

Vircon32: Guía del compilador de C

Documento con fecha 2022.02.25

Escrito por Carra

¿Qué es esto?

Este documento es una guía rápida sobre cómo usar el lenguaje C para crear programas en la consola Vircon32. El objetivo de esta guía es mostrar las características del lenguaje, y sus diferencias con el estándar y con otros compiladores. Este documento es aplicable al compilador versión 2022.02.25.

Índice

Este documento se organiza en secciones, y cada una cubre una de las características del lenguaje C de Vircon32 y su compilador.

Índice	1
Introducción	2
Comentarios.....	2
Tipos de datos	3
Valores literales	5
Variables	6
Funciones	8
Expresiones.....	9
Control de flujo	10
Incrustar código ensamblador.....	13
El preprocesador	13
La librería estándar	15
Diferencias globales con C estándar.....	16
Limitaciones generales del compilador	17
Usando el compilador	18

Introducción

El compilador de C es parte de las herramientas de desarrollo de Vircon32, y su función es interpretar nuestros programas en lenguaje C para traducirlos a lenguaje ensamblador. Luego, un segundo programa (el ensamblador) lo traducirá a su vez a instrucciones en código máquina que la CPU de Vircon32 es capaz de ejecutar.



El lenguaje C que se usa en este compilador es en su mayoría igual al estándar, pero hay algunas diferencias y limitaciones que hace falta tener en cuenta a la hora de programar. Algunas de estas diferencias vienen dadas por cómo funciona la propia consola, y otras son simplemente resultado de mis propias limitaciones a la hora de programar un compilador. Aún así creo que merece mucho más la pena poder programar a más alto nivel, incluso si es con algunas limitaciones, que poner barreras a la creación de juegos en la consola exigiendo aprender ensamblador.

En las próximas secciones voy a ir describiendo las características del lenguaje, y resaltaré **en color rojo** las diferencias con un compilador de C estándar, para que quien ya sepa programar en C pueda ver de un vistazo rápido las principales cosas a tener en cuenta.

Comentarios

Los comentarios son una forma de incluir en el programa nuestras propias aclaraciones sin que el compilador los intente interpretar, es decir, no forman parte del programa en sí. Existen 2 tipos de comentarios, y funcionan igual que en el lenguaje C estándar:

Comentario de línea

Estos comentarios comienzan con `//`, y acaban al final de la línea. Se pueden alargar hasta las siguientes líneas si la línea se continúa acabando en `\`.

```
// comentario de línea continuado \  
    en la línea siguiente  
  
int x = 7 + 5;    // comentario sin continuar  
int y = x;
```

Comentario de bloque

Un comentario de bloque comienzan con `/*`, y no acaba hasta que lo terminemos con `*/`, aún si se hace en una línea distinta.

```
/* este comentario anula la línea de debajo  
int x = 7 + 5;  
*/  
  
int y;
```

Tipos de datos

Todos los valores o variables que maneja el lenguaje pertenecen a algún tipo de dato que indica cómo interpretarlo. Aquí veremos qué tipos de datos podemos utilizar.

Tipos básicos

El lenguaje C de Vircon32, a diferencia del C estándar, **sólo dispone de datos de 32bits. Además no hay variantes de esos tipos: ni de signo (signed, unsigned), ni de tamaño (short, long).**

Vircon32 organiza la memoria en palabras de 32bits porque en esta consola la unidad mínima de memoria es la palabra (y no el byte), con lo que debemos tener en cuenta que:

- **Todos los tamaños de datos se miden en palabras y no bytes**
- **Todas las direcciones de memoria y offsets se dan en palabras y no bytes**

Los 4 tipos de datos básicos en este compilador son:

<code>int</code>	<code>float</code>	<code>bool</code>	<code>void</code>
------------------	--------------------	-------------------	-------------------

El tipo void es especial, y sólo se puede usar para indicar que una función no devuelve ningún valor. Los demás tienen tamaño 1 (es decir, 32 bits).

Tipos derivados

Podemos crear nuevos tipos usando los tipos básicos en arrays y punteros. Esto puede hacerse múltiples veces, por ejemplo:

<code>int*</code>	<code>void**</code>	<code>float[3][5]</code>	<code>void*[4]</code>
-------------------	---------------------	--------------------------	-----------------------

Podemos usar void para crear punteros, pero no arrays (pues no pueden existir valores de tipo void). Un puntero a void se usa para guardar una dirección de memoria en la que se lee o guarda información sin interpretarla (se usa void para indicar la ausencia de tipo).

Punteros a funciones

En la versión actual del compilador no existen punteros a funciones. Sin embargo está previsto que se añadan en las próximas versiones.

Tipos compuestos

Podemos agrupar varios tipos de datos en un tipo mayor, mediante estructuras y uniones. Cada miembro de estos grupos es un campo al que se accede por su nombre.

Mientras que en C estándar, al declarar una struct/enum se declara una variable y no un tipo (a no ser que se use typedef), el C de Vircon32 las trata como declaración de tipo, más similar a como C++ trata las clases:

```
// declaramos el tipo Point
struct Point
{
    int x, y;
};

// declaramos el tipo Word
union Word
{
    int AsInteger;
    void* AsPointer;
};

// usamos estructuras
Point P1,P2;
P1.x = 10;
P1.y = -7;
P2 = P1;

// usamos uniones
Word W;
W.AsInteger = 0xFF110AF;
int* Pointer = W.AsPointer;
```

Las estructuras y uniones también pueden contener a su vez otras estructuras y uniones. También se pueden incluir a sí mismas, aunque sólo con punteros por razones obvias.

El C estándar permite usar en estructuras campos de bits para agrupar bits individuales. No están soportados aquí, y no está previsto añadirlos.

Enumerados

Podemos definir una serie de constantes enteras y agruparlas en su propio tipo. El lenguaje C permite esto con los enumerados. En general estos valores se tratan como enteros y se pueden usar de la misma forma, pero su tipo es más restrictivo. Al igual que con struct y union, usar enum siempre declara un tipo como si se usara typedef.

Si no se especifican valores, la primera constante tendrá un valor numérico y cada constante siguiente tendrá el valor de la anterior + 1.

```
// declarar tipo semáforo
enum Semaphore
{
    Red = 1,
    Yellow,
    Green,
};

// estas operaciones se pueden hacer
Semaphore S1,S2;
S1 = Red;
S2 = S1;
int Value = Yellow + Green; // es seguro convertir enum a int

// estas asignaciones producirían errores!
S1 = Green - Red;
S2 = 1; // mismo valor que Red, pero no se puede convertir int a enum
```

Definir tipos con typedef

Podemos usar `typedef` para dar un nombre a un tipo de dato. Luego ese nombre se puede utilizar para representar ese tipo de datos:

```
// definimos un tipo
typedef int** int_ptr2;

// y luego lo utilizamos
int_ptr2 P = NULL;
```

Recuerda que **en este compilador la forma de escribir tipos complejos está simplificada respecto al C estándar**. Por ejemplo: en C estándar para definir un puntero a array, hace falta usar paréntesis y poner el nombre en medio del propio tipo:

```
typedef int (*ptr_to_array)[5];
```

En cambio, en este compilador siempre se mantienen separados: <tipo> <nombre>. Los tipos se identifican más fácilmente, leyendo de derecha a izquierda:

```
typedef int[5]* ptr_to_array;
```

Valores literales

Podemos usar en nuestros programas valores constantes que escribimos literalmente. Según el tipo de dato que representan estos valores, tenemos notaciones y representaciones numéricas diferentes. En este compilador existen las siguientes:

```
-15;    // int en decimal
0xFF1A; // int en hexadecimal
0.514;  // float
true;   // bool
'a';    // int como caracter (no existe char)
"hola"; // int[5] (cadena de 4 caracteres + terminación 0)
NULL;   // puntero nulo
```

Los valores de literales booleanos son: true = 1, false = 0.

Mientras que en muchos compiladores de C el valor de NULL es 0, **en este compilador su valor es -1**. Esto es así porque en Vircon32 la dirección de memoria 0 es una dirección válida.

Sin embargo se puede seguir operando con NULL normalmente. Por ejemplo, la comprobación `if(puntero)`, aquí comprueba si el puntero es distinto de -1.

En valores float no está soportada la notación científica (por ejemplo, 1.35e-7).

Variables

Además de datos constantes podemos manejar variables: son direcciones de memoria en las que se guarda el valor que va variando. Se accede a ese valor mediante un nombre que identifica la variable, y un tipo de dato que interpreta el valor guardado.

Declarar variables

Una variable se declara con un tipo y un nombre, y opcionalmente se puede inicializar con un valor:

```
float Speed = 2.5;
bool Enabled;
```

Al igual que vimos al hablar de typedef, **en este compilador los tipos siempre se mantienen separados del nombre, en este caso: <tipo> <nombre> (= <valor>)**. Por ejemplo, para declarar un array en este compilador debe hacerse así:

```
// este es un array que tiene 20 arrays con 10 ints cada uno
int[20][10] LevelBricks;
```

Si tratamos de declarar un array como en C estándar, el compilador producirá un error.

Declaraciones múltiples

También se pueden declarar varias variables del mismo tipo en una sola declaración, separándolas con comas:

```
// declaramos 3 punteros a int
int* ptr1 = &number, ptr2 = ptr1, ptr3;
```

En este compilador todas las variables declaradas a la vez son siempre del mismo tipo (aquí, todas son `int*`), mientras que en C estándar sólo sería puntero la primera de ellas.

Listas de inicialización

Los arrays y estructuras se pueden inicializar con una lista de múltiples valores. Estas listas también se pueden anidar para tipos complejos, como se ve en los ejemplos:

```
// inicializar una estructura
struct Point
{
    int x, y;
};

Point P = { 3, -7 };

// usar listas anidadas para un array de estructuras
Point[3] TrianglePoints = { {0,0}, {1,0}, {1,1} };

// para arrays de int también podemos usar cadenas en vez de listas
int[10] Text = "Hello"; // cuidado, C añade 1 caracter extra (0) de terminación
```

Tipos de variables

En lenguaje C existen 2 tipos básicos de variables: locales y globales.

- Una variable local está declarada dentro del cuerpo de una función, y sólo es accesible en ese ámbito (porque se almacena en la pila, que varía con la ejecución).
- Una variable global se declara fuera de funciones y es accesible a todo el programa después de haber sido declarada porque su localización es fija.

En el C de Vircon32 se añade un tercer tipo de variable: los archivos incrustados. Se declaran usando `embedded` como en el siguiente ejemplo:

```
// el contenido de la variable se lee del archivo al compilar
embedded int[200][100] TileMap = "GameData\\LevelMap.bin";
```

Lo que hará el compilador en este caso es almacenar ese array en el cartucho en lugar de en la memoria RAM. Esto permite usar datos de mucho volumen sin tener que escribirlos en el código, aunque estando en el cartucho esos datos siempre son de sólo lectura. Este mecanismo nos permite usar datos externos en nuestro programa, aunque no existan archivos desde los que cargarlos externamente como se hace en un ordenador.

Es importante saber que el compilador exige que este archivo, además de existir, tenga el mismo tamaño que la variable que lo almacena. Es decir, en este caso el archivo LevelMap.bin deberá ocupar:

Nº de ints = $200 \times 100 = 20.000$
Bytes por cada int = 4 (son ints de 32 bits)
Tamaño total = 80.000 bytes

Los archivos incrustados sólo pueden usarse en el ámbito global, es decir, no pueden aparecer dentro de funciones (ya que nunca se almacenan en la pila).

Modificadores

En este compilador, por el momento, no existe ningún modificador en la declaración de variables: **const**, **static**, **volatile**, y **register** no están soportados. El único de ellos que podría llegar a implementarse en un futuro es **const**.

Funciones

El lenguaje C sólo no permite ejecutar sentencias fuera de una función más allá de declaraciones. Para poder ejecutar código en nuestro programa, necesitaremos declarar funciones que contengan las sentencias a ejecutar. Las funciones en el C de Vircon32 se declaran igual que en C estándar (**aunque con la notación de tipos propia que ya vimos**), por ejemplo:

```
// funcion que suma 2 enteros
int AddValues( int a, int b )
{
    return a + b;
}
```

El cuerpo de la función puede contener múltiples sentencias que se ejecutarán de forma secuencial.

Una diferencia importante con el C estándar es que, por limitaciones del compilador, **las funciones no pueden usar parámetros ni devolver valores de tamaño distinto de 1. Es decir: no pueden usar arrays, uniones o estructuras (a no ser que ocupen una sola palabra)**. En su lugar deben operar con punteros a los mismos.

Sí que es posible “pasar un array” como parámetro cuando el array decae a un puntero, de la misma forma que en lenguaje C estándar. Es decir, podemos hacer esto:

```
// funcion que suma N enteros
int SumValues( int* Values, int N )
{ // ... }
```



```
// array de 10 enteros
int[10] Values;

// sumamos los 5 primeros valores del array
int Sum = SumValues( Values, 5 );
```

El lenguaje C permite usar puntos suspensivos en la lista de argumentos de una función para poder recibir un número de parámetros indeterminado. Esto se usa en funciones estándar como printf. **Ese mecanismo no está soportado en este compilador.**

Expresiones

Las expresiones son operaciones que hacemos combinando variables y valores literales. Estas combinaciones las podemos hacer aplicándoles funciones u operadores, hasta que se llega a un único valor final como resultado.

Llamadas a funciones

Podemos llamar a cualquier función que ya haya declarada usando la notación estándar de paréntesis y comas:

```
// sumamos una variable y una constante
int Sum = SumValues( Variable, 5 );

// las llamadas se pueden anidar
PrintNumber( SumValues( Variable, 5 ) );
```

Cuando una función no devuelve ningún valor (es decir, su tipo de retorno es void), esa llamada es una expresión que no produce resultado y por tanto no puede ser usada a su vez por otras expresiones.

Operadores

Los operadores, en su mayoría, representan operaciones matemáticas que en lugar de escribirse como funciones se insertan como símbolos entre los operandos (como + y -). Este compilador usa mismos los operadores del lenguaje C estándar, y con su misma precedencia y asociatividad.

Las únicas excepciones son el operador ternario `a ? B : c` y el operador coma `a , b`, que no están soportados en este compilador.

Operadores especiales

El operador `sizeof()` determina el tamaño (siempre en palabras de 32 bits) de un tipo de datos, o del resultado de una expresión.

```
// tamaño del resultado de una expresión
int Size1 = sizeof( 2+5 );           // Size1 = 1 (int)

// tamaño de un tipo de datos
int Size2 = sizeof( int[2][5] );    // Size2 = 10
```

En el lenguaje C estándar existe también el operador de conversión de tipos explícita. **No existe en la versión actual del compilador, pero está previsto incorporarlo en el futuro.** Sin embargo el compilador sí hace las conversiones implícitas donde se necesitan.

Control de flujo

Un programa en C no se construye sólo con secuencias de expresiones, sino que necesita mecanismos para controlar el flujo de ejecución. Disponemos de los siguientes.

Condiciones

La manera más básica de controlar hacia dónde continúa la ejecución es una simple condición. Usando `if` y `else` podemos evaluar una condición y determinar lo que debe ocurrir si se cumple, y si no se cumple.

También podemos enlazar varios ifs para comprobar condiciones relacionadas.

```
// ifs encadenados
if( x > 0 )
    print( "positive" );

else if( x < 0 )
    print( "negative" );

else
{
    print( "zero" );
    showAlert();
}
```

Switches

Si necesitamos elegir entre varios valores enteros, en vez de encadenar múltiples condiciones if-else podemos usar `switch` para elegir entre un conjunto de casos:

```
switch( WeaponPowerLevel )
{
    // impacto de potencia media
    case 1:
    case 2:
        Player.Health -= WeaponPowerLevel;
        break;
}
```

```

// impacto fuerte que destruye al jugador
case 3:
    MakePlayerExplode();
    break;

// para impactos muy débiles, o posibles casos de error
default:
    MakeBulletBounce();
    break;
}

```

Etiquetas y goto

Podemos definir etiquetas dentro de una función para marcar ciertas posiciones en el código. Luego si usamos `goto` la ejecución de nuestro programa saltará a esas posiciones.

```

// las etiquetas sólo son válidas dentro de funciones
Void ProcessData( DataStructure* Data )
{
    Step1:

    // evitar este paso en algunos casos
    if( SkipPreprocessing )
        goto Step2;

    Preprocess( Data );

    Step2:

    // seguir procesando datos hasta que el error sea lo bastante bajo
    if( !Transform( Data ) )
        goto Step1;
}

```

Bucles

Nos permiten repetir una sección del programa hasta que se cumpla una condición. Existen 3 tipos de bucles en C: `while`, `do` y `for`.

```

// while comprueba la condición al principio
while( x != 0 )
    x /= 2;

// do comprueba la condición al final
do
{
    x -= 7;
    y = x;
}
while( x > 0 );

// for es un bucle más configurable
for( int i = -5; i <= 5; i += 2 )
    print_number( i );

```

Control de bucles

En el interior de un bucle podemos usar `break` para terminar el bucle y continuar la ejecución después de su final. Por otra parte, `continue` nos permite adelantar la siguiente repetición del bucle sin tener que llegar al final del mismo.

```
while( Enemy.EnemyID < MaxEnemyID )
{
    ++Enemy;

    // procesar solo enemigos activos
    if( !Enemy.Active )
        continue;

    ProcessEnemy( &Player, Enemy );

    // no hace falta continuar si el jugador ha muerto
    if( Player.Health <= 0 )
        break;
}
```

Retorno de funciones

En el cuerpo de una función, podemos usar `return` para salir de la función en cualquier momento. La ejecución volverá al punto del programa donde se llamó a esa función.

Return también se usa para devolver un valor (cuando el tipo de retorno de la función no es void). En ese caso debe usarse return con un valor de tipo compatible.

```
// funcion que no devuelve valores
void DoNothing()
{
    return;
}

// funcion que devuelve un puntero
int* FindLetterA( int* Text )
{
    while( Text )
    {
        if( *Text == 'A' )
            return Text;

        Text++;
    }

    return NULL;
}
```

Incrustar código ensamblador

El lenguaje C de Vircon32, al igual que algunas otras implementaciones, permite incrustar instrucciones en lenguaje ensamblador usando `asm`. Esta característica sólo se puede utilizar dentro del cuerpo de una función. La sintaxis es muy similar a la que usa el compilador gcc, y se usa una cadena de texto para escribir cada instrucción.

```
// funcion de C implementada en ensamblador
void select_sound( int sound_id )
{
    asm
    {
        // usamos el valor del parametro sound_id
        "mov R0, {sound_id}"
        "out SPU_SelectedSound, R0"
    }
}
```

Además podemos utilizar las variables de nuestro programa en C en las instrucciones de ensamblador. Para ello usamos el nombre de la variable entre llaves.

Esta manera de conectar C y ensamblador tiene limitaciones: sólo soporta nombres, no expresiones. Por ejemplo, no podríamos usarla para leer un miembro de un array. Además sólo se permite referenciar una variable en cada instrucción. Debido a cómo funciona la CPU de Vircon32, nunca será posible usar más de una variable a la vez.

El preprocesador

El lenguaje C realiza un procesamiento previo del texto de nuestros programas en C. El preprocesador recorre línea por línea los archivos que forman el programa buscando directivas, y aplica las que encuentra y reconoce.

Las directivas comienzan con el carácter almohadilla `#`, que debe ser el primer carácter de la línea (salvo espacio en blanco). Las directivas que se pueden usar son las siguientes.

Directiva `#include`

Usar `#include` nos permite insertar en un punto de nuestro código el contenido de otro archivo. Es útil para poder organizar nuestro programa separando las distintas características en archivos separados.

```
// incluimos nuestras cabeceras
#include "Enemies\\Boss.h"
```

El preprocesador buscará primero en el directorio de la librería estándar, y después en el directorio del propio archivo fuente.

En C se permite escribir la ruta del archivo entre comillas ("ruta") o bien entre ángulos (<ruta>). Este compilador sólo permite usar las comillas. La ruta, además, se trata como cualquier otra cadena de texto: si contiene caracteres especiales como '\', deben escribirse con su secuencia de escape.

Directivas #define y #undef

El preprocesador mantiene una lista de variables internas (no confundir con las variables del programa en C). Podemos definir una variable con `#define`, y eliminar una definición con `#undef`. Las definiciones pueden tener un valor pero también pueden estar vacías.

```
// definición simple con un valor literal
#define BallDiameter 16

// definición que usa la anterior en una expresión
#define BallRadius (BallDiameter / 2)

// eliminamos definiciones antiguas
#undef BallSize
```

La directiva define tiene limitaciones en este compilador. No se puede emplear con parámetros como en el preprocesador de C estándar. Además, no está soportado el uso de los operadores `#` y `##` dentro de las definiciones para formar cadenas de texto.

Directivas condicionales

En C se puede usar el preprocesador para delimitar partes del programa que se deben compilar y otras que no. Esa compilación condicional se consigue usando directivas que comprueban si una variable del preprocesador está definida o no, y su valor. Las directivas condicionales que permiten esto son `#ifdef`, `#ifndef`, `#else` y `#endif`.

```
#define DEBUG // comentar esta linea cambiaria el programa

#ifdef DEBUG
    Debug = true;
#else
    Debug = false;
#endif
```

Una limitación a tener en cuenta es que en este compilador, las partes no compiladas no se analizan pero sí se leen a un nivel básico (para reconocer cadenas, números, etc). Por tanto podríamos recibir algunos errores incluso en regiones que no se compilan.

Las directivas `#if` y `#elif` del lenguaje C estándar no están soportadas por el momento.

Directivas #error y #warning

Estas directivas nos permiten obligar al compilador a mostrar mensajes durante la compilación. Si usamos `#warning`, el compilador sólo mostrará una advertencia. En cambio, con `#error` se detendrá la compilación.

```
// tras esta linea nuestra compilacion continua
#warning "careful, error on the next line"

// pero esta linea la detendra con un error
#error "this part should not have been compiled"
```

Otras características no soportadas

Otras directivas del lenguaje C, como `#line` y `#assert`, no están soportadas.

Además, en un preprocesador de C estándar existen algunas variables internas que son `__FILE__`, `__LINE__`, `__DATE__`, `__TIME__` y `__func__`. Se usan para crear cadenas de texto de forma dinámica al compilar, en función de distintos parámetros: la hora, el nombre del archivo fuente, etc. En este compilador, no existen esas variables.

La librería estándar

Cualquier compilador de C debe implementar una serie de funciones en diferentes ámbitos (matemáticas, manejo de cadenas, etc) para que los programas dispongan de una funcionalidad mínima, y de forma homogénea. Estas funciones son accesibles a los programas incluyendo en el código las cabeceras estándar del lenguaje.

La librería estándar de C para Vircon32 incluye un buen número de funciones del estándar de C, pero no todas. Una de las ausencias más destacadas son las funciones para emplear memoria dinámica (malloc, calloc, realloc y free).

Igualmente, por ser un compilador para una consola específica, la librería estándar de Vircon32 añade además funciones para trabajar con los distintos sistemas de la consola: audio, video, mandos, etc.

La colección actual de librerías en el compilador de Vircon32 es la siguiente:

“audio.h”

Sirve para usar el chip de audio y reproducir sonidos.

“input.h”

Permite leer el estado de los mandos.

“math.h”

Las funciones matemáticas más habituales del lenguaje C.

“memcard.h”

Sirve para acceder a la tarjeta de memoria y leer ó guardar datos.

“misc.h”

Funciones misceláneas: manejo de memoria, números aleatorios y otros.

“string.h”

Incluye funciones para construir y manejar cadenas de texto.

“time.h”

Nos permite medir el paso del tiempo y controlar la velocidad de los programas.

“video.h”

Sirve para acceder al chip de video y mostrar imágenes en pantalla.

Diferencias globales con C estándar

La función main

La función main es una función especial que siempre debe existir, ya que la ejecución de nuestro código siempre comienza al principio de esta función. En el lenguaje C estándar, el prototipo de la función main es el siguiente:

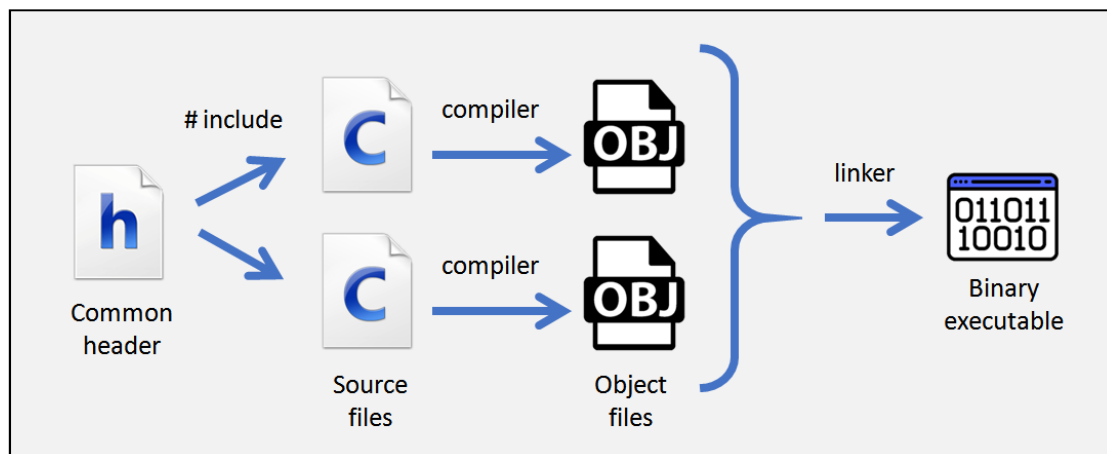
```
int main( int NumberOfArguments, char* Arguments[] )
{
    // ...
}
```

Vircon32 es una consola en la que no existe un sistema operativo y, tras el arranque inicial de la BIOS, nuestro programa es el único que se está ejecutando en la máquina. Como no existe ningún otro programa que nos pueda pasar parámetros o al que podamos devolver un valor, **en Vircon32 no tiene sentido ese prototipo de main. Por ello, la función main en este compilador debe declararse así:**

```
void main()    // o bien: void main( void )
{
    // ...
}
```


Estructura de un programa

Un compilador de C estándar permite dividir un programa en varias “unidades de compilación”. Esto significa que podemos tener varios archivos con sus propias variables y funciones que se compilan por separado y luego se unen mediante un enlazador.



Normalmente, cada uno de esos archivos puede acceder a las funciones de los demás incluyendo una cabecera común que define los prototipos de esas funciones, pero no las implementa. Es lo que se llama una declaración parcial.

Este compilador sigue una estructura simplificada: **no existen declaraciones parciales, y no hay un enlazador que pueda unir varios archivos compilados. La manera de usar funciones de otros archivos es incluir esos archivos desde el archivo principal, con todo su código completo.**

La librería estándar, por ejemplo, también está hecha de esta manera e incluye las implementaciones completas de todas sus funciones.

Limitaciones generales del compilador

El compilador de C de Vircon32 ha sido programado por una sola persona y con conocimientos limitados. Aunque funciona relativamente bien, tiene algunas limitaciones comparado con lo que sería un compilador más serio en cuanto al proceso de compilación en sí.

Optimización

Este compilador no produce código muy optimizado. Sólo detecta ciertas situaciones muy sencillas en las que optimizar, y si analizamos el código ensamblador que genera no nos será difícil encontrar secciones que tienen instrucciones innecesarias.

Detección de errores

La detección de errores en el compilador informa correctamente de la localización de cada error. Sin embargo en bastantes situaciones detiene el programa al primer error detectado, mientras que otros compiladores podrían dar varios errores al mismo tiempo.

Juego de caracteres

Por el momento este compilador no soporta Unicode. Se pueden usar caracteres que no sean los del inglés, pero los archivos fuente se van a interpretar siguiendo la codificación de Windows Latin 1/CodePage 1252. La fuente de texto que incluye la BIOS (la que se utiliza en print) también está codificada de esta forma. Esta codificación incluye 256 caracteres, siendo los primeros 128 los del ASCII estándar. En el resto están incluidos todos los caracteres del español, portugués, francés y alemán, además de algunos símbolos de uso habitual como el euro.

Usando el compilador

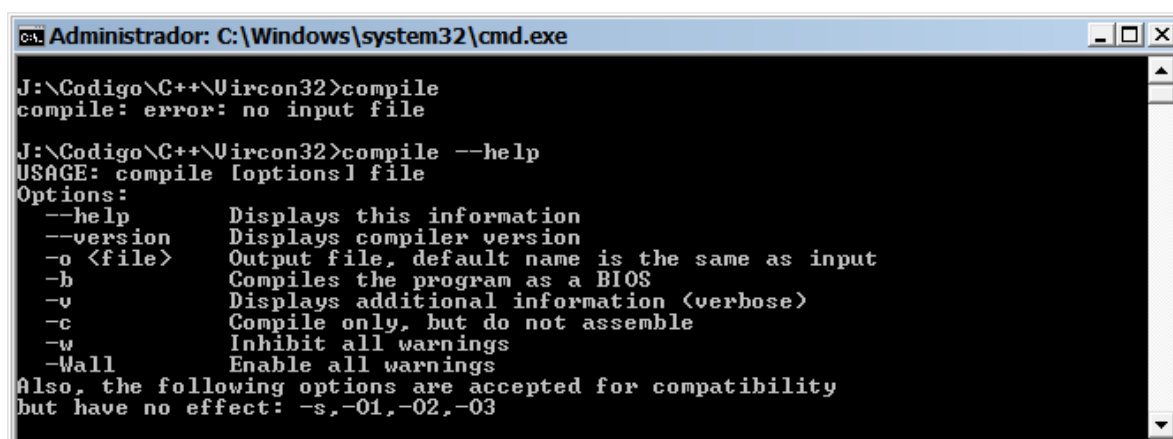
A grandes rasgos hay 2 maneras de usar el compilador, y en general las herramientas de desarrollo de Vircon32.

Línea de comandos

Las herramientas de Vircon32 son programas de línea de comandos, con lo que se puede llamar al programa a mano o con un script para compilar según se necesite. El código se puede escribir en cualquier editor de texto. La desventaja de hacerlo así es que si el compilador nos informa de algún error deberemos observar el mensaje y localizar a mano el error en nuestro código.

Los parámetros de entrada del compilador se han elegido para que se use igual que el compilador gcc, como muestra la imagen de debajo. Es decir, el comando es de la forma:

`compile.exe [opciones] archivo`.



```
C:\Windows\system32\cmd.exe
J:\Codigo\C++\Vircon32>compile
compile: error: no input file

J:\Codigo\C++\Vircon32>compile --help
USAGE: compile [options] file
Options:
  --help      Displays this information
  --version   Displays compiler version
  -o <file>   Output file, default name is the same as input
  -b          Compiles the program as a BIOS
  -v          Displays additional information <verbose>
  -c          Compile only, but do not assemble
  -w          Inhibit all warnings
  -Wall       Enable all warnings
Also, the following options are accepted for compatibility
but have no effect: -s, -01, -02, -03
```

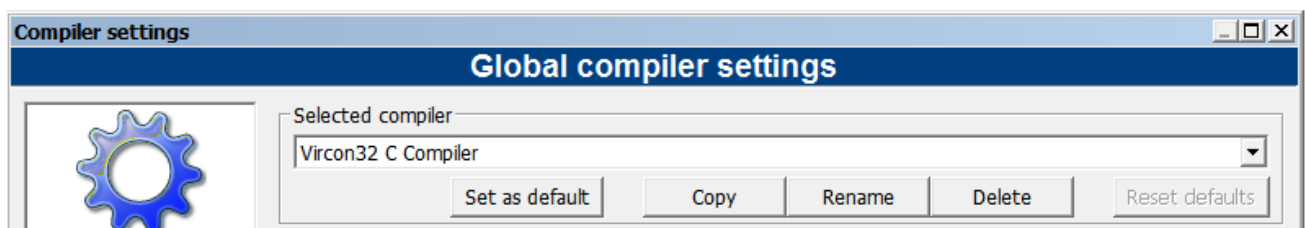
Aún usando los mismos parámetros, sigue habiendo 2 diferencias importantes en su uso comparado con gcc:

- Este compilador sólo genera código ensamblador (con extensión por defecto .asm), y no crea binarios por sí mismo. Para ello hace falta llamar después al programa ensamblador `assemble.exe`.
- El binario que se crea con el ensamblador tampoco es directamente ejecutable por Vircon32. Esto se debe a que debemos empaquetar el binario en una ROM usando el empaquetador `packrom.exe`. De esta forma la ROM integra el binario junto con sus imágenes y sonidos, si se usan.

Integración con un IDE

La presentación de los errores y advertencias del compilador de Vircon32 también se han elegido para coincidir con los del compilador gcc. Esto hace más sencillo que se pueda integrar en entornos de desarrollo si son lo bastante configurables.

Por ejemplo una de las opciones sería crear una configuración para el compilador de Vircon32 a partir de la de gcc (que debería existir en cualquier IDE), y configurar nuestros proyectos para usarlo.



En concreto, en el entorno Code::Blocks se puede trabajar de varias formas con las herramientas de Vircon32. De hecho, mediante scripting se puede incluso añadir menús de opciones personalizados como se ve en la imagen a continuación.

