HCMC University of Technology
Faculty of Computer Science & Engineering

MC

Micro C Language

Author

Dr. Nguyen Hua Phung

August 2019

Contents

MC Specification

version 1.2

1 Introduction

MC (Micro C) is a language which consists of a subset of C plus some Java language features.
The C features of this language are (details will be discussed later): a few primitive types, one-dimensional arrays, control structures, expressions, compound statements (i.e., blocks) and functions.
The Java features of this language are as follows:

1. MC has type **boolean**, borrowed from Java. In MC, therefore, all boolean expressions must be evaluated to a value of type **boolean**, which is either true or false. (That is, boolean variables have numerical values, like those in C++).

2. In MC, like Java, the operands of an operator are guaranteed to be evaluated in a specific evaluation order, particularly, from left to right. In C and C++, the evaluation order is left unspecified. In the case of *f() + g()*, for example, MC dictates that *f* is always evaluated before *g*.

For simplicity reason:

1. MC does not support variable initialization.

```
float f = 1.0;   // ERROR
float f;         // CORRECT
```

2. In an array declaration, the size of array must be known explicitly.

```
int i[];         // ERROR
int i[5];        // CORRECT
```

Conventionally, the **sequence '\n' must be used as a new line character in MC**.

2 Program Structure

MC does not support separate compilation so all declarations (variable and function) must be resided in one single file.

An MC program consists of many declarations which include variable declarations and function declarations.

2.1 Variable declaration:

- **Not** support variable initialization
- Array size must be known during declaration.
- Syntax: There are two types of variable declaration:
  1. *<primitive type> <variable> ';'*
     where a *<primitive type>* is a primitive type which is described in Section 4; a *<variable>* can be an identifier alone or an identifier followed by a '[', an integer literal, and then a ']'.
     For example: *int i; float j[5];*
  2. *<primitive type> <many-variables> ';'*
     where *<many-variables* is a comma-separated list of *<variable>*.
     For example: *int i,j,k[5];*

2.2 Function declaration:

In MC, every function declaration is also a function definition. That is, a function declaration specifies the name of the function, the type of the return value and the number and types of the arguments that must be supplied in a call to the function as well as the body of the function. A function is declared as follows:
<type> <function-name> '(' <parameter-list> ')' <block-statement>
where

- *<type>* is the function return type which is a primitive type, array pointer type or void type.
- *<function-name>* is an identifier used to represent the name of the function.
- *<parameter-list>* is a nullable comma-separated list of *<parameter-declaration>*'s. A *<parameter-declaration>* is declared as follows:
  <primitive type> <identifier> or *<primitive type> <identifier> '[' ']'*.
- *<block-statement>* is described in Section 7.8

MC does not support function overloading. Thus, a function must be defined exactly once.

MC does not support nested function. For example:

```
void foo(int i) {
        int child_of_foo(float f){...} //ERROR
}
```

3 Lexical Specification

This section describes the character set, comment conventions and token set in the language.

3.1 Character Set

An MC program is a sequence of characters from the ASCII character set. Blank, tab, formfeed (i.e., the ASCII FF), carriage return (i.e., the ASCII CR) and newline (i.e., the ASCII LF) are *whitespace characters*. A line is a sequence of characters that ends up with a LF. This definition of lines can be used to determine the line numbers produced by an MC compiler.

3.2 Comments

There are two kinds of comments:

- A traditional block comment:
  / This is*
  *a block comment */*
  All the text from /* to */ is ignored as designed in C, C++ and Java.

- A line comment:
  //This is a line comment
  All the text from // to the end of the line is ignored.
  As designed in C, C++ and Java, the following rules are also enforced:

  – Comments do not nest.

  – /* and */ have no special meaning in comments that begin with //.

  – // has no special meaning in comments that begin with /*.

3.3 Token Set

In an MC program, there are five categories of tokens: identifiers, keywords, operators, separators and literals.

- Identifiers: An identifier is an unlimited-length sequence of letters, digits and underscores, the first of which must be a letter or underscore. MC is **case-sensitive**, meaning that *abc* and *Abc* are distinct.
- Keywords: The following character sequences are reserved as *keywords* and cannot be used as identifiers:
  boolean break continue else for float if int return
  void do while true false string
- Operators: There are 15 different *operators* in MC:

Operator	Meaning	Operator	Meaning
+	Addition	-	Subtraction or negation
*	Multiplication	/	Division
!	Logical NOT	%	Modulus
||	Logical OR	&&	Logical AND
!=	Not equal	==	Equal
<	Less than	>	Greater than
<=	Less than or equal	>=	Greater than or equal
=	assign		

3.4 Separators

The following characters are the **separators**: left square bracket ('['), right square bracket (']'), left parenthesis ('('), right parenthesis (')'), left bracket ('{'), right bracket ('}'), semicolon (';') and comma (',').

3.5 Literals

A **literal** is a source representation of a value of either an int type, float type, boolean type or string type.

- An *integer literal* is **always expressed in decimal (base 10)**, consisting of a sequence of at least one digit. An integer literal is of type **int**.
- A *floating-point literal* has the following parts: a whole-number part, a decimal point (represented by an ASCII period character), a fractional part and an exponent. The exponent, if present, is indicated by the ASCII letter **e** or **E** followed by an optionally signed (-) integer. At least one digit, in either the whole number or the fraction part,

and either a decimal point or an exponent are required. All other parts are optional.
A floating-point literal is of type **float**.
For example: The following are valid floating literals:
1.2 1. .1 1e2 1.2E-2 1.2e-2 .1E2 9.0 12e8 0.33E-3 128e-42
The following are **not** considered as floating literals:
e-12 (no digit before 'e') 143e (no digits after 'e')

- The *boolean literal* has two values, represented by the literals **true** and **false**, formed from ASCII letters.
- A *string literal* consists of zero or more characters enclosed in double quotes '"'. The quotes are not part of the string, but serve to delimit it.
  It is a compile-time error for a backspace, newline, formfeed, carriage return, tab, double quote or a backslash to appear inside a string literal. The following escape sequences are used instead:

\b	backspace
\f	formfeed
\r	carriage return
\n	newline
\t	horizontal tab
\"	double quote
\\	backslash

  A string literal is of type **string**.

4 Types and Values

Types of all variables and expressions in MC must be known at compile time. Types limit the values that a variable can hold (e.g., an identifier x has type int cannot hold value true...), the values that an expression can produce, and the operations supported on those values (e.g., we cannot apply a plus operator to 2 boolean values...).
MC types are divided into three categories:

- Primitive types: **boolean, int, float, string.**
- Void type.
- Array and Array pointer types.

Function declarations can specify **void** in place of a return value to indicate that the function does not return a value.

4.1 The void Type and Values

The **void** type specifies an empty set of values. It is used only as the type returned by functions that generate no value.

4.2 The boolean Type and Values

The **boolean** type represents a logical quantity with two possible values: *true* and *false*. The following operators can act on boolean values:
== != ! && || =
A *boolean expression* is an expression that evaluates to *true* or *false*. Boolean expressions determine the control flow in **if**, **for** and **do while**. Only boolean expressions can be used in these control flow statements.

4.3 The int Type and Values

The values of type **int** are 32-bit signed integers in the following ranges:
-2147483648 ... 2147483647
The following operators can act on integer values:
+ - * / % < <= > >= == != =
The first five operators always produce a value of type **int**. The next six operators always result a value of type **boolean**. The last operator (=) will result a value of type **int** when both operands are in type **int**. While the operands of operators %, ==, and != must be in the same type, those of the other operators may be in different types.
Here, - represents both the binary subtraction and unary negation operators.

4.4 The float Type and Values

A float value is a 32 bit single-precision number. The exact values of this type are implementation-dependent. The following operators can act on floating-point values:

- The binary arithmetic operators +, -, * and /, which result in a value of type **float**.
- The unary negation operators -, which results in a value of type **float**.
- The relational operators <, <=, > and >=, which result in a value of type **boolean**.
- The assign operator = which results in a value of type **float** if the left hand side operand is in type **float**.

4.5 The string Type and Values

Strings can only be used in an assignment or passed as a parameter of a function invocation. For example, a string can be passed as a parameter to the built-in function *putString()* or *putStringLn()* as described in Section 8.

4.6 Array Types and Their Values

MC supports only **one-dimensional arrays**. Originally, arrays in C support the following features:

- Array subscripts start at 0. If an array has n elements, we say n is the length of the array; the elements of the array are referenced from 0 to n-1.
- A subscript can be any integer expression, i.e., any expression of type **int**.
- The values (or precisely, r-values) of an array type are *pointers* to array objects. In other words, the value of a variable of an array type is the *address* of element zero of the array.

However, for simplicity purpose, one-dimensional arrays in MC are more restrictive :

- The element type of an array can only be a primitive type such as **boolean**, **int**, **float** or **string**.
- The length of an array must be specified by an integer literal in the array declaration (e.g., *int i[5];* is a correct array declaration; while *int i[];* is not).

4.7 Array Pointer Type

Array pointer type is used to declare the input or output parameter type in a function declaration. The value passed to this type must be in an **array** type or an **array pointer** type.

- Input parameter: *<primitive type> <identifier> '[' ']'.*
- Output parameter: *<primitive type> '["]'*

  For example,
  int[] foo(int a, float b[]) {int c[3];...; if (a>0) foo(a-1,b); ...; return c; }

5 Variables

In MC, every variable declaration is also a variable definition. That is, a variable declaration specifies not only the type for the variable but also reserves storage for the variable.
As discussed above, MC **does not support variable initialization**. Thus, the following declaration is incorrect:
int i=0;(correct one must be *int i;*).
All variables must be **declared before used**. Every variable declared in the program must be in primitive, array or array pointer type where array pointer type is applied only to function parameters.
There are two kinds of variables: global and local variables.

5.1 Global Variables:

Global variables are declared outside all functions. These are legal variable declarations:

```
boolean b;        // a variable of type boolean
int i;            // a variable of type int
float f;          // a variable of type float
boolean ba[5];    // a variable of type array on boolean
int ia[3];        // a variable of type array on int
float fa[100];    // a variable of type array on float
```

While these are **illegal**:

```
int i=5;                        //no initialization => int i;
float f[];                      //must have size => float f[5];
boolean boo[2]={true,false};    //no initialization => boolean boo[2];
```

A global variable is created at the program startup and destroyed when the program completes. A global variable is **initialized implicitly** to a default value as follows:

Type	Default Value
boolean	false
int	0
float	0.0
boolean[2]	{false, false}
int[2]	{0, 0}
float[2]	{0.0, 0.0}

5.2 Local Variables:

Local variables are declared as function parameters or inside the body of a function as well as inside a block. For example,

```
int foo(int a, float b[])
```

```
{
    boolean c;
    int i;
    i = a + 3;
    if (i>0) {
        int d;
        d = i + 3;
        putInt(d);
    }
    return i;
}
...
```

In the above example, **a, b, c, i** and **d** are local variables. The scope of these variables will be discussed in Section 9.

A storage of a local variable declared in a block is allocated when the flow of control enters the block and destroyed as soon as the flow of control leaves the block. Unlike a global variable, a local variable may be associated with more than one storage during the execution of the program.

6 Expressions

An expression is a finite combination of operands and operators. An operand of an expression can be a literal, an identifier, an element of an array or a function call.

6.1 Precedence and Associativity

The rules for precedence and associativity of operators are shown as follows:

Operator	Arity	Notation	Precedence	Associativity
[]	unary	postfix	1	none
- !	unary	prefix	2	right to left
/ * %	binary	infix	3	left to right
+ -	binary	infix	4	left to right
< <= > >=	binary	infix	5	none
== !=	binary	infix	6	none
&&	binary	infix	7	left to right
||	binary	infix	8	left to right
=	binary	infix	9	right to left

The operators on the same row have the same precedence and the rows are in order of decreasing precedence. An expression which is in '(' and ')' has highest precedence.

6.2 Type Coercions

In MC, like C and Java, mixed-mode expressions whose operands have different types are permitted in some operators.

The operands of the following operators:
+ - * / < <= > >= =
can have either type **int** or **float**. If one operand is **float**, the compiler will implicitly convert the other to **float**. Therefore, if at least one of the operands of the above binary operators is of type **float**, then the operation is a floating-point operation.

Exceptionaly, for an assignment, when the left hand side is in type **float**, the right hand side can be in type **int** or **float**. In both cases, the return type of the assignment is **float**. When the left hand side is in type **int**, the right hand side is only allowed in type **int** and the return type is **int**.

Assignment coercions occur when the value of an expression is assigned to a variable – the value of the expression is converted to the type of the left side, which is the type of the result.

The following type coercion rules for an assignment are permitted:

- If the type of the left-hand-side (LHS) is **int**, the expression in the right-hand-side (RHS) must be of the type **int** or a compile-time error occurs.
- If the type of the LHS is **float**, the expression in RHS must have either the type **int** or **float** or a compile-time error occurs.
- If the type of the LHS is **boolean** or **string**, the expression in the RHS must be of the same type or a compile-time error occurs.

6.3 Index Expression

An **index operator** is used to reference or extract selected elements of an array. It must take the following form:
<expression> '[' expression ']'
The type of the first *<expression>* must be an array or array pointer type. The second expression, i.e. the one between '[' and ']', must be of integer type. The index operator returns the corresponding element of the array.
For example,
foo(2)[3+x] = a[b[2]] +3;
The above assignment is valid if the return type of foo is an array pointer type whose element type is **int**, x is in **int** type and a and b are in an array type whose element type is **int**.

6.4 Invocation Expression

An invocation expression is a function call which starts with an identifier followed by "(" and ")". A nullable comma-separated list of expressions might be appeared between "(" and ")" as a list of arguments.

Like C, all arguments (including arrays) in MC are passed "by value." The called function is given its value in its parameters. Thus, the called function cannot alter the variable in the calling function in any way.

When a function is invoked, each of its parameters is initialized with the corresponding argument's value passed from the caller function.

A parameter of an array type has exactly the same meaning as that in C:

- When an array variable is passed (as an argument) to a function, the location (i.e., address) of element zero of the array (which, by definition, is the value of the array variable) is passed. Thus, any modifications made by the callee on the array will be visible on the corresponding argument.
- The array length in a parameter declaration is illegal. For example,
  void f(int a[10]) { }
  is illegal.

The type coercion rules for assignment are applied to parameter passing where LHS's are formal parameters and RHS's are arguments. An exception in parameter passing is that when the parameter is in an array pointer type, the corresponding argument must be in an array or array pointer type whose member type must be the same.

For example,

```
void foo(float a[])...
void goo(float x[]) {
    float y[10];
    int z[10];
    foo(x); //CORRECT
    foo(y); //CORRECT
    foo(z); //WRONG
}
```

The type coercion rules and the exception in parameter passing are also applied to return type where LHS is the return type and RHS is the expression in the return statement. An exception is that when the return type is **void**, there must be no expression in the return statement.

For example,

```
void foo() {
    if (...) return; //CORRECT
    else return 2; //WRONG
}
```

and,

```
int[] foo(int b[]) {
    int a[1];
    if () return a; //CORRECT
    else return b; //CORRECT
}
```

6.5 Evaluation Order

MC requires the left-hand operand of a binary operator must be evaluated first before any part of the right-hand operand is evaluated.

Similar, in a function call (called a method call in Java), the actual parameters must be evaluated from left to right.

Every operand of an operator must be evaluated before any part of the operation is performed. The two exceptions are the logical operators && and ||, which are still evaluated from left to right, but it is guaranteed that evaluation will stop as soon as the truth and falsehood is known. This is known as the **short-circuit evaluation**.

7 Statements and Control Flow

MC supports these statements: **if**, **for**, **do**. . . **while**, **break**, **continue**, **return**, expression, and block. All statements except **if**, **for** and the block one must be followed by a semi-colon.

7.1 The if Statement

There are two types of **if** statement: if-else and if-no else. The if-else is written as follows:
if '(' <expression> ')'
    <statement1>
else
    <statement2>
where *<expression>*, which must be of the type **boolean**, is first evaluated. If it is *true*, *<statement1>* is executed. If it is *false*, *<statement2>* is executed.

The if-no else is like if-else but there is no *else* and *<statement2>*. In this type of if statement, if the *<expression>* is *false*, the next statement will be executed.

Like C, C++ and Java, the MC language suffers from the so-called dangling-else problem. MC solve this by decreeing that an else must belong to the innermost if.

7.2 The do while Statement

do <*statement*$_1$> <*statement*$_2$> ... <*statement*$_n$> *while* <*expression*> *';'*
where $n \geq 1$.

When **do**...**while** statement is executed, *statement*$_1$, *statement*$_2$,...,*statement*$_n$ are first executed sequentially. Then, <*expression*>, which must be of the type **boolean**, will be evaluated. If it is *true*, then *statement*$_1$, *statement*$_2$,...,*statement*$_n$ are re-executed and then <*expression*> is re-evaluated. This execution/ evaluation cycle repeats until the <*expression*> becomes *false*.

7.3 The for Statement

The **for** statement is written as follows:
for '(' <*expression*$_1$> *';'* <*expression*$_2$> *';'* <*expression*$_3$> *')'* <*statement*>

where <*expression*$_1$> is executed first to give an **int** value and then <*expression*$_2$> is executed to result an **boolean** value. If the result is *true*, the <*statement*> and then <*expression*$_3$> are executed before the <*expression*$_2$> is re-checked. The type of <*expression*$_3$> must be **int**. If the result of <*expression*$_2$> is *false*, the **for** will stop execute.

7.4 The break Statement

This statement must appear inside a loop such as **for** or **do while**. When it is executed, the control will transfer to the statement next to the enclosed loop. This statement is written as follows:
break ';'

7.5 The continue Statement

This statement must appear inside a loop such as **for** or **do while**. When it is executed, the control will jump to the end of the body of the loop. This statement is written as follows:
continue ';'

7.6 The return Statement

A **return** statement aims at transferring control to the caller of the function that contains it.

A **return** statement with no expression must be contained within a function whose return type is void or a compile-time error occurs.

A **return** statement with an expression must be contained within a non-void function or a compile-time error occurs.

7.7 The expression Statement

An *expression* becomes a *statement* if it is followed by a semicolon. For example, the following four expressions

```
i = 1
foo(1,2)
i + 2
100
```

become statements as follows:

```
i = 1;
foo(1,2);
i + 2;
100;
```

7.8 The block statement

A block statement starts with a '{', followed by a nullable list of variable declaration and statement, and ends up with a '}'.

For example:

```
{
    int a,b,c;   //variable declaration
    a=b=c=5;     // assignment statement
    float f[5]; // variable declaration
    if (a==b) f[0] = 1.0; // if statement
}
```

8 Built-in Functions

MC has some following built-in functions:
int getInt(): reads and returns an integer value from the standard input
void putInt(int i): prints the value of the integer i to the standard output
void putIntLn(int i): same as putInt except that it also prints a newline
float getFloat(): reads and returns a floating-point value from the standard input
void putFloat(float f): prints the value of the float f to the standard output
void putFloatLn(float f): same as putFloat except that it also prints a newline
void putBool(boolean b): prints the value of the boolean b to the standard output
void putBoolLn(boolean b): same as putBoolLn except that it also prints a new line
void putString(string s): prints the value of the string to the standard output

void putStringLn(string s): same as putStringLn except that it also prints a new line
void putLn(): prints a newline to the standard output

9 Scope Rules

Scope rules govern declarations (defining occurrences of identifiers) and their uses (i.e., applied occurrences of identifiers).

The scope of a declaration is the region of the program over which the declaration can be referred to. A declaration is said to be in scope at a point in the program if its scope includes that point.

A block is a language construct that can contain declarations. There are two types of blocks in the MC language:

- The outermost block is the entire program.
- Each block statement forms a block by itself. A special case is that a function has its block from '(' (before the parameter list) to '}' (the end of its body).

MC exhibits nested block structure since blocks may be nested one within another. Therefore, there may be many scope levels:

- All function declarations, built-in or user-defined, and all global variables make up the outermost block at scope level 1, or global scope.
- Variable declarations inside an inner block are local to that block. Every inner block is completely enclosed by another block. If enclosed by the outermost block, that inner block is said to be at scope level 2. If enclosed by a level-2 block, that inner block is at scope level 3, and so on.

There are four additional rules on the scope restrictions:

1. All declarations in global scope are effective in the entire progran.
2. All declarations in local scope are effective from the place of the declaration to the end of its scope.
3. No identifier can be defined more than once in the same scope. This implies that **no** identifier represents both a global variable and a function name simultaneously.
4. *Most closed nested rule*: For every applied occurrence of an identifier in a block, there must be a corresponding declaration, which is in the smallest enclosing block that contains any declaration of that identifier.

Consider the following MC program:

```
1  int i;
2  int f() {
3      return 200;
4  }
5  void main() {
6      int main;
7      main = f();
8      putIntLn(main);
9      {
10         int i;
11         int main;
12         int f;
13         main = f = i = 100;
14         putIntLn(i);
15         putIntLn(main);
16         putIntLn(f);
17     }
18     putIntLn(main);
19 }
```

The above program will be compiled and print the following results:

200
100
100
100
200

In this program, there are three scope levels:

Declaration	Level	Scope
putIntLn (built-in)	1	Entire program
i (line 1)	1	Entire program
f (line 2)	1	Entire program
main (line 5)	1	Entire program
main (line 6)	2	line 6-8 and line 18
i (line 10)	3	line 10-16
main (line 11)	3	line 11-16
f (line 12)	3	line 12-16

The variable *main* declared in line 6 is said to *hide* the function declaration *main* in line 5. The variable *main* declared in line 11 hides the variable declaration *main* in line 6. The variable i declared in line 10 hides the global variable i in line 1. The variable f

declared in line 12 hides the function declaration f in line 2.

The scopes of the declarations f in line 2, i in line 1 and main in line 5 are not contiguous. Such gaps are known as scope holes, where the corresponding declarations are hidden or invisible. As a matter of style, it is advised not to introduce variables that conceal names in an outer scope. This is the major reason why Java disallows a variable declaration from hiding another variable declaration of the same name in an outer scope. Therefore, the MC program above is a bad programming style.

10 The main function

A special function, i.e. main function, is an entry of a MC program where the program starts:

```
void main() { // no parameters are allowed
   ...
}
```