C// programming language

Introduction

C// is a placeholder name for the language specified in this document since I couldn't come up with anything more meaningful. The name implies that a lot of features were "sliced off"C language, hence the double slash.

1 Lexical structure

1.1 Input format

C// compiler takes ASCII encoded text files as the input.

1.2 Keywords

C// features the following keywords:

1. `break`
2. `byte`
3. `continue`
4. `else if`
5. `else`
6. `while`
7. `if`
8. `import string`
9. `int`
10. `return`
11. `void`

All keywords are reserved.

1.3 Identifiers

Identifiers of C// can be described by the following regular expression:

`_*[a-zA-Z][a-zA-Z0-9_]*`

To introduce an example, `my_var` is a valid identifier.

1.4 Comments

C// features multi-line comments which have the same form as they do in C. The comment block begins with `'/*'` and ends with `'*/'`. Single-line comments starting with `'//'` are also supported.

1.5 Whitespace

Whitespace characters work as delimiters of tokens and have no semantic meaning (except some keywords).

1.6 Literals

1.6.1 Character literals

Character literals represent an ASCII character. In the source code, the character is enclosed in single quotes to express the corresponding ASCII character.

1.6.2 Integer literals

Integer literals are sequences of digits representing integer constants. C// only supports decimal radix for integer constants.

1.6.3 String literals

Just like in C, string literals are sequences of ASCII characters enclosed in double quotes to express ASCII strings. A null terminating byte is automatically appended by the compiler.

1.7 Operators

1.7.1 Arithmetic and assignment operators

+	addition
-	subtraction
-	unary minus
*	multiplication
/	integer division
%	integer modulo
=	assignment

All of the above operations can be applied on all types except void. However, types of both operands must match, i.e. addition where one operand is of type byte and the second of type int won't compile. Furthermore, each arithmetic operation must only be applied

on non-array operands. The following snippet demonstrates legal and illegal assignment usages:

```
byte a1[50], a2[60];
byte x[];
/* This is illegal. */
x = a1 + a2;

/* This is OK. */
x[0] = a1[0] + a2[0];
```

Assignment to array variables is supported, however the dimension count of both operands must equal. Therefore, it is not possible to assign a simple variable to an array variable or vice versa. Moreover, when an array is assigned to another array variable, the contents of the array is not copied, but a reference to it is stored in the new variable. Any changes to the underlying values made by any of these variables will propagate to the others. The following code fragment demonstrates this:

```
byte x[50], y[];
/* This is OK. */
x[0] = 'a';
y = x;
y[0] = 'b';
/* Prints 'b'. */
<==x[0];
/* Prints 'b'. */
<==y[0];
/* This is illegal. */
y = 'a';
```

1.7.2 Relational operators

==	equal to
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal to
!=	not equal to

These operators can be applied on operands of type byte or int and their types must match. They return a value of the corresponding type which is either 1 when the condition is met or 0 otherwise. These operations can only be applied on non-array operands. The following snippet demonstrates this:

```
int x = 1 == 1;
/* Prints 1 */
<==x;
x = -1 == 1;
/* Prints 0 */
<==x;
```

1.7.3 Logical operators

&&	logical and
||	logical or
!	logical not

These operators can be applied on operands of type byte or int and their types must match. They return a value of the corresponding type which is either 1 when the condition is met or 0 otherwise. These operations can only be applied on non-array operands.

2 Types

2.1 Numeric types

`byte`	signed byte
`int`	signed 4-byte word

2.2 Booleans

C// does not support any boolean type. However, any integer type can be used for this purpose with `byte` being the most straightforward option.

2.3 Strings

Strings are handled in a C-like way. They are represented as arrays of byte type values. A valid string must be terminated by a notorious null byte. A set of function is provided for string manipulation. The user must specify to import these library functions by writing `import string;` at the beginning of the program. These functions can be overriden by the user, however, import must not be specified in such a case.

```
/* Returns length of str. */
int strlen(byte str[]);

/* Lexicographic comparison.
 * Returns -1, 0, 1 if str1 is less, equal or greater than str2.
 */
int strcmp(byte str1[], byte str2[]);

/* String concatenation. */
void strcat(byte dest[], byte src[]);

/* String copying. */
void strcpy(byte dest[], byte src[]);
```

When using functions `strcat` and `strcpy`, the caller must guarantee that the size of the destination buffer for the result is sufficient, i.e. allocated size for `dest` must be at least:

```
/* + 1 for a null terminating byte */
strlen(str1) + strlen(str2) + 1
```

Further details about arrays such as its declaration and lifetime are described in subsection 4.2.

3 Statements and expressions

C// supports loop, if, break, continue and return statements. Loop and if statements can be nested.

3.1 Loop statements

The only loop construct C// supports is while loop. The syntax is similar to that of C. The expression in the loop header is evaluated and the loop executes the body if the expression is not equal to zero. As opposed to C, a code block belonging to a loop must be enclosed in a pair of curly brackets. Therefore, even a code block containing one line must be enclosed in curly brackets. Lastly, the loop header and loop body must not be empty. The following example demonstrates how to use a for loop:

```
int i = 0;
while (i < 100) {
        /* do something */
        i = i + 1;
}

/* Infinite loop */
while (1) {
        <=="infinite loop";
        <==;
}
```

3.2 Continue statement

Continue statements might come useful inside loops. When executed, the control flow jumps to the expression evaluation inside the loop header resulting in the execution of the next loop iteration or termination.

3.3 Break statements

Executing break causes immediate loop termination.

3.4 If statements

Every if, else if and else statement body must be enclosed in a pair of curly brackets. The body of an if statement is executed if and only if the expression in the header returns a non-zero value. If the condition in the if header does not pass, optional subsequent else if statements are evaluated in the same way. If none of else if conditions passes, optional

else block is executed. The result of a condition expression must always be an integer value of any of the numeric types described in subsection 2.1. If, else if and else header and body must not be empty. A valid if statement might look like this:

```
if (1 > 2) {
        /* This never executes. */
        shutdown();
} else if (2 == 2) {
        /* This always executes. */
        <=="pass";
        <==;
} else {
        /* This never executes. */
        shutdown();
}
```

3.5 Return statement

Return statement terminates the current function execution and returns the control flow to the parent function on the call stack. It is used in non-void functions to return a value. Return keyword must be followed by an expression, therefore it cannot be used in functions with void return type.

3.6 Expressions

An expression returns or assigns a value to a variable after performing computations and applying functions on operands. Expressions are separated with `';'` character when used outside of while/if/else if headers.

3.6.1 Assignment expression

This expression assigns a value to a variable and the value is returned.

3.6.2 Function calls

Function `func` can be invoked with the following expression:

```
func(param_1, param_2, ..., param_k);
```

In case `func` returns a value, it can be assigned to an operand:

```
func_result = func(param_1, param_2, ..., param_k);
```

Functions with variable parameter count are not supported.

3.6.3 Type casts

Since C//only supports numeric primitive types, all variables can be cast. For instance, a `int` value `x` is converted to `byte` value `y` by storing the least significant byte in `y`. The other way around is performed by signed extension. The arrays are assigned by a reference and can be cast as well. However, their dimension counts must match along with their types. In other words, a two dimensional array cannot be cast to a three dimensional. To introduce an example:

```
/* The language does not support hex representation of literals,
 * used only for the demonstration.
 */
 int array[2];
 byte byte_array[];

 array[0] = 0xccff;
 array[1] = 0xccff;
 byte_array = (byte[]) array;
 /* byte_array == {0xff, 0xcc, 0x00, 0x00, 0xff, 0xcc, 0x00, 0x00} */
```

Each expression has a strict type including literals which is checked at compile-time. Implicit type casts are not supported. Therefore, to assign an integer literal value to a byte variable, one must do the following:

```
byte x = (byte) 0;
```

4 Declaration and scope

4.1 Variable declaration and scope

Each variable must be an instance of one of the primitive types or array. C//does not support global variables. Every variable must therefore be declared in a function body. Their lifetime is restricted to the lifetime of the function as they reside on the stack frame of the current function. The local variables are destroyed when the function returns.

Each variable has its identifier which is specified at the variable declaration. Furthermore, variables of the same type can be declared on one line and can be assigned a value. The following snippet demonstrates declaration of variables:

```
int useless_func() {
        int x = 5, y;

        y = 41;
        return x + y;
}
```

4.2 Arrays

Arrays are allocated on the stack frame of current function. Variable length arrays are also supported. Arrays are not allowed to be returned from a function as they are allocated on stack and are destroyed when the function returns. They can however be passed as function arguments and are passed by reference. This means that if an array element is changed in a called function, changes are propagated to the original array. The size of allocated arrays is specified in a pair square brackets just like in C:

```
        int array_of_32_integers[32];
```

There are two options when no size is specified in square brackets at declaration of an array. First, another array will be assigned to it by a reference. Secondly, the array represents a string and a literal is assigned to it. In such a case, the size of the array is calculated by the compiler.

The following code snippet demonstrates how to work with arrays:

```
/* Caesar */
void encrypt(byte result[], byte plaintext[], int plaintext_size) {
        int i = 0;

        while (i < plaintext_size) {
                byte tmp = (plaintext[i] - 'a') % (byte) 26;
                result[i] = 'a' + tmp;
                i = i + 1;
        }
        result[i] = (byte) 0;
}

int main() {
        byte plaintext[] = "Long live the Roman empire!";
        byte ciphertext[strlen(plaintext) + 1];

        ciphertext = encrypt(plaintext, plaintext, strlen(plaintext));
        <==ciphertext;
        <==;
        return 0;
}
```

C// supports multidimensional arrays and everything described above works in the same way for multidimensional arrays. The following code snippet demonstrates a declaration and usage of a 2 dimensional array:

```
int matrix[10][10];
int i = 0;

/* matrix[i] returns array of dimension 1 */
while (i < 10) {
        int j = 0;
        while (j < 10) {
                if (i == j) {
                        matrix[i][j] = i;
                } else {
                        matrix[i][j] = 0;
                }
                j = j + 1;
        }
        i = i + 1;
}
```

4.3 Function declaration

A function needs an identifier, function parameters and a return value type. The function body contains the code which is executed when the function is called. A function must contain at least some code, otherwise the code won't compile. The code block must be enclosed in curly brackets. Specially, if a function is a procedure and does not return any value, the return value type should be `void`. The following example demonstrates how to declare a simple function:

```
int sum(int x, int y) {
        return x + y;
}
```

4.3.1 Main

The program execution begins with `main` function and every program must contain it in order to be fully executable. Function `main` has the following signature:

```
        int main();
```

5 IO operations

C// supports standard IO operations such as writing to standard input and output by providing a pair of operators serving this purpose. For input, operator `==>` is used. It takes a variable (not an expression) as an argument, determines its type and loads the value from the standard input. Only simple values can be passed as arguments to the operator (including array elements) and byte arrays representing strings. Operator `<==` is used for output. An expression argument is passed to it and its type determined and the corresponding string is printed to the standard output. With no argument, a new line is printed. The following snippet demonstrates how to use these operators:

```
void f()
{
        int n;
        // load an integer value
        ==>n;
        byte s[n + 1];
        ==>s;
        int i = 0;
        while (i < n) {
                s[i] = s[i] + 1;
                i = i + 1;
        }
        <==s;
        /* Print new line character. */
        <==;
}
```

For both of these operators to work, a LLVM file containing format string definitions must be passed to clang compiler. This file is provided as a library file. Under the hood, IO operators use `printf` and `scanf` functions from standard C library for IO operations.