ccZ80++ language specification

http://ccz80pp.webcindario.com

Index

Introduction

The ccZ80++ language allows to write programs based on the Intel Z80 microprocessor, as Amstrad CPC, Amstrad PCW, MSX and Spectrum among the most popular computers.

Be used on a Windows system or even from Linux using WINE or Mono or from Mac using Mono. Microsoft .NET 4 Client Profile or another higher version needs to be installed, you can download from http://www.microsoft.com/en-us/download/details.aspx?id=24872.

It can be considered an evolution of ccz80 language, but ccZ80++ adds class definition, use of objects and sintax closer to the C++ language to allow faster learning.

Capitalization differ on keywords and identifiers defined in the program. Not differ for filenames in the Windows environment.

Although it is recommended to write the code in a structured format, newlines, blank lines and spaces do not affect the generation of object code.

To get a first impression of this language then his version of "Hello World" is given:

```
include Text.ccz80++

class MainClass
{
  static void main()
  {
    Text.PrintString("Hello World!");
  }
}
```

This first program uses a Text class and its static member printString, included in Text.ccZ80++ file, which versions for Amstrad CPC 464, CPC 6128 Amstrad, Spectrum and MSX are provided in the download section of the page web.

Syntax of compiler

The ccZ80++ compiler is used from the command line, producing an object file output of the compilation of the type described in the corresponding parameter. This file is created, overwriting if there previously, with the same name as the main source file and one of the following extensions as the result type requested to the compiler:

- **. cc80** if indicated ccz80 output code.
- **. asm** if indicated assembly output code.
- **.bin** if indicated machine code output.

After the build is returned in the ERRORLEVEL variable value 0 if it was successful, or 1 if an error has occurred preventing compilation.

All output information produced by the compilation shown in the same command line window from which you invoked the compiler.

The compiler is fully integrated into the file ccZ80++.exe which is available in the download section of the website.

The compiler syntax is:

```
ccZ80++ file /org=address /out=type /msg=level /path=paths
```

The order of the parameters given to the compiler and indicate whether upper or lower case is not significant.

Besides ccZ80++ which is the name of the corresponding file to the compiler, the meaning of each element is:

file

Required. The name of the main program source file, which has recommended extension. ccZ80++. It is not necessary that this is the wrapper class that contains the main function, you can include other files, one of which contain the class with the main function.

/org=address

Optional. Specify a start address to generate the binary code resulting from the compilation, where address is the start address can be specified in decimal or hexadecimal with the prefix #. If this parameter is not specified the code generated from address 0.

/out=type

Optional. Specifies the type of result given by the compiler, where type can be:

- **ccz80** to generate ccz80 code.
- **assembler** to generate assembly code.
- **binary** to generate machine code.

If this parameter is not specified machine code is generated.

/msg=level

Specifies the level of messages that show the compiler. There are three levels, from lowest to highest importance:

- notes, reporting on conditions not impede or prejudice the compilation speed or size of the resulting machine code.
- warnings, reporting on conditions that do not prevent compilation, but may impair the speed or size of the resulting machine code.
- errors, that prevent the generation of machine code.

As level may indicate:

- **all: d**isplays messages from all levels, notes, warnings and errors.
- **warning: d**isplays only warning and error level messages.
- **error: d**isplays only error level messages.

If this parameter is not specified all messages are showed.

/path=paths

Allows you to specify a list of paths where the compiler should look for the source files, both indicated to the compiler as main source file as the given files in the include statements within the program.

You can give several paths, separated by semicolons.

If the main source file or include files are specified with a full path indicated no use of the routes determined by this choice is made; if indicated by a relative path, the file with its relative path is sought in principle from the current path of the command line and then on all routes provided by this option, in the order they are written.

Examples of use of the compiler:

```
ccz80++ C:\Source\games\asteroid.ccz80++ /org=#4000 /msg=warning
ccz80++ "Source files\program.ccz80++" /org=5000 /out=assembler
ccz80++ ..\test.ccz80++ "/path=C:\Source;C:\ccz80++\Include files"
```

Remember that if a parameter contains spaces, as the main source file name or path routes parameter, the parameter must be enclosed in double quotes indicate.

Compile error handling

In detecting errors the compiler reports the line, position and file where the error is. Keep in mind that the position reports on the number of character in the line where the error occurred, so if the line contains tabs might not match the position indicated in the error column in the editor text you are using, because a tab is a position, but may correspond to several spaces in the text editor.

Sometimes, if it is an undetectable error for the ccZ80++ compiler and has been produced in the compilation that makes ccz80 or rear assembly, the error information is in reference to the file ccz80 or assembler file that they are generated during program compilation. In those cases should be compiled with the /out=ccz80 or /out=assembler to generate these files and study the cause of the error on them.

Data types in ccZ80++

In a program can handle the following types of data:

- **short**: value of 8-bit unsigned, ranging from 0 to 255.
- **int**: value of 16-bit unsigned, ranging from 0 to 65535.
- **object**: instance of a class defined in the program. In it you can find other short, int data and objects of other classes.
- **string**: set of short ASCII values of the different characters that make up the chain, finished with a value more with code 0.

Short and int values may be used equivalently, but considering that if assigned or passed as a parameter an element of type int to an element type short only the low byte of the int value is saved.

When an overflow in an operation for a short or int value occurs, the resultant value is truncated to the last 8 or 16 bits of the result respectively.

Tables

You can define one or more dimensions tables of the types short, int and object data. It can be done in two ways:

- Defining dimensions, indicating after the table name in brackets the number of elements in each dimension. Example:

```
short table [3][4]; // Table with 12 items short
```

- For one dimension tables, declaring the values of the elements, without specifying the number of elements. This form is only valid for static tables. Examples:

```
static short exampleText[] = "Example"; // Table with 8 elements short, including the
       indicator character to end string
static int values[] = { 3, 8, 25, 2, -5, 3 }; // Table with 6 elements int
```

To access the elements of a table and obtain or assign its value can be done in two ways:

- Indicating the name of the table and indexes that identify the element you want to access. Example:

```
short screen[32][25];
short i = 3;
n = screen[i][6]; // Sets to n the value of elment of indexes 3, 6
```

- Declaring a variable type int to use as a pointer and using operators of memory reference. When the pointer should point to the next or previous item must increase or decrease respectively the pointer variable the number of bytes of an item in the table, which can be obtained with the @ operator if not known. Example:

```
int table[10];
int pointer = &table;
n = **pointer; // Assign to n the first element table
n = **(pointer + (5 * 2)); // Assign to n the element of index 5 of table without changing
       the pointer value
pointer += table [0]; // Pointer indicates the second table element; since the size of the
       element is known, better do pointer += 2
```

If you want pass to a function a table as parameter, function must declare the corresponding argument identical to the declaration of the table is given as a parameter. If you want to pass the start address of the table and use that parameter in the function as a pointer to the address table must be passed with the & operator. Examples:

```
void function(short table[5]) { … }
void function2(int tablePointer) { … }
…
short table[5];
function(table); // Sending parameter as table
function2(&table); // Sending parameter as a pointer to table
```

The tables are passed as a parameter by reference, so any changes to the table within the function that will receive is made on the original table.

Strings

The string type is implemented in ccZ80++ as the list of all characters that are more a character with code 0 at the end, end of string indicator.

A character string is specified in the program as the characters that are enclosed in double quotes, no indicator end character string, which is automatically added. When a string is used in an expression is being used the numerical value of the address in memory where it is stored.

Keep in mind that if the same string in several places in a program only stored only once in memory, so that if a change of any character is used is specified it is changing the string for all occurrences that have in the program. You must make a copy of the string in another memory location to avoid this.

To store a string the best way is to declare a one-dimensional table of short values with a number of items equal to or greater than the number of characters you can have up the chain, plus one for the end of string character.

If you want to implement a table of strings could be done by declaring a two-dimensional table of short values, with the first dimension equal to the number of strings to store and a second dimension equal to the maximum number of characters that may have the longest chain of that will store, plus one for the end of string character.

You can also declare the table of elements short and initialize with the codes of the characters in the string of characters you want to store in it.

Examples:

```
short name[11]; // For strings up to 10 characters long
short names[5][21]; // For 5 strings up to 20 characters long
Short name = "Anna"; // Table with 5 elements short
Short name = {'A', 'n', 'n', 'a', '\ 0'}; // Same definition as in the previous example
```

To manage strings not sentences exist natively in ccZ80++. You need to use functions for managing such data. In the download section you can get the String that contains a set of functions for using strings.

To get the memory address of a string stored in a table of short elements can be done with the & operator applied to the short table that stores values. Example: &name for a declaration short name[11].

If it is a string list would have to get the address of the first character of the string you want to know your address in memory. Example: &names[4][0] for a declaration short names[5] [21], thus obtaining the address of the last string in the list.

In an expression can treat constant strings, between double quotes, with the following operators: ==, !=,>,> =, <, <= and +. Comparisons are made in the alphabetical order of the chains.

Expressions

There are two types of expressions:

- **Constant expressions,** which should produce a constant numeric or string value. They can only use constant values and a subset of operators supported by the language. A constant expression can consist of a single constant, numeric or string value.

  These expressions can be used at points in the program where a constant value is necessary.

  Operators valid for a constant expression are:

  () Parentheses to override the precedence of operations in the expression
  & And bitwise
  && And for logical conditions
  * Product
  ^ Xor bitwise
  : False separator of ternary operator
  . Specifier class member (for public and static members only)
  == Equality comparison for number and string
  ! Not for logical conditions
  != Comparison not equal to number and string
  > Comparison greater for number and string
  >= Comparison greater than or equal to number and string
  >> Right shift bitwise
  < Comparison smaller for number and string
  <= Comparison less than or equal to number and string
  << Left shift bitwise
  - Subtraction and negative specification
  % Module
  + Suma, specifying positive value and string concatenation
  ? True separator of ternary operator
  / Division
  ~ Not bitwise
  | Or bitwise
  || Or for logical conditions

  The operator precedence, from highest to lowest, is:

  1. Parentheses: ( )

2. Specifier class member: .
3. Unary pperators: -, !, ~, +
4. Multiplication, division and modulus: *, /, %
5. Sum, concatenation and subtraction: +, -
6. Displacement: <<, >>
7. Comparison: <, <=, >, >=
8. Equality comparison: ==, !=
9. And bitwise: &
10. Xor bitwise: ^
11. Or bitwise: |
12. And for logical conditions: &&
13. Or for logical conditions: ||
14. Ternary operator: ?, :

Examples:

```
(3 + 5) / 4 // An expression that produces a value of 2
"Hello " + name + "." // name is a constant defined with a string value
25 * Constants.width // width is a numerical constant public in class Constants
10 // expression that produces the value 10
```

- **General expressions**, Which must produce a numeric value. They can use constant values, arguments, variables and results returned by functions. An expression could be formed by a single operand constant argument, variable or a function call.

  These expressions can be used within the code of the functions along with the sentences, where a constant value is not strictly necessary.

  Valid operators are:

()	Parentheses to override the precedence of operations in the expression
&	And bitwise and direction of an element
&&	And logical conditions
&=	Auto and bitwise
*	Product and reference to values in memory of type short
**	Reference to values in memory of type int
*=	Auto product
@	Size
^	Xor bitwise
^=	Auto xor bitwise
:	False separator of ternary operator
.	Specifier class or object member
=	Assignation
==	Equality comparison for number and string
!	Not for logical conditions
!=	Comparison not equal to number and string
>	Comparison greater for number and string
>=	Comparison greater than or equal to number and string
>>	Right shift bitwise
=>>	Auto right shift bitwise
<	Comparison smaller for number and string
<=	Comparison less than or equal to number and string
<<	Left shift bitwise
=<<	Auto shift left bitwise
-	Subtraction and negative specification
-=	Auto subtraction
--	Auto decrement
%	Module
%=	Auto module

+	Sum, specifying positive value and string concatenation
+=	Auto sum
++	Auto increment
?	True separator of ternary operator
/	Division
/=	Auto division
~	Not bitwise
|	Or bitwise
|=	Auto or bitwise
||	Or for logical conditions

The operator precedence, from highest to lowest, is:

1. Parentheses: ( )
2. Specifier class or object member: .
3. Autoincrement, autodecrement: ++, --
4. Unary operators: -, !, &, *, **, @, ~, +
5. Multiplication, division and modulus: *, /, %
6. Sum, concatenation and subtraction: +, -
7. Displacement: <<, >>
8. Comparison: <, <=, >, >=
9. Equality comparison: ==, !=
10. And bitwise: &
11. Xor bitwise: ^
12. Or bitwise: |
13. And for logical conditions: &&
14. Or for logical conditions: ||
15. Ternary operator: ?, :
16. Autoasignations: %=, &=, *=, /=, ^=, |=, +=, <<=, =, -=, >>=

Examples:

```
a = b / 3;
function(a + 3, function2(3, 5));
n = (v++) + objectShip.coordinateX();
```

Operators

Most operators have the same function, syntax and precedence in other languages like C ++. Some new or differences from other languages operators:

- Reference to values of type short in memory: *. This operator in unary form allows access to a short type value in memory indicated by its operand. With this you can get and set a value of type short for a memory location.

```
*address
```

Sample expressions:

```
*1000 = 10; // Store the value 10 in memory address 1000
*n = 10; // Store the value 10 in memory address n
v = *n; // Assign to v the value of memory address n
```

- Reference to values of type int in memory: **. This unary type operador allows access to a value of type int indicated by its operand. The value is stored in the address specified by the operand, the low byte of the value, and the next, the high byte of value. With this you can get and set a value of type int for a memory location.

```
**address
```

Sample expressions:

```
**1000 = #ABCD; // Store the hexadecimal value #ABCD in memory at addresses 1000 (low
       byte) and 1001 (high byte)
**n = 2014; // Store the value 2014 in memory at addresses n (low byte) and n + 1 (high
       byte)
v = **n; // Assign to v the value in memory of addresses n (low byte) and n + 1 (high
       byte)
```

- Size: @. This unary type operador giving the size in bytes in memory of parameter or variable given as operand. It may be useful to increase int variables that act as pointers to point to the next or previous item in a table.

```
@element
```

Sample expressions:

```
p += @table[i]; // Increase p the number of bytes of element of index i of table
screen.value(@table); // Call the member function valud of object screen passing as
       parameter size in bytes of table
```

- Address of an item: &. This unary operator can get the memory address of an argument or variable of class or function, of an item in a table and also of a function.

  In the case of a function must simulate their call, which really is not done, with dummy parameters with any value, but of type that need to receive the function, so the compiler knows exactly which function you want to get your address if is overloaded. Examples:

```
n = &process(0, 0); // Assign to n the address of function process declared with two
       numeric arguments, short or int indifferently
n = &process(); // Assign to n then address of function process declared with no arguments
```

- Specifier class or object member: . (dot operator). This binary operator can access public members of a class or object. In the code of a function can directly access the constants, variables and member functions of its containing class, but if you want to access other class members you must use the dot operator, with the following rules:

  - You can not access a private member.
  - If the member is a constant or is static can be accessed through the class name or through an instantiated class object.
  - If the member is a constant and is not static must be accessed through an instantiated class object.

```
classNameOrObjectName.member
```

Sample expressions:

```
Game.maxScore; // Game is a class and maxScore is a public static member
game.lives; // game is an object and lives a public member
game.start(5); // game is an object and start a public member function
```

- Ternary operator: ? :. Evaluate the expression for the first operand, and if it is other than 0 is considered true and the operator returns the second operand, or if 0 it is considered false and the operator returns the third operand.

  In ccZ80++ are always evaluated three operands before evaluating if the first operand is true or false, so that the terms of the second and third operand is evaluated, modifying elements if they contain assignment operators or modification, and calling functions if they contain call someone.

Any operator, in addition to the corresponding operation produces a result. This result can be considered for use in the expression and construct expressions with multiple operations. For example, the + operator for numerical values perform the sum and produces a numeric result, which can be assigned to a variable, leaving the expression as result = value1 + value2. Another example may be less trivial if ((n = v) == 5) … in which is assigned the v value and the output produced, which is the first operand n, is compared to 5.

For know which result produces each operator can noted the following:

- The operators performing an arithmetic, bitwise or shift operation produce the result of that operation.
- The operators performing a logical operation for conditions or a comparison operation produces a value 0 if the operation is false or 1 if the operation is true.
- Assignment operators produce as the result the operand that receives the assignment.
- The operators of auto increment and auto decrement produce the value of the affecting operating, before or after the increase or decrease, depending on whether the operator is indicated before or after the operand.
- Operators of referente to memory values produce the value of the memory location given by the operand.
- The size operator produces the value for the size in memory of the operand.
- The ternary operator yields the result of the expression of the true part of the expression or the false part.
- The specifier operator class member or object produces the value for the member indicated as second operand.

In ccZ80++ all operators are left associative. This is important for the assignment operators because are evaluated first assignment more left. For example in the expression n1 = n2 = n3 compilation error occurs because takes first n1 = n2 and n3 assigned that outcome; to allow this expression had to be written: n1 = (n2 = n3).

Constant values for expressions

When in an expression is needed to indicate a constant value, the following types are supported:

- Decimal numbers: expressed with digits 0 through 9. Example: 123.
- Hexadecimal numbers: expressed with the prefix # and the digits 0 through 9 and the letters A through F, case sensitive indifferently. Example: #1A.
- Characters: represent the numerical value of its ASCII code, and are expressed in single quotes. You can specify escape sequences representing an unprintable character. Examples: 'a', '\ n'.
- Strings: represent a list of characters stored in memory, complete with ASCII character code 0. Are expressed in double quotes. You can specify escape sequences within the list that represent non-printable characters. Example: "hello\n\rbye".

Escape Sequences

To word a special character in a character constant or within a string you can use the following escape sequences:

- \0: represents the character code 0.
- \a: represents the character code 7.
- \b: represents the character code 8.
- \e: represents the character code 27.
- \f: represents the character code 12.
- \n: represents the character code 10.
- \r: represents the character code 13.
- \t: represents the character code 9.
- \v: represents the character code 0.
- \xNN: represents the character code NN, where NN is worded in hexadecimal.

Keywords

The language keywords are:

- **asm**: allows to insert assembler code directly into the program.
- **break**: jumps the process to the next instruction after the loop where it is or ends performing sentences and expressions of a switch sentence.
- **case: i**ndicates a possible value for the expression of a switch sentence to locate below the corresponding sentences and expressions.
- **class**: define a class.
- **const**: define a constant as a member of a class or local of a function.
- **continue**: does jump the process at the end of the current loop.
- **default: i**ndicates the beginning of sentences and expressions to be taken if the value of the expression of a switch sentence does not match any value where noted.
- **do**: set a loop checking completion at the end of each iteration.
- **else**: specify the sentences and expressions to be performed when the expression of an if sentence is false.
- **for**: set a loop with initialization values, verification of completion at the beginning of each iteration and updating values at end at end of each iteration.
- **if**: set a condition to perform some sentences and expressions or not depending on the result of the expression corresponding to the condition.
- **include**: allows to include in the program a source file at the point where this sentence is encountered during the compilation process.
- **inline: i**ndicates that the code of assembler type function is inserted at the point where you call rather than call and return to the calling point.
- **int**: declares arguments and variables of class and function, simple and table, of elements of type int. Also define a function that returns a value of type int.
- **public: i**ndicates that a constant, variable or class member function can be used from outside the class.
- **register: i**ndicates that the parameter of a function type assembly is in a register rather than inserted into the stack.
- **repeat**: defines a loop for a fixed number of times it is repeated.
- **return**: exit a function, returning a value as a result of the function if necessary.
- **short**: declares arguments and variables of class and function, simple and table, of elements of type short. Also define a function that returns a value of type short.
- **static: i**ndicates that a variable or class member function or a local variable of function are common to all object instances that are created for the wrapper class.
- **switch**: performs a group of several sentences and expressions depending on the value that produces an expression.
- **void**: specify that a function returns no value.
- **while**: defines a loop checking at the top end of each iteration and also complete a loop checking for completion at the end of each iteration.

You can not declare any kind, constant, variable or function with the same name as any of the keywords.

Identifiers

Identifiers allow naming classes, constants, variables, functions and arguments.

The identifiers of the program should begin with a letter and contain uppercase letters, lowercase letters, digits 0 through 9 and the underscore character _. The valid characters are only 26 English alphabet and accented letters are not allowed.

They may have an indefinite length and are sensitive to the use of case-sensitive.

An identifier can not be repeated with others of their same level or the upper level. This rule can be broken down as follows:

- An identifier of a class can not be repeated with another class, as they are at the same level

- A constant identifier class member can not be repeated with another member of the same class as another class constant, class or function variable, because they are of the same level; nor with the identifier of a class as it is of a higher level
- An argument identifier, local constant of function or local variable of function can not be repeated with another argument, local constant or variable local of the same function, they are at the same level; nor with identifiers of members of its containing class or class identifiers, because they are all top-level.
- A constant or variable identifier defined within a block of a function, which can be a block sentence or a nested block, can not be repeated with any identifier within the same block or any other block containing at present, to main block of the wrapper function, nor with the wrapper class identifiers of the function or class identifiers, because they are all top-level in the program hierarchy.
- An identifier can be repeated if it is declared in a block that is not in line up in the hierarchy of the program, such as a different function, or in a different block to the current that is not nested.
- As an exception to the rule, an identifier can be repeated in a block of upper level if is defined after is closed this block, and not before this block.

Using objects

To create an object in the program must first define the class to which belong the object using the class sentence.

An object can be created as a class member or as a local member of a class. In both cases, it is declared as a variable of short type or int type but as indicating the name of the class. You can also declare object tables just as tables of items short or int. Examples:

```
Game game; // Declares a object game of class Game
Real numbers[10]; // Declares a table with 10 objects of class Real
```

You can access public members of the object with the dot operator, not private members. If a member is static can also be accessed through the class name as well as through the object name.

Since the code of the member functions of the class for the object can be accessed from any public or private member. If a function declares a object of the same wrapper class can also access any public or private member of that object.

A function can not return a result object, but may receive as parameter an object or an object table, and it must declare the corresponding argument identical to the declaration of the object or object table form. Example:

```
void function(Score argument) …
…
Score s;
function(s);
```

The objects and object tables are passed as a parameter by reference, so any change in the object or some element of the table within the function that receives it, will be done on the original object.

Builders

In ccZ80++ no constructors and destructors as known in other languages. To solve this deficiency exists the possibility of declare in the class following function:

```
public void init()
{
  …
}
```

This function is not callable explicitly from your code.

If a non-static local object is declared in a function, it is not valid for objects declared as class members, the init function will automatically called at the point of the declaration and as many times as the program flow passes through the declaration.

Thus, in this function can be assigned and perform checks to give consistent values to variables or class members and do other actions of object initialization operations.

If the class is derived from another, string of calls to init functions for classes you have defined, from the deepest base class to the base class of the object being declared is created.

If the declaration is a table of objects rather than a single object, a call in the same manner for each of the objects in the table is performed.

If the object or table are declared static call is not made to the init function. A static object can only access class variables declared as static, so the init function is not needed, since the values of static variables can be initialized in its declaration.

This object

All functions declared as static receive implicitly one parameter this corresponding to wrapped object of the function when it is called. With this parameter can be, for example, pass the container object for the current function to another function; also optionally access members of the object, although it is not necessary to use this, only with the identifier can be referenced those members.

Structure of a ccZ80++ program

A ccZ80++ program consists of one or more files. Each file can define one or more classes so not nested. Within each class its constants, variables and member functions are declared. In each function arguments, local variables and code that specifies what operations performs the function defined.

At any point in a file, class or function can indicate comments, preceded by double slash // to the end of line, to document the definitions or the process.

Structure of a file

```
include file

class name …
{
  …
}
```

The include statement can be repeated as often as necessary, one for each file to include in the program. The class sentence could be repeated as often as necessary, one for each class you want to define. These statements can be in the order needed.

Structure of a class

```
class className [: classBase]
{
  constant member definitions
  member variables definitions
  member functions definitions
}
```

A class can optionally extend another classBase which must be previously defined. The effect is that all members of the base class become part of the derived class with the same kind of access private or public that in base class. Equivalent to copy the contents of the base class at begin of the derived class.

With inheritance, along with a better structure of the program and save writing code, you get a smaller size of the final machine code, since a single instance of the code of the member functions of the base class is generated for all derived classes that inherit.

Inside a class constants, variables and functions can be declared in the order needed.

Declaration of constants in a class

A constant allow to define an identifier that will be associated with a numeric or string value that can later be used in any expression as many times as you like in the same way you would use directly the numeric or string value and with the same manner of use within an expression.

```
[public] const name = constantExpression, name2 = constantExpresion2 name2, …;
```

You can declare as many constants as you like with a const sentence. Each constantExpression is a numeric or string value or a constant expression that produces one of these types of result value.

The optional public modifier makes the constant member can be referenced from objects that do not belong to the class through an object of type class being defined and also through the class name if you also have the static modifier . If not specified public, constant is considered private.

The names of the constants should not coincide with any other constant, variable or function within the class or the name of any class defined in the program.

Examples:

```
const max = 100, title = "Points";
public const initialValue = 0;
```

Declaration of variables in a class

A variable can be numeric or object of a class type defined in the program. As a member of a class can not declare objects of one's class or a class that has not yet been defined in order to compile the program. This is motivated because at the time of that declaration the size of the class itself or even undefined class is yet known.

```
[public] [static] type name, name2, …;
```

You can declare as many variables as you want the same type with the same sentence. The type can be:

- **short**: to declare a variable of type short.
- **int**: to declare a variable of type int.
- **class**: to declare a variable of the specified object class.

If the variable is of type short or int and indicates the static modifier can be specified after the variable name equal sign and a constant expression whose value is assigned to the variable as initial value. Example:

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

Each variable can be a simple type if none is specified, or table type if each of its dimensions are shown below as a constant expression in brackets. Example:

```
short table[5][10 * 2]; // Table with 100 items short, 5 x 20
```

If the static modifier indicates you can declare a variable of type table in a single dimension of items short or int indicating only the brackets, without specifying the number of elements, and then the equal sign and braces a set of constant expressions separated by commas. This table is initialized with the given values. Example:

```
static int table[] = { 1, 17 * 3, 45 }; // Table with 3 elements int
```

Also with the static modifier can declare a table of type short only a single dimension indicating the brackets, without specifying the number of elements, and then the equal sign and a string. This declares a short table with a number of elements equal to the number of characters in the specified string plus one for the character code 0 that terminates automatically added. This table is initialized with the ASCII codes of the characters in the string plus a value 0.

```
static short text[] = "hello"; // Table with 6 elements short
```

The public modifier makes the member variable can be referenced from objects that do not belong to the class through an object of type class being defined and also through the class name if you also have the static modifier. If not specified public variable is considered private.

The static modifier causes the variable to be common to all objects of the type of the class being defined. Also allows is variable accessible from outside the class using the class name if also has the public modifier.

The variable names should not coincide with any other constant, variable or function within the class or the name of any class defined in the program.

Other examples:

```
short n1, n2, tableN[10]; // Variables short n1, n2 and table tableN with 10 elements
       short
public int table2[4][3][2]; // Table table2 with 24 elements int
Real reals[5], aux; // Table reals with 5 elements objects of class Real and variable
       object aux class of class Real
```

Definition of functions

Within a function is where a program performs operations that are defined.

```
[public] [static] resultType name(argumentType argumentName, argumentType2 argumentName2,
       …)
{
  sentences and sentence blocks
  expressions
  nested blocks
}
```

Return value resultTypeTipoResultado can be short or int.

Parameter types argumentType can be short, int, object, short table, int table and object table. The name of the argument argumentName is a identifier to access the argument within the function.

The names of the arguments of the function must be unique within the list of arguments to the function being defined and does not match the name of any constant, variable or member function of the containing class, and with no constant or local variable of function, or any class of the program.

You can overload a function if declared with a different set of arguments in number, type or order. It is considered equivalent argument type short or int. An argument of type table is considered different if has different dimensions. Example:

```
void function(short a, int b) { …  }
int function(int x, int y) { … } // Error, this function is equivalent to the previous
void function(short a, int b, int c) { … } // This function is not equivalent
```

When the call is made to a function must be passed as parameters values that match the arguments of his sentence with the following correspondence:

- Short or int argument: parameter can be given as a short variable, an int variable, a constant value in decimal or hexadecimal numeric base, a string or an expression that produces a numeric value. For arguments short, if a value that exceeds the capacity of the short type precision is lost by taking only the low byte of the given value.
- Object argument: must be passed an object of the same type.
- Table argument short: should be passed a short table with the same dimensions and as in the declaration of the argument.
- Table argument int: should be passed a int table with the same dimensions as in the declaration of the argument.
- Argument object array: must be passed a table of objects of the same type and with the same dimensions as in the declaration of the argument.

A short or int argument is received as a copy of the original value, while an argument of type object or table of any type of item is received by reference to the original item specified as a parameter.

The public modifier makes the member function can be referenced from objects that do not belong to the class through an object of type class being defined and also through the class name if you also have the static modifier. If not indicated public function is considered private.

The static modifier makes the function accessible from outside the class using the class name if you also have the public modifier. A static function can not access members of the class to which it belongs, and it is not generated for the this parameter, you can only use parameters it receives and its constants and local variables. If a function is not going to use members of its containing class can be declared as static to improve program performance.

Within a function may exist the following types of blocks:

- Main block of the function, delimited by opening and closing braces of function.
- Blocks corresponding to any sentence blocks, delimited by opening and closing braces of block after a sentence if, repeat, do, while, or for.
- Nested blocks anywhere in the function.

Examples of blocks:

```
void function()
{ // Begin of main block of the function
  …
  if (a == 0)
  { // Begin of block of the if sentence
    …
    { // Begin of nested block within the block of the if sentence
      …
    } // End of nested block within the block of the if sentence
    …
  } // End of block of the if sentence
  …
  { // Begin of nested block within the main block
    …
  } // End of nested block within the main block
} // End of main block of the function
```

Within each block can be specified sentences and declare constants and variables, whose visibility will be local within the block, but whose name does not match that of any constant, variable, function argument, or the function of class, or with any class of program. The visibility of these constants and variables is only within the block where are declared and within other blocks that are inside.

Definition of constants and variables in a function

The same rules apply as for the definition of constants and variables in a class, with the following differences:

- Variables of type short int or simple, no tables, can be initialized with an expression if after its name is the equal sign and the expression shown. This expression will be evaluated at the point of variable declaration and as often as the flow passes through this point program. Example:

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

- The names of constants and local variables of a function should not be repeated with the name of any function argument, nor with any other constant or variable of the same function, besides not repeated with any constant, variable or function defined in wrapper class, or with any class of program.
- The exception to the previous point is that you can declare another variable or constant with the same name as a constant or variable if one has been defined within a function block and the other is defined in another function block or container but then at the end of first block block. Example:

```
…
short a = 0;
if (a! = 0)
{
  …
  short a = 1; // Error: duplicate name
  short b;
  …
}
else
{
  …
  short b; // Correct: duplicate name but in another block
  …
}

short b; // Correct: duplicate name but after closing the block where the first
       declaration is
…
```

- The public modifier is not valid for defining constants and local variables of a function.

- A variable must be declared before being used within the code of the function, but may be at any point of the function, not necessarily at the beginning.

- For a function you can declare a variable object of the wrapper class itself or another not yet defined class type.

Sentences

Besides constant definitions and variables a function may contain sentences that perform operations themselves function. Sentences may appear in the main function block, in a block sentence or a nested block.

Sentences that ccZ80++ supports are:

- **asm**: allows to insert assembler code directly.

```
asm assembler statement
```

or

```
asm
{
  assembler statements
}
```

To create a more structured source code is recommended encapsulate the assembler code in assembler type functions. This is not too penalizes speed and generated machine code space, and even if the function is declared inline type if possible, there is no difference as to whether the assembly code is inserted with the asm statement.

Examples:

```
asm call 5000

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

- **if**: executes the code given if the condition specified by the given expression is true. If the expression produces a value 0 is not true, if it causes any other value is considered to be fulfilled.

The else part is done if the condition is not met, and is optional.

```
if (expression)
  sentence, expression or block
[else
  sentence, expression or block]
```

Example:

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

- **switch**: assessment a value and takes sentences and expressions for the value.

```
switch (expression)
{
  case constantExpression:
    sentences and expressions
    [break;]
  case constantExpression2:
    sentences and expressions
    [break;]
  …
  [default:]
    sentences and expressions
}
```

Sentences and expressions following the keyword case with the value of the constant expression that matches the expression are performed. If no value of all constant expressions given case coincides with, the sentences expressions expressions that follow default are performed, if specified, or otherwise any sentence or expression in the switch sentence is made.

If the break sentence is not specified, continue to make sentences and expressions of the next block case.

For constant expressions following case the ternary operator is not allowed, but any other operator admitted in constant expressions is allowed.

In the sentences and expressions that follow each case or default keyword no constant or variable declarations allowed, unless a nested block is opened to include the necessary declarations and sentences and expressions that correspond. However, you can not use break within this block, but you have to close and off to indicate the break sentence.

Example:

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

- **repeat**: repeat one or more sentences or expressions the number of times indicated.

```
repeat (expression)
  [sentence, expression or block
    …
    [continue;]
    …
    [break;]
  …]
```

The sentences, expressions or block are optional and can be omitted indicating only a semicolon instead.

The continue sentence causes the program control to skip all the sentences of current iteration. The break sentence exits the loop and passes the following sentence after the loop.

Example:

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

- **do**: repeat one or more sentences or expressions while the condition is true, that is evaluated at the end. If the expression produces a value 0 is not true, if it causes any other value is considered to be fulfilled.

```
do
  [sentence, expression or block
    …
    [continue;]
    …
    [break;]
  …]
while (expression);
```

The sentences, expressions or block are optional and can be omitted.

The continue sentence causes the program control to skip all the sentences to evaluation of the expression. The break sentence exits the loop and passes the following sentence after the loop.

Example:

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

- **while**: repeat one or more sentences or expressions while the condition is true, that is evaluated at the beginning. If the expression produces a value 0 is not true, if it causes any other value is considered to be fulfilled.

```
while (expression)
  [sentence, expression or block
    …
    [continue;]
    …
    [break;]
    …]
```

The sentences, expressions or block are optional and can be omitted indicating only a semicolon instead.

The continue sentence causes the program control to skip all the sentences and return to the beginning of the evaluation of the expression. The break sentence exits the loop and passes the following sentence after the loop.

Example:

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

- **for**: initialized values at begin, repeat one or more sentences or expressions while a condition is true, evaluated at the beginning and updated values at the end.

```
for ([initialization expressions; [condition expression]; [expressions update])
  [sentence, expression or block
  …
  [continue;]
  …
  [break;]
  …]
```

The three groups of expressions are optional and you can skip any of them indicating only the semicolon that separates the rest.

The sentences, expressions or block are optional and can be omitted indicating only a semicolon instead.

As initialization expressions can enter one or several sentences separated by commas that will be made before the loop repeats.

If the condition expression yields a value 0 is not true, if it causes any other value is considered to be fulfilled.

As updating expressions can enter one or several expressions separated by commas that will be held at the end of sentences and expressions of the loop again before evaluating the condition expression and a new repetition if this is true.

The continue sentence causes the program control to skip all the sentencias and go to make the final for updating expressions. The break sentence exits the loop and passes the following sentence after the loop.

Example:

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

- **continue**: takes control at the end of the current loop.

```
continue;
```

- **break: e**xits the current loop or end the embodiment of the switch sentence.

```
break;
```

- **return**: exit a function and returns the value corresponding to the result if the function type is not void.

```
return [expression];
```

If the function returns where returns short or int the expression is mandatory. If the function is of type void any term not indicated.

Examples:

```
return; // For function void
return 0; // For function short or int
return function(3, 2); // For function short or int
```

In addition to these sentences, you can specify them with expressions that perform assignments and function calls. Expressions must end with a semicolon, whether they are independently as if they are part of the action of a sentence. Examples:

```
a = function(3, 5) + Screen.currentPositionAddress() * 2;
++n;
Utility.Evaluate(a == 5, "right", "error");
scrore.ChangeValue(game.getScore());
```

Using assembler

Within the code of a function you can use the asm sentence to include assembly code directly.

However, the source code is more structured, and not less efficient, if to indicate operations directly in assembler, assembler functions are created that may be called from code in a general function.

To define a function type assembly is within the class where it is needed with the following syntax:

```
[public] [static] resultType name(argumentType argumentName, argumentType argumentName2,
       …) asm [register|inline] [(supportFunction, supportFuncion2, …)]
{
  assembler statements
}
```

The return value resultType and argument types argumentType can be the same type and with the same restrictions as a general function. The name of the arguments is not significant, since within the function does not be referenced by name, but by its offset within the stack.

An assembler type function can be overloaded with other also of assembler type also or with other of general type.

The parameters passed to a assembler function with respect to the type indicated in his sentence argument must meet the same rules as for a general function.

The parameters in a function type assembly are received by value or by reference in the same way as a general function.

Public and static modifiers have the same effect on a general function.

You can specify one of the two modifiers register or inline with the following effect:

- **register**: only applies to functions that receive a parameter. It makes the parameter value reaches the function in register A if it is an argument of type short or HL register if it is an argument type int or object or table, that in these last two cases will receive its address in memory.
- **inline**: only applies to functions that receive one or no parameters and not declare labels within. The parameter value, if any, is received as a function with the register modifier. The inline switch causes the function code is inserted directly into the point where it is called, instead of a call and a return to the calling point made; ie, is equivalent to inserting its code in asm sentence. A function with this modifier should not have a ret instruction to complete and return to the point from which it was called, just when you have made your last sentence will continue with the program.

After the asm keyword to be specified after the arguments you can specify a list of support functions supportFunction to be assembler type.

These functions can be one's class or other classes, and in the latter case should be referred to them by the class name and function name using the dot operator, so it must be public and static and be contained in a class that has been previously defined in the build order program.

To specify a support function, class or of another, after his name is noted simulated parameters, regardless of its value but the type, since it is not actually performs the call to the function, so that the compiler can determine which version of the function if it is overloaded. In the following example, an assembly type uses other three as support functions, the first one of the class itself and receiving a short or int numeric parameter, the second of class Tools receives two numeric parameters, and the third in the class Main that does not receive any parameters:

```
short function() asm (printString(0), Tools.add(0, 0), Main.process())
{
  …
}
```

The code, labels and memory areas of all support functions can be used from this type assembler function without repeating. With this you can call code of these support functions and use the contents in the memory areas reserved for them values.

The contents of a function type are only Z80 assembler sentences, including labels and directives if necessary.

If the assembler function type has indicated a value of void return, to return the corresponding result must be loaded into a register the return value and exit the function, either with the ret instruction or load this being the last instruction function if it has the inline modifier. The register to load before the end of the function is:

- If the function returns a short type, register A.
- If the function returns an int type, register HL.

Access parameters

We must distinguish whether access to parameters from assembler is a general function with instructions in a block of the asm sentence or from an assembler function type.

In both cases the function parameters are on the stack, but with different ways to get the address where they are. Also in both cases according to the parameter type a value in the stack with a different meaning will have to access it:

- Short and int parameters: in the stack is a copy of its value.
- Object parameters: in the stack is the address of the memory area where his data is located. The data of an object is not static member variables defined in its class, and are in the same order in which they are declared in the program and in consecutive positions, occupying size needed for each data; ie 1 byte for a short variable 2 bytes for an int variable, the size of the object to for an object variable, and the product of the number of items in the table by the size of one of its elements for tables.
- Parameters table of short, int or object: in the stack is the address of the first item; the other elements are in consecutive positions.

For objects size is the sum of the number of bytes of each of the non-static member variables. Static variables of an object are in a different memory area and to access them would need to know his internal label, and this would require generating the assembly code of the program with the compiling option /out=assembler and examine the assembler code to find it, although it should be noted that the name of identification label vary for each compilation.

Access to parameters of a general function.

If you want to access a parameter of a general function by assembler code included in a asm sentence must perform the following steps to get the memory address where a parameter is:

1. Get the address reference data of current function taking her from label _FunctionDataReferenceStore predefined by the program.
2. Calculate the address of the parameter with the formula: (no. parameters - parameter position) x 2 + 4 + reference address.
3. If the parameter is of type short have to add 1 to the result.

For example, for a function that receives 5 parameters, if we get the value of the fourth parameter of type int:

```
ld hl,(_FunctionDataReferenceStore) ; HL contains address reference data
ld de,6 ; Offset of the fourth parameter int = (5 - 4) x 2 + 4
add hl,de ; HL contains the address fo fourth parameter
ld e,(hl)
inc hl
ld d,(hl); DE contanins the value of the fourth parameter
```

For the same function of the value of the second parameter of type short:

```
ld hl,(_FunctionDataReferenceStore) ; HL contanins address reference data
ld de,11; Offset of the second parameter short = (5 - 2) x 2 + 4 + 1
add hl,of; HL contanins the address of second parameter
ld a, (hl); A contanins the value of second parameter
```

Keep in mind that if the function is not defined as static, in addition to those corresponding to the arguments given in its definition parameters, receives a first parameter that is the address of this, ie the container object of the function.

In the general functions can also access their local variables from your reference. The address where the local variables of a function are can be calculated with the formula: reference address - sum of size of all non-static variables. From this direction are local variables of the function, in the same order as they are declared. Example:

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

  asm
  {
    ld ix,(_FunctionDataReferenceStore) ; IX contains reference address data
    ld,-3 ; Offset for local function variables
    add ix,de ; IX contains the address of member variables of object
    ld a,(ix+0) ; A contains the value of variable a
    ld l,(ix+1)
    ld h,(ix+2) ; HL contains the value of variable hl
    ld c,(ix+3); C contains the value of variable c
  }
}
```

Static variables of a function are in a different memory area and to access them would need to know his internal label, and this would require generating the assembly code of the program with the option of compiling /out=assembler and examine the assembler code to find it, although it should be noted that the name of identification label vary in each compilation.

Access to the parameters of a function type assembly.

Access to a parameter in an assembly type function is from the value of registry value SP with the following formule: SP + (no. parameters - position parameter) x 2 + 2 If the parameter is of short type must add 1 to this result. Example:

```
static void function(short a, int hl) asm
{
  ld ix,0
  add ix,sp ; IX has the value of SP
  ld a,(ix+5) ; A contains the value of parameter a, offset (2 - 1) * 2 + 2 + 1
  ld l,(ix+2)
  ld h,(ix+3) ; HL contains the value of parameter hl, offset (2 - 2) * 2 + 2
}
```

Keep in mind that if the function is not defined as static, in addition to those corresponding to the arguments given in its definition parameters, receives a first parameter that is the address of this, ie the container object of the function.

Assembler syntax

The syntax for writing assembly code for both asm sentence as to the functions of assembler type is based on the DEVPAC GENA assembler package, with the differences and features listed below:

- Accepts all instructions and includes the use of statements that use the registers ixh, ixl, iyh, iyl and undocumented instructions of rotation and shift.

- The declaration of labels and symbols with EQU need colon (:) after the label name.
- The length of name labels and symbols is unlimited.
- Numeric constants can be specified in decimal, hexadecimal with the prefix # and in binary prefixed with %.
- String constants and character constants are delimited by double quotes.
- Do not use escape sequences in strings or character constants.
- Expressions can use numeric constants, character constants, symbols defined with EQU, labels and the operators + (addition), - (subtraction), * (product), / (integer division), ? (module), & (and logical), @ (or logical) and ! (xor logical). You can also use the $ symbol to specify the address of the current instruction.
- Permitted directives are ORG, EQU, DEFB, DEFW, DEFM and DEFS.
- Conditional directives IF, ELSE and END are not accepted.
- Not case sensitive for user names, register names, labels and symbols.
- Assembler commands (*E, *H, *S etc..) are not recognized.
- Line numbers are not allowed.

Restrictions of ccZ80++

The language does not natively uses dynamic memory management, so creating classes is declaring variables of type object, not creating them using the new operator such as in C++. Nor can other features that require the use of dynamic memory, such as the return of and object for a function, but can be sent as a parameter to a function or an object table and manage objects within it, changing its state.

The decision not to use objects in dynamic memory in ccZ80++ is due to penalizing too speed of generated program, as they must create and remove objects from memory and periodically make a release of the memory areas that no longer uses any object active in the program.

Nor is there class polymorphism, ie, allow reference an object as if it were one of its base classes. In ccZ80++ there are no references to objects but variables of type object, so this feature does not apply to language.

Programs generated with ccZ80++ management make no error handling to allow maximum speed code execution.

The ccZ80++ language does not provide pointers natively. To use a pointer, you must declare a variable of type int and assign the address of the element you want to pointer using the & operator, and if it is a short element type, access or assign its value by the * operator, or if an element of type int, using the ** operator. If the pointer points table elements, so that the pointer points to the next or previous element is necessary to add or subtract, respectively, the size of the element, which can be obtained with the @ operator.

No structures exist with that name, but you can declare a class, with or without membership functions with its public member variables, and objects that are declared with that class have exactly the same functionality and use a structure of other languages. The data will be in memory at the address of the object in the same order as declared in its class.

Optimizations

The machine code produced by ccZ80++ making a normal programming is very fast, but if you want more you can optimize with the following tips:

- Using arguments and variables of type short instead of int if the range of values that will be stored int not need, ie not exceeding the value 255. Z80 microprocessor manages faster short values that int values.
- If a class contains member variables which are assigned value without any prior testing and its value is obtained without any treatment of the data, you can avoid creating functions for gettins and setting the value of such members, and declare them as public to be they can be used directly.
- If a function is not recursive, directly or indirectly, can be declared local variables as static. Access to a static variable is faster than a non-static variable.
- If a function is not recursive, directly or indirectly, whether accessed numerous times in the code to an argument of a function, its value can be passed to a local static variable in the function. Access to a static variable is faster than an argument.

- Finally, if for some action top speed is needed, it can be use for her of assembler, preferably encapsulated in assembler type functions for better readability of source code.