la primera de ellas de la propia clase y que recibe un parámetro numérico short o int, la segunda de la clase Utilidades y que recibe dos parámetros numéricos, y la tercera de la clase Principal que no recibe ningún parámetro:

```
short funcion() asm (imprimirCadena(0), Utilidades.suma(0, 0), Principal.proceso())
{
  …
}
```

El código, etiquetas y zonas de memoria de todas las funciones de soporte se puede utilizar desde la presente función de tipo ensamblador sin necesidad de repetirlo. Con esto se puede llamar a código de estas funciones de soporte y utilizar los valores contenidos en las zonas de memoria reservadas por ellas.

El contenido de una función de tipo ensamblador son únicamente instrucciones ensamblador Z80, incluyendo si es necesario etiquetas y directivas.

Si la función de tipo ensamblador no tiene indicado un valor de retorno void, para que devuelva el resultado correspondiente se debe cargar en un registro el valor de retorno y salir de la función, bien con la instrucción ret o bien siendo esta carga la última instrucción de la función si ésta tiene el modificador inline. El registro a cargar antes de finalizar la función es:

- Si la función devuelve un tipo short, el registro A.
- Si la función devuelve un tipo int, el registro HL.

## Acceso a parámetros

Hay que distinguir si el acceso a parámetros con ensamblador se hace desde una función general con instrucciones incluidas en un bloque de la sentencia asm o desde una función tipo ensamblador.

En ambos casos los parámetros de la función se encuentran en la pila, pero con formas diferentes de obtener la dirección donde se encuentran. También en ambos casos según el tipo de parámetro en la pila se tendrá un valor con un significado diferente para acceder a él:

- Parámetros short e int: en la pila se encuentra una copia de su valor.
- Parámetros objeto: en la pila se encuentra la dirección de la zona de memoria donde se encuentran sus datos. Los datos de un objeto son las variables miembros no estáticas definidas en su clase, y se encuentran en el mismo orden en el que han sido declarados en el programa, y en posiciones consecutivas, ocupando el tamaño necesario para cada dato; es decir, 1 byte para una variable short, 2 bytes para una variable int, el tamaño del objeto para una variable objeto, y el producto del número de elementos de la tabla por el tamaño de uno de sus elementos para tablas.
- Parámetros tabla de short, int u objeto: en la pila se encuentra la dirección de inicio del primer elemento; el resto de elementos van en posiciones consecutivas.

Para los objetos su tamaño es la suma del número de bytes de cada una de sus variables miembro no estáticas. Las variables estáticas de un objeto se encuentran en una zona de memoria diferente y para acceder a ellas sería necesario conocer su etiqueta identificativa, y para ello habría que generar el código ensamblador del programa con la opción de compilación /out=assembler y examinar el código para encontrarla, aunque hay que tener en cuenta que el nombre de la etiqueta identificativa puede variar en cada compilación.

### Acceso a parámetros de una función general.

Si se quiere acceder a un parámetro de una función general mediante código ensamblador incluido en una sentencia asm hay que realizar los siguientes pasos para obtener la dirección de memoria donde se encuentra un parámetro:

1. Obtener la dirección de referencia de los datos de la función actual tomándola de la etiqueta _FunctionDataReferenceStore predefinida por el programa.
2. Calcular la dirección del parámetro con la fórmula: (nº parámetros - posición parámetro) x 2 + 4 + dirección referencia.
3. Si el parámetro es de tipo short hay que sumar 1 al resultado obtenido.

Por ejemplo, para una función que reciba 5 parámetros, si queremos obtener el valor del cuarto parámetro de tipo int:

```
ld hl,(_FunctionDataReferenceStore) ; HL tiene dirección de referencia de datos
ld de,6 ; Desplazamiento del cuarto parámetro int = (5 – 4) x 2 + 4
add hl,de ; HL tiene la dirección del cuarto parámetro
ld e,(hl)
inc hl
ld d,(hl) ; DE tiene el valor del cuarto parámetro
```

Para obtener de la misma función el valor del segundo parámetro de tipo short:

```
ld hl,(_FunctionDataReferenceStore) ; HL tiene dirección de referencia de datos
ld de,11 ; Desplazamiento del segundo parámetro short = (5 – 2) x 2 + 4 + 1
add hl,de ; HL tiene la dirección del segundo parámetro
ld a,(hl); A tiene el valor del segundo parámetro
```

Hay que tener en cuenta que si la función no está definida como estática, además de los parámetros correspondientes a los argumentos indicados en su definición, recibe un primer parámetro que es la dirección de this, es decir, del objeto contenedor de la función.

En las funciones generales también se puede acceder a sus variables locales a partir de su dirección de referencia. La dirección en la que se encuentran las variables locales de una función se puede calcular con la fórmula: dirección de

referencia - suma de tamaño de todas las variables no estáticas. A partir de esta dirección se encuentran las variables locales de la función, en el mismo orden que se han declarado. Ejemplo:

```
static void function()
{
  short a;
  int hl;
  short c;

  asm
  {
    ld ix,(_FunctionDataReferenceStore) ; IX tiene dirección de referencia de datos
    ld de,-3 ; Desplazamiento para variables locales de la función
    add ix,de ; IX tiene la dirección de las variables miembro del objeto
    ld a,(ix+0) ; A tiene valor de variable a
    ld l,(ix+1)
    ld h,(ix+2) ; HL tiene valor de variable hl
    ld c,(ix+3) ; C tiene valor de variable c
  }
}
```

Las variables estáticas de una función se encuentran en una zona de memoria diferente y para acceder a ellas sería necesario conocer su etiqueta identificativa, y para ello habría que generar el código ensamblador del programa con la opción de compilación /out=assembler y examinar el código para encontrarla, aunque hay que tener en cuenta que el nombre de la etiqueta identificativa puede variar en cada compilación.

### Acceso a parámetros de una función tipo ensamblador.

El acceso a un parámetro de una función tipo ensamblador se hace a partir del valor del registro SP con la siguiente fórmula: SP + (nº parámetros – posición parámetro) x 2 + 2. Si el parámetro es de tipo short hay que sumar 1 a este resultado. Ejemplo:

```
static void function(short a, int hl) asm
{
  ld ix,0
  add ix,sp ; IX tiene el valor de SP
  ld a,(ix+5); A tiene el valor del parámetro a, desplazamiento (2 – 1) * 2 + 2 + 1
  ld l,(ix+2)
  ld h,(ix+3) ; HL tiene el valor del parámetro hl, desplazamiento (2 – 2) * 2 + 2
}
```

Hay que tener en cuenta que si la función no está definida como estática, además de los parámetros correspondientes a los argumentos indicados en su definición, recibe un primer parámetro que es la dirección de this, es decir, del objeto contenedor de la función.

## Sintaxis ensamblador

La sintaxis para escribir el código ensamblador tanto para las sentencia asm como para las funciones de tipo ensamblador está basada en la del ensamblador GENA del paquete DEVPAC, con las diferencias y características que se indican a continuación:

- Acepta todas las instrucciones e incluye el uso de las instrucciones que utilizan los registros ixh, ixl, iyh, iyl y las instrucciones indocumentadas de rotación y desplazamiento.
- La declaración de etiquetas y símbolos con EQU necesita dos puntos (:) tras el nombre de la etiqueta.
- La longitud del nombre de las etiquetas y símbolos es ilimitada.
- Las constantes numéricas pueden indicarse en decimal, hexadecimal con el prefijo # y en binario con el prefijo %.
- Las constantes de cadena y constantes de carácter se delimitan con comillas dobles.
- No se deben usar secuencias de escape en cadenas o constantes de carácter.

- Las expresiones pueden usar constantes numéricas, constantes de carácter, símbolos definidos con EQU, etiquetas y los operadores + (suma), - (resta), * (producto), / (división entera), ? (módulo), & (y lógico), @ (o lógico) y ! (xor lógico). También se puede usar el símbolo $ para especificar la dirección de la instrucción actual.
- Las directivas permitidas son ORG, EQU, DEFB, DEFW, DEFM y DEFS.
- Las directivas condicionales IF, ELSE y END no se aceptan.
- No se diferencian mayúsculas y minúsculas para instrucciones, nombres de registros, etiquetas y símbolos.
- Los comandos de ensamblador (*E, *H, *S, etc.) no se reconocen.
- Los números de línea no están permitidos.

# Restricciones de ccZ80++

El lenguaje no utiliza de forma nativa gestión de memoria dinámica, por lo que la creación de clases se hace declarando variables de tipo objeto, no creándolas mediante el operador new como por ejemplo en C++. Tampoco permite otras características que requieran el uso de memoria dinámica, como la devolución de un objeto por una función, aunque se puede enviar como parámetro a una función un objeto o tabla de objetos y gestionarlos dentro de ella, modificando su estado.

La decisión de no utilizar objetos en memoria dinámica en ccZ80++ es debida a que penaliza demasiado la velocidad del programa generado, ya que se deben crear y eliminar los objetos de memoria y hacer periódicamente una liberación de las zonas de memoria que ya no utiliza ningún objeto activo en el programa.

Tampoco existe el polimorfismo de clases, es decir, permitir referenciar un objeto como si fuese de una de sus clases bases. En ccZ80++ no existen referencias a objetos, sino variables de tipo objeto, por lo que esta carácterística no es aplicable al lenguaje.

Los programas generados con ccZ80++ no hacen ninguna gestión de control de errores para permitir una velocidad máxima en la ejecución del código.

El lenguaje ccZ80++ no contempla los punteros de forma nativa. Para utilizar un puntero se debe declarar una variable de tipo int y asignarle la dirección del elemento al que se quiera de apunte mediante el operador &, y si se trata de un elemento tipo short, acceder o asignar su valor mediante el operador *, o si es un elemento de tipo int, mediante el operador **. Si el puntero señala elementos de una tabla, para que el puntero señale al siguiente elemento o al anterior es necesario sumar o restar respectivamente el tamaño del elemento, que se puede obtener con el operador @.

No existen las estructuras con ese nombre, pero se puede declarar una clase, con o sin funciones miembros, con sus variables miembro públicas, y los objetos que se declaren con esa clase tendrán exactamente la misma funcionalidad y uso que una estructura de otros lenguajes. Los datos se encontrarán en memoria en la dirección del objeto en el mismo orden que se han declarado en su clase.

# Optimizaciones

El código máquina producido por ccZ80++ haciendo una programación normal es muy rápido, pero si se quiere optimizar más se pueden seguir los siguientes consejos:

- Utilizar argumentos y variables de tipo short en lugar de tipo int si el rango de valores que van a almacenar no necesita el tipo int, es decir, que no sobrepasa el valor 255. El microprocesador Z80 gestiona más rápido valores short que valores int.
- Si una clase contiene variables miembro a los cuales se les asigna valor sin ninguna comprobación previa y se obtiene su valor sin ningún tratamiento del dato, se puede evitar crear funciones de obtención y asignación de valor de dichos miembros, y declararlos como públicos para que se puedan utilizar directamente.
- Si una función no es recursiva, ni directa ni indirectamente, se pueden declarar sus variables locales como estáticas. El acceso a una variable estática es más rápido que a una variable no estática.

- Si una función no es recursiva, ni directa ni indirectamente, si se accede numerosas veces en el código a un argumento de una función, su valor se puede pasar a una variable estática local de la función. El acceso a una variable estática es más rápido que a un argumento.
- Por último, si para alguna acción se necesita una velocidad máxima, se puede hacer para ella uso de ensamblador, preferiblemente encapsulado en funciones de tipo ensamblador para mejor legibilidad del código fuente.