Language Manual

November 13, 2015

This manual presents a subset of a programming language that contains simplified C constructs. For quick references, Tables 1, 2, and 3 list the sections that explain basic syntax, data operations, and control structures, respectively.

Table 1: Basic syntax

Keyword	Description	Section
	formatting	1.1
	comments	1.2
	variables	1.3
`print`	output an integer value	1.4
`assert`	assert an expression to be "true" (nonzero)	1.5

Table 2: Data operations

Keyword	Description	Section
	integer expressions	2.1
	array accesses	2.2
`malloc`	allocate memory for an array	2.2
`free`	deallocate memory from an array	2.2
`open`	open an input file	2.3
`read`	read a byte from an input file	2.3
`end`	test the end of the current input unit or input file	2.3
`seek`	update the input offset	2.3
`pos`	return current input offset	2.3

Table 3: Control structures

Keyword	Description	Section
`if/if-else`	conditional statements	3.1
`while`	loops	3.2
`break`	exit the current loop	3.2
`continue`	skip the current loop iteration	3.2
`main`	define the main function where a program starts	3.3
`func`	define a user function	3.4
`return`	return a value for a function	3.4

1 Basic syntax

1.1 Formatting

- **Statements:** Each statement terminates with a semicolon "`;`".
- **Spacing:** Code indentation does not matter. Arbitrary white spaces are allowed between different components.

1.2 Comments

A comment starts with "`//`" and contains all text up to the end of the line.

1.3 Variables

- **Naming:** Variable names must start with a letter and may contain the following characters: lowercase letters "a"... "z", uppercase letters "A"... "Z", digits "1"... "9", and underscore "_".
- **Assignment:** Figure 1 presents the syntax for assignment statements.
- **Types:** Each variable may belong to one of three types: integer, array, or file. The type of each variable remains unchanged during its lifetime.
- **Definition:** To define a new variable, assign it a value of the desired type.
- **Scoping:** Integer and array variables are local and must be defined inside functions. File variables are global and must be defined before all functions.

Example: Figure 2 presents an example program that defines variables. The program defines a file variable `f` for an input file named "data", an integer variable `i` of value `7`, and an array `a` consisting of 12 integers.

1.4 System output

Figure 3 presents the syntax for `print` statements. A `print` statement takes the lowest byte (8 bits) of the argument, converts the byte to display form, and prints it on the screen. Table 4 describes the display form.

1.5 Assertions

Figure 4 presents the syntax for `assert` statements. An `assert` statement triggers an error when the specified condition is "false" (evaluates to zero).

```
x = some_expression;
// x becomes the result of some_expression.
```

Figure 1: Syntax for variable assignments

```
f = open("data");
main {
  i = 7;
  a = malloc(12);
  // Do something here.
  return 0;
}
```

Figure 2: Example variable assignments

```
print(some_expression);
// Output the lowest byte in the result of some_expression.
```

Figure 3: Syntax for system output

```
assert(condition);
// If condition is false, trigger an error.
// If condition is true, this is a no-op.
```

Figure 4: Syntax for assertions

Table 4: System output display form

Byte value	Display form
9	horizontal tab (`'\t'`)
10	new line (`'\n'`)
32	space (`' '`)
33 . . . 126	the ASCII character for the value
others	a backslash "\" followed by the value

2 Data operations

2.1 Integers

- **Representation:** All integers are 32-bit two's-complement integers.
- **Boolean conversion:** When using integers in logical expressions, value `0` is "false" and non-zero values are "true". When using logical expressions in arithmetic operations, value "false" is `0` and value "true" is `1`.
- **Operators:** Tables 5, 6, and 7 describe arithmetic operators, logical operators, and brackets, respectively. Table 8 lists the precedence and associativity of these operators in descending precedence. There are no self-modifying operators such as "`++`" and "`+=`" in C.

2.2 Arrays

- **Allocation and deallocation:** Figure 5 presents the syntax for array allocation and deallocation. To allocate an array, the `malloc` statement requests space of a given size from the system memory, initializes all elements to `0`, and assigns the space to an array variable. To deallocate an existing array, the `free` statement returns its space to the system for future allocation and makes the array variable invalid.
- **Reading and writing:** To read or write an element in an array, use a pair of brackets "`[]`" to surround the index of the element. The indices range from `0` to `(n-1)` for arrays with `n` elements. There are no multi-dimensional arrays as in C.

Example: Figure 6 presents example code for array operations. The code allocates an array `a` with two integers, updates the elements to `1 0`, updates variable `x` to 0, and deallocates array `a`.

2.3 Files

- **Opening:** Figure 7a presents the syntax for opening files. An `open` statement associates an input file to a file variable and initializes the input offset to the start of the file. The `open` statement must appear outside any functions.
- **Reading:** Figure 7b presents the syntax for reading files. A `read` statement returns a byte from an input file and advances the input offset by one. The receiving variable to the left is mandatory.
- **Testing the end:** The `end` predicate tests whether all bytes in the current input unit or input file has been read.
- **Seeking:** Figure 7c presents the syntax for seeking input files to specific offsets.
- **Current offset:** The `pos` expression returns the current offset of a file.

Examples: Figure 8a presents an example program that outputs the contents in file "data". Figure 8b presents example code that seeks to the previous byte in a file.

Table 5: Arithmetic operators

Operator	Operation	Example expression	Example result
`+`	add	`9 + 4`	`13`
`-` (binary)	subtract	`9 - 4`	`5`
`-` (unary)	negation	`-4`	`-4` (0xfffffffc)
`*`	multiply	`9 * 4`	`36`
`/`	divide	`9 / 4`	`2`
`%`	modulo (remainder)	`9 % 4`	`1`
`&`	bitwise AND	`9 & 5`	`1` (0x00000001)
`|`	bitwise OR	`9 | 5`	`13` (0x0000000d)
`~`	bitwise NOT	`~9`	`-10` (0xfffffff6)
`>>`	shift right arithmetic	`15 >> 2`	`3` (0x00000003)
`<<`	shift left	`15 << 2`	`60` (0x0000003c)

Table 6: Logical operators

Operator	Operation	Example expression	Example result
`==`	equal to	`9 == 4`	`0` (“false”)
`!=`	unequal to	`9 != 4`	`1` (“true”)
`<`	less than	`9 < 4`	`0` (“false”)
`<=`	less than or equal to	`9 <= 4`	`0` (“false”)
`>`	greater than	`9 > 4`	`1` (“true”)
`>=`	greater than or equal to	`9 >= 4`	`1` (“true”)
`&&`	logical AND	`(1 < 2) && (3 < 2)`	`0` (“false”)
`||`	logical OR	`(1 < 2) || (3 < 2)`	`1` (“true”)
`!`	logical NOT	`!(1 < 2)`	`0` (“false”)

Table 7: Brackets

Operator	Operation
`()`	force precedence
`[]`	access an array

Table 8: Operator precedence and associativity

Precedence	Operator	Associativity
1	`()` `[]`	left to right
2	`-` (unary) `~`	N/A
3	`*` `/` `%`	left to right
4	`&` `|` `>>` `<<`	left to right
5	`+` `-` (binary)	left to right
6	`==` `!=` `<` `<=` `>` `>=`	left to right
7	`!`	N/A
8	`&&` `||`	left to right

```
arr = malloc(n);
// If the allocation succeeds, arr becomes a valid array of n elements initialized to 0.
```

(a) Allocation

```
// arr was a valid array.
free(arr);
// arr becomes invalid.
```

(b) Deallocation

Figure 5: Syntax for arrays

```
a = malloc(2); // a initializes to 0 0.
a[0] = 1; // a becomes 1 0.
x = a[1]; // x becomes 0.
free(a); // a becomes invalid.
```

Figure 6: Example array operations

```
f = open("data");
// If opening "data" succeeds, f becomes valid and is ready to read its 0th byte.
// Define functions here.
```

(a) Opening

```
// f was ready to read the i-th byte.
x = read(f);
// If reading succeeds, x becomes the value of the i-th byte, and f is ready to read
// the (i+1)-th byte.
```

(b) Reading

```
seek(f, x);
// f is ready to read the x-th byte.
```

(c) Seeking

Figure 7: Syntax for files

```
f = open("data");
main {
  while (!end(f)) {
    x = read(f);
    print(x);
  }
  return 0;
}
```

(a) Sequential accessing

```
seek(f, pos(f)-1);
```

(b) Random accessing

Figure 8: Example file operations

3 Control structures

3.1 Conditional statements

Figure 9 presents the syntax for two forms of conditional statements: `if` and `if-else`. The curly braces "`{}`" are mandatory. There are no shorthanded "`else if`" structures as in C.

Example: Figure 10 presents example code that uses an `if` statement to set variable `i` to `5`.

3.2 Loops

Figure 11 presents the syntax for `while` loops. A `while` loop repeats executing a block of code as long as a given condition holds. The curly braces "`{}`" are mandatory. There are no "`for`" loops as in C.

To manipulate the execution of `while` loops, `break` and `continue` statements exit the innermost surrounding loop and stop the current iteration of the innermost surrounding loop, respectively.

Examples: Figure 12 presents two pieces of example code that use `while` loops. Figure 12a is a no-op. Figure 12b contains an infinite loop.

3.3 The `main` function

Every program begins execution inside a special `main` function. Figure 13 presents the syntax for the `main` function.

3.4 User functions

- **Definition:** Figure 14a presents the syntax for the `func` keyword which defines user functions. Each user function may take an integer argument that is passed by value or may take no argument. Global files are accessible inside functions. Each function returns an integer value.
- **Naming:** Function names have the same rule as variable names discussed in Section 1.3.
- **Scoping:** All functions are global and must be defined in the top level of the program, parallel to the `main` function.
- **Invocation:** Figure 14b presents the syntax for calling user functions. There must be a variable that receives the return value.
- **Recursion:** User functions may be recursive, expressing operations by calling themselves.

Example: Figure 15 presents an example function, `inc`, which calculates `x` plus one.

```
if (condition) {
  // Execute if condition is true.
}
```

(a) `if`

```
if (condition) {
  // Execute if condition is true.
} else {
  // Execute if condition is false.
}
```

(b) `if-else`

Figure 9: Syntax for conditional statements

```
i = 2 - 7; // i becomes -5.
if (i < 0) { // Condition -5 < 0 is true.
  i = -i; // i becomes 5.
}
```

Figure 10: Example conditional statement

```
while (condition) {
  // Repeat executing while condition is true.
}
```

Figure 11: Syntax for `while` loops

```
while (i < 5) {
  break;
  i = i + 1;
}
```

(a) No-op

```
i = 0;
while (i < 5) {
  continue;
  i = i + 1;
}
```

(b) Infinite loop

Figure 12: Example `while` loops

```
// Open input files here.
// Define other functions here.
main {
  // Do something here.
  return some_expression;
}
```

Figure 13: Syntax for the `main` function

```
// Open input files here.
func foo (x) {
  // The argument is passed by value.
  // Do something here.
  return some_expression;
}
func bar () {
  // Do something here.
  return some_expression;
}
// Define other functions here.
```

(a) Definition

```
y = foo(some_expression);
z = bar();
```

(b) Invocation

Figure 14: Syntax for user functions

```
func inc (x) {
  return x + 1;
}
```

Figure 15: Example function definition