Vircon32: Learning C language

Document date 2023.01.23 Written by Carra

What is this?

This document is a quick guide to get started programming in C language if you didn't know it before. The purpose of this guide is to give a basic understanding about the structure of a program, basic features of the C language and show what is needed for anyone wanting to create C programs for the Vircon32 console. Still, you are expected to know basic programming concepts, such as functions, variables, arrays, etc.

Something you should know!

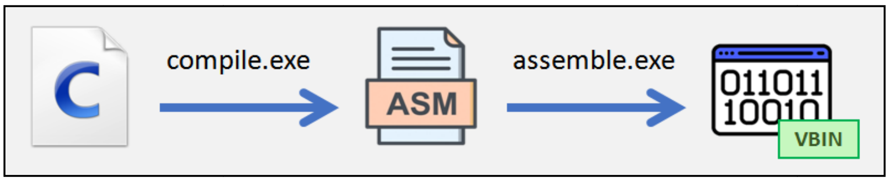
This guide only teaches the basic features of C: the minimum needed to create C programs. For clarity, this guide deliberately omits several of the more advanced language features. Also, you should be aware that there are minor differences between standard C language and the one used in Vircon32. This guide focuses on the Vircon32 version. For more details about this, please read the guide for the Vircon32 C compiler.

Summary

This document is organized into sections, each of which will progressively show more details about the C language used in Vircon32 and its compiler.

Introduction

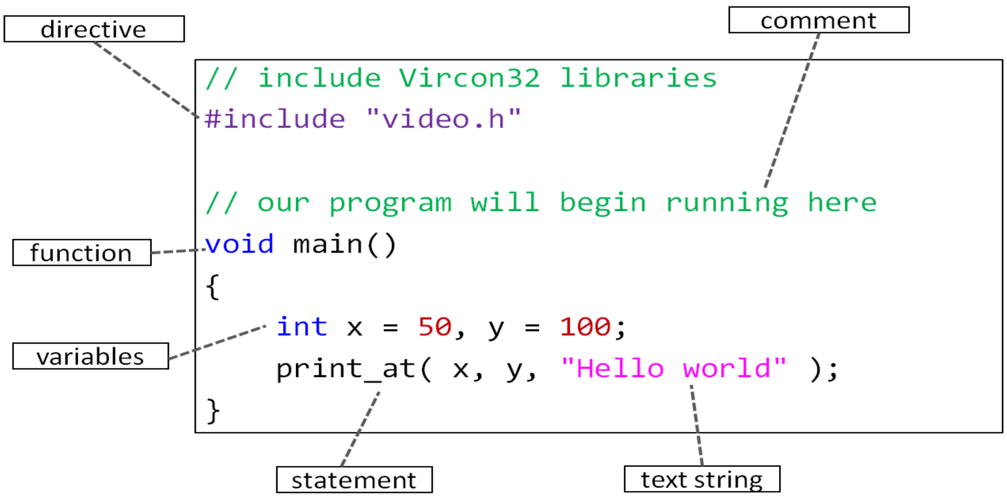
C is a compiled language. This means that before being able to run, the code you write needs to first be transformed into machine instructions that the Vircon32 CPU is able to execute. In this console the process looks like this:



After that binary file with CPU instructions is created, you will need to pack that program into a .v32 file together with the images and sounds it needs. But this guide only focuses on writing the programs. You can read about the whole process of game creation in the other Vircon32 guides.

Structure of a C program

It is mandatory to start any language tutorial with a simple hello world program, so here you go. Here you can see the most basic structure of a C program.



```
// include Vircon32 libraries
#include "video.h"

// our program will begin running here
void main()
{
    int x = 50, y = 100;
    print_at( x, y, "Hello world" );
}
```

The main function

The main function is a special function that must always exist, since execution of our code always starts at the beginning of this function. The main function in this compiler takes no arguments and returns no value, so it must be declared with this prototype:

```
void main()   // either this or: void main( void )
{
    // ...
}
```

Comments

Comments are a way to include in the program our own explanations without having the compiler try to interpret them (which would cause errors). That is to say, comments are not part of the program itself. There are 2 types of comments, and they work the same way as in the standard C language:

Line comment

These comments begin with `//`, and finish at the end of the line.

```
int x = 7 + 5;    // comment affects only this line
int y = x;
```

Block comment

A block comment begins with `/*`, and will not finish until we end it with `*/`, even if it's done in a different line.

```
/* this comment negates the line below
int x = 7 + 5;
*/

int y;
```

Data types

All values or variables handled by the language belong to some data type that indicates how to interpret that value. Here we will see what data types we can use.

Basic types

Vircon32 organizes memory in 32-bit words because in this console the minimum unit of memory is the word (and not the byte), so we must take into account that:

- All sizes for data types are measured in words and not bytes
- All memory addresses and offsets are given in words and not bytes

The 4 basic data types in this compiler are:

```
int     float     bool     void
```

All of these types have size 1 (i.e. 32 bits). Type void is special, and can only be used on its basic form to indicate that a function does not return any value.

Derived types

We can create new types using the basic types in arrays and pointers. This can be done multiple times, for example:

```
int*    void**    float[3][5]    void*[4]
```

We can use void to create pointers, but not arrays (because there can be no void type values). C allows pointers to void as a way to refer to a memory address where information is read or stored as just bits, without interpreting it (void indicates the absence of type).

Compound types

We can group several data types into a larger type, by creating structures and unions. Each member of these groups is a field that is accessed by name.

```
// declare type Point
struct Point
{
    int x, y;
};

// declare type Word
union Word
{
    int AsInteger;
    void* AsPointer;
};

// use some structures
Point P1,P2;
P1.x = 10;
P1.y = -7;
P2 = P1;

// use some unions
Word W;
W.AsInteger = 0xFF110AF;
int* Pointer = W.AsPointer;
```

Structures and unions, in turn, may also contain other structures and unions. They can also include themselves, but only through pointers for obvious reasons.

Enumerations

We can define a series of integer constants and group them into their own type. The C language allows this through enumerations. In general these values are treated as integers and you can use them in the same ways, but their type is more restrictive.

If no values are specified, first constant will have a numerical value of 0 and every following constant has a value equal to the previous constant plus 1.

```
// declare type Semaphore
enum Semaphore
{
    Red = 1,
    Yellow,   // has value 2
    Green,    // has value 3
};

// these operations are fine
Semaphore S1,S2;
S1 = Red;
S2 = S1;
int Value = Yellow + Green;  // can safely convert enum to int

// these assignments would produce errors!
S1 = Green - Red;
S2 = 1;    // same value as Red, but can't convert int to enum
```

Literal values

In our programs we can use constant values that we write literally. Depending on the data type each of these values is representing, we have different notations and numerical representations. In this compiler there exist the following:

```
-15;     // int in decimal
0xFF1A;  // int in hexadecimal
0.514;   // float
true;    // bool
'a';     // int as a character (char does not exist)
"hi!";   // int[4] (string of 3 characters + terminating 0)
NULL;    // null pointer
```

The values for boolean literals are: true = 1, false = 0.

Variables

In addition to constant data we can also handle variables: they are memory addresses where the value that varies is stored. This value is accessed by using a name that identifies the variable, and a data type that interprets the stored value.

In C, your variables can only store the type of values with which they are declared (for example, integers). In some other languages you can declare a typeless variable and keep storing different values to it (numbers, strings, etc). This is not allowed in C.

Declaring variables

A variable is declared with a type and a name, and can optionally be initialized with a value:

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

In this compiler types are always kept separate from the name, in this case: <type> <name>. For example, to declare an array in this compiler it must be done like this:

```
// this is an arrays of 5 floats
float[5] BallSpeeds;

// this is an array with 20 positions, each of which is an array of 10 ints
int[20][10] LevelBricks;
```

Multiple declarations

It is also possible to declare several variables of the same type in a single declaration, separating them with commas. Variables declared in this way will all be of the same type.

```
// declare 3 pointers to int
int* ptr1 = &number, ptr2 = ptr1, ptr3;
```

Initialization lists

Arrays and structures can be initialized with a list of multiple values. These lists can also be nested for more complex types, as seen in these examples:

```
// initialize a structure
struct Point
{
    int x, y;
};

Point P = { 3, -7 };

// here we use nested lists for an array of stuctures
Point[3] TrianglePoints = { {0,0}, {1,0}, {1,1} };

// for int arrays we can also use a string instead of a list
int[10] Text = "Hello";  // careful, C adds 1 extra character (0) as termination
```

Scope of a variable

In C language 2 basic kinds of variables exist: locals and globals.

- A local variable is declared in the body of a function, and is only accessible within its scope (because it's stored in the stack, which changes through execution).

- A global variable is declared outside functions and is accessible to the whole program after its declaration because its address is fixed.

Pointers

Pointers are not really needed to create basic games, but you may want to have an idea of what they are. A pointer is simply a variable that contains a memory address (i.e. a number that represents a position in memory). For example we could have:

```
int Variable = 7;    // let's say Variable is stored in memory position 1250
int* Pointer = &Variable;  // Pointer now contains the number 1250
```

In this case Pointer is declared as type int*, which means that whatever memory position it is pointing to, will be interpreted as an integer. So, we can read or modify the value stored in our variable in an indirect way, by using our pointer as an intermediary.

Why are pointers useful?

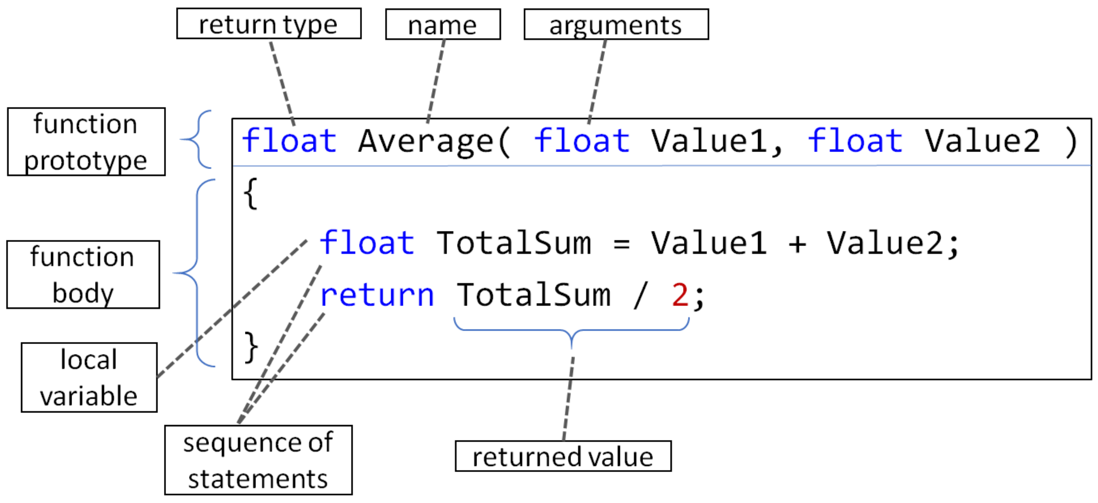
Pointers can be used for many things, but a typical example is passing a pointer to a function to allow it to manipulate external data. For example you could have a function that returns 2 values, by passing it 2 pointers to external variables. Or, if you have a structure or array too large to pass as a parameter, you just work with a pointer to it and read/modify that structure in place.

The NULL pointer

C also has a special literal value, NULL, which is intended for pointers. NULL is provided so that we can have a way of specifying that a pointer is currently not pointing to anything. The value of NULL is usually set to some invalid memory address so that, if we try to read/write from it, it will produce a memory violation. This helps detect program errors.

Functions

In C language it is not allowed to execute any statements outside of a function (other than just declarations). In order to execute code in our program, we will need to declare functions that contain the statements to be executed. Functions in C are declared in this way:



The body of the function can contain multiple statements that will be executed in a sequential manner.

Returning from functions

Inside the body of a function, we can use `return` to exit the function at any time. The execution will return to the point in the program where that function was called.

Return is also used to return a value (when the return type of the function is not void). In that case return must be used with a value of compatible type.

```
// function that does not return a value
void DoNothing()
{
    return;
}

// function that returns a pointer
int* FindLetterA( int* Text )
{
    while( Text )
    {
        if( *Text == 'A' )
          return Text;

        Text++;
    }

    return NULL;
}
```

Limitations of functions

An important difference with standard C is that, due to compiler limitations, functions cannot receive parameters or return values of size different from 1. That is: they cannot use arrays, unions or structures (unless their size is just a single word). Instead they must operate with pointers to them.

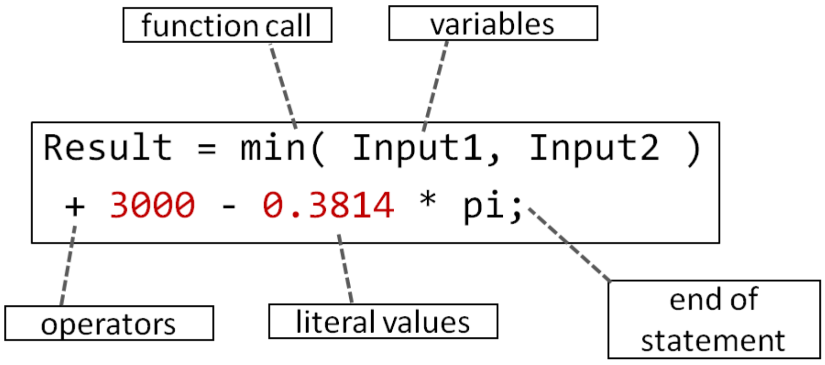
It is however possible to "pass an array" as a parameter when the array decays to a pointer, in the same way as in standard C language. As an example, we may do this:

```
// function that adds N integers
int SumValues( int* Values, int N )
{  // ...  }


// array of 10 integers
int[10] Values;

// we add the first 5 values in the array
int Sum = SumValues( Values, 5 );
```

In this case, the "array decay" made array Values evaluate as a pointer to its first element, and was therefore accepted by a function which expected a pointer to integer.

Expressions

Expressions are operations that we can do by combining variables and literal values. These combinations can be made by applying functions or operators to them, until we obtain a single final value as a result. See those elements in this example expression:



Operators

For the most part operators represent mathematical operations that, instead of being written as functions, are inserted as symbols between the operands (such as + and -). This compiler uses the same operators of the standard C language, and with their same precedence and associativity.

These are all the available operators, grouped by type:

Arithmetic operators

Operator	Meaning
+a	a (no effect)
-a	Change sign
a + b	Sum
a - b	Difference
a * b	Product
a / b	Division
a % b	Modulo

Comparison operators

Operator	Meaning
a == b	Equal to
a != b	Not equal to
a < b	Less than
a <= b	Less or equal to
a > b	Greater than
a >= b	Greater or equal to

Logic operators

Operator	Meaning
!a	Logic NOT
a || b	Logic OR
a && b	Logic AND

Bitwise operators

Operator	Meaning
~a	Bitwise NOT
a | b	Bitwise OR
a & b	Bitwise AND
a ^ b	Bitwise XOR
a << b	Bit shift left
a >> b	Bit shift right

Increment & decrement operators

Operator	Meaning
a++	a is first evaluated and THEN incremented by 1
a--	a is first evaluated and THEN decremented by 1
++a	a is first incremented by 1 and THEN evaluated
--a	a is first decremented by 1 and THEN evaluated

Assignment operators

Operator	Meaning
`a = b`	Assignment
`a += b`	a = a + b
`a -= b`	a = a - b
`a *= b`	a = a * b
`a /= b`	a = a / b
`a %= b`	a = a % b
`a |= b`	a = a | b
`a &= b`	a = a & b
`a ^= b`	a = a ^ b
`a <<= b`	a = a << b
`a >>= b`	a = a >> b

Operators for pointers

Operator	Meaning
`&a`	Memory address of a (makes a pointer to a)
`*a`	The value pointed by a (contents of that address)
`!a`	a == NULL

Operators for arrays

Operator	Meaning
`a[ b ]`	Element with index b within array a

Operators for structures and unions

Operator	Meaning
`a . b`	Member b within a
`a -> b`	Member b within the structure pointed by a

Other operators

Operator	Meaning
`(a)`	Operations inside are grouped and evaluated before other operators
`sizeof(a)`	Returns the size of its argument, in words

Type promotions

As programmers we should be aware that the language will perform automated conversion between basic data types when needed. For example, if we do this:

```
int PlayerScore = 28.3;   // float needs to be converted to int
```

The compiler will automatically convert this value to an integer and the variable will actually store 28. Something similar also happens when an expression operates with different types: to add int + float, the int must first be converted to float so that operation is now valid between coherent types (float + float).

Function calls

We can call any function that has already been declared using the standard notation of parentheses and commas:

```
// add a variable and a constant
int Sum = SumValues( Variable, 5 );

// function calls can be nested
PrintNumber( SumValues( Variable, 5 ) );
```

When a function does not return any value (i.e. its return type is void), a call to that function is an expression that does not produce a result and therefore cannot be used by other expressions.

Sizeof operator

The `sizeof()` operator determines the size (always in 32-bit words) of either a data type or an expression’s result.

```
// size of an expression's result
int Size1 = sizeof( 2+5 );        // Size1 = 1 (int)

// size of a data type
int Size2 = sizeof( int[2][5] );  // Size2 = 10
```

Flow control

A C program is not only built with sequences of expressions, but will also need tools to control the execution flow. The following are available.

Conditions

The most basic way to control where execution will continue is a simple condition. Using `if` and `else` we can evaluate a condition and determine what should happen if it is met, and what when it is not met.

We can also chain together several ifs to check for related conditions.

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

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

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

Switches

If we need to choose from several integer values, instead of chaining multiple if-else conditions we can also use `switch` to select from a series of cases:

```
switch( WeaponPowerLevel )
{
    // mid-power impact
    case 1:
    case 2:
        Player.Health -= WeaponPowerLevel;
        break;

    // hard impact that destroys player
    case 3:
        MakePlayerExplode();
        break;

    // for impacts too weak, or possible error cases
    default:
        MakeBulletBounce();
        break;
}
```

Loops

C supports direct jumps using `goto`, but loops are a better alternative. They allow us to repeat a section of the program until a condition is met. There are 3 types of loops in C: `while`, `do` and `for`.

```
// while checks the condition at the beginning
while( x != 0 )
  x /= 2;

// do checks the condition at the end
do
{
    x -= 7;
    y = x;
}
while( x > 0 );

// for is a more configurable loop: it features an initial
// statement, an exit condition and an iteration statement
for( int i = -5; i <= 5; i += 2 )
  print_number( i );
```

Loop control

Inside a loop we can use `break` to terminate the loop and continue execution after its end. On the other hand, using `continue` allows us to advance to the next repetition of the loop without having to reach the end of the loop.

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

    // process only active enemies
    if( !Enemy.Active )
      continue;

    ProcessEnemy( &Player, Enemy );

    // no need to continue if player has died
    if( Player.Health <= 0 )
      break;
}
```

The preprocessor

C language performs a preprocessing of all text in our C programs. The preprocessor walks line by line through the files that make up the program looking for directives and applies those it can find and recognize.

Directives begin with the hash character `#`, which must be the first character in the line (except for whitespace). The directives that can be used are the following.

Directive #include

Using #include allows us to insert at some point in our code the content of another file. It is useful because it enables us to organize our program by separating its different features in separate files.

```
// include our headers
#include "Enemies\\Boss.h"
```

The preprocessor will first look for the file in the standard library folder, and then in the directory of the source file itself. The path is treated like any other text string: if it contains special characters such as '\', they must be written using their escape sequence.

Directives #define and #undef

The preprocessor maintains a list of internal variables (not to be confused with C program variables). We can define a variable with #define, and delete a definition with #undef. Definitions can have a value but they can also be empty.

```
// simple definition with a literal value
#define BallDiameter  16

// definition that uses the previous one in an expression
#define BallRadius  (BallDiameter / 2)

// now remove other old definitions
#undef BallSize
```

The define directive has some limitations in this compiler. It cannot be used with parameters as in the standard C preprocessor.

The standard library

Any C compiler must implement a series of functions for different subjects (mathematics, string handling, etc.) so that programs can access some minimum functionality, that works in a uniform manner. These functions are accessible to programs by including the standard language headers in the code.

The standard C library for Vircon32 includes a good number of standard C functions, but not all of them. On the other hand, since this is a compiler for a specific console, the Vircon32 standard library also adds functions to work with the different console systems: audio, video, gamepads, etc.

The current collection of headers in the Vircon32 compiler is the following:

“audio.h”
It is used to operate the audio chip and play sounds.

“input.h”
It allows to read the state of gamepads.

“math.h”
The most common mathematical functions of the C language.

“memcard.h”
Used to access the memory card and read or save data.

“misc.h”
Miscellaneous functions: memory management, random numbers and others.

“string.h”
Includes functions for building and handling text strings.

“time.h”
It allows us to measure the flow of time and control the speed of the programs.

“video.h”
Used to access the video chip and show images on screen.