

C Programs: Simple Statements and Expressions

C Program Structure

A C program that consists of only one function has the following form:

```
preprocessor directives

main function heading
{
    declarations

    statements
}
```

We have already seen some `preprocessor directives` and have discussed some `variable declarations`.

Statements

Statements are instructions to be executed by the computer. C has a number of different types of instructions. In the course we will study most of them.

Today we start with three basic types: function call, assignment and function return.

Function Call

Function call has the following syntax:

```
<functionName>(<arg1>,<arg2>,...,<argN>);
```

Here, `<functionName>` is the name of the function (e.g., `printf`), and `<arg1>`,... `<arg2>` are arguments that the function takes.

C distinguishes between C standard library functions and user-defined functions.

Functions in standard C library. a wide range of tasks in C are not performed by C statements. Instead, these tasks are *outsourced* to a collection of functions that is available for use with *any* C program. This collection is called the standard C library.

These functions have already been implemented by the C developers: all you need to do in order to use them is to provide a preprocessor directive `#include` which would specify the exact part of the standard C library which contains the function(s) you are using.

The following standard C library files contain functions important for our course:

File	Explanation	Examples
<code>stdio.h</code>	Standard Input/Output, File Input/Output	<code>printf()</code> , <code>scanf()</code>
<code>math.h</code>	Mathematical functions	<code>sqrt()</code> , <code>sin()</code> , <code>cos()</code> , <code>log()</code> , <code>exp()</code>
<code>stdlib.h</code>	Memory allocation functions, misc functions	<code>malloc()</code> , <code>free()</code> , <code>rand()</code> , <code>atoi()</code>
<code>string.h</code>	String manipulation functions	<code>strcpy()</code> , <code>strcat()</code> , <code>strlen()</code>
<code>time.h</code>	Functions related to time and system time	<code>clock()</code> , <code>localtime()</code> , <code>difftime()</code>

(note, other components of standard C library exist, but they will not be covered in the course).

To specify that the included file belongs to the standard C library put the file name in angle brackets (`<>`) in the `#include` command:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
```

User-defined functions. User-defined functions are reusable program components that the programmer elected to make structurally independent from the main program. Each user-defined function must have a name, a type and a list of input parameters (if any). The programmer is responsible for providing proper definition and (if necessary) declaration of the function.

More on this - in two weeks!

List of standard library functions. You have already seen two I/O functions: `printf()` and `scanf()`. The list below highlights some of the mathematical functions available to you in C. See Appendix B. of the textbook for full list.

Function declaration	Library header file	Purpose	Example
<code>int abs(int)</code>	<code>stdlib.h</code>	absolute value	<code>abs(-4) = 4</code>
<code>double fabs(double)</code>	<code>math.h</code>	<code>abs()</code> for floating point values	<code>fabs(-3.432)=3.432</code>
<code>double ceil(double)</code>	<code>math.h</code>	smallest integral value no less than argument	<code>ceil(7.453) = 7.0</code> <code>ceil(-4.53)= -4.0</code>
<code>double floor(double)</code>	<code>math.h</code>	largest integral value no less than the argument	<code>floor(-4.56)=-5.0</code> <code>floor(6.54) = 6.0</code>
<code>double cos(double)</code>	<code>math.h</code>	cosine	<code>cos(0.0)=1.0</code>
<code>double sin(double)</code>	<code>math.h</code>	sine	<code>sin(0.0) = 0.0</code>
<code>double tan(double)</code>	<code>math.h</code>	tangent	<code>tan(0.0) = 0.0</code>
<code>double exp(double)</code>	<code>math.h</code>	e^x (x is argument)	<code>exp(1.0)=2.71828...</code>
<code>double log(double)</code>	<code>math.h</code>	natural logarithm	<code>log(exp(1.0))= 1.0</code>
<code>double log10(double)</code>	<code>math.h</code>	base-10 logarithm	<code>log10(1000.0)= 3.0</code>
<code>double pow(double, double)</code>	<code>math.h</code>	power	<code>pow(2.0,4.0) = 16.0</code>
<code>double sqrt(double)</code>	<code>math.h</code>	square root	<code>sqrt(9.0)=3.0</code>
<code>int rand(void)</code>	<code>stdlib.h</code>	pseudorandom number generator	<code>rand()</code>
<code>void srand(unsigned int)</code>	<code>stdlib.h</code>	reset random number generator	<code>srand(100)</code>

int main(). One user-defined C function, `main()` has special meaning in C. This function represents the main body of the C program. As such, when the

executable file of the C program is loaded into main memory to be run by the CPU, the execution of the program starts at the beginning of the `main()` function.

In ANSI standard `main()` **must** have return type `int` and **must** have not input parameters. The type of the function **must** be explicitly declared in ANSI C (modern dialects of C do not require it, though).

0.0.1 Function Return Statement

Syntax. Function return statement has the following syntax:

```
return <Expression>;
```

Here, `<Expression>` is a C expression (see below).

Note, ANSI C standard requires a `return` statement to be the last executed statement of any function (including `main()`).

As such, it is a good practice to put a `return` statement at the end of each function *as soon as the function is created*.

For example, start your `main()` functions with the following stub:

```
/* Comment goes here */
int main() {

    return 0;
}
```

Semantics. When `return` statement is encountered during the run of the program, the following actions are performed:

1. Expression `<Expression>` gets evaluated.
2. Current function terminates its operation.
3. The computed value is returned to the caller function.
4. If current function is `main()`, the program terminates its work.

Type match. The key rule to remember is that the value of the expression in the `return` statement must have the same type as the declared type of the function.

For example, `main()` is always declared as `int main()`. Therefore, the following code:

```
/* Comment goes here */
int main() {

    return 1.234;
}
```

will result in a compiler error.

Assignment

Assignment statements are the building blocks of the C program. Assignment statements allow variables to receive new values as a result of various computations performed within a program.

Syntax. The assignment statements we will be studying for now have the following syntax:

```
<VariableName> = <Expression>;
```

Here, <VariableName> is a name of a program variable and <Expression> is a C expression (see below).

Semantics. The assignment statement is evaluated as follows:

1. <Expression> is evaluated.
2. Computed value of <Expression> is stored in the memory cells allocated for the variable <VariableName>.

Example.

```
int x, y;
float a,b;

x = 0;
y = x + 1;
x = x + x;
a = 2.0;
b = a / (y + x);
```

Constraints. The evaluation of the assignment statement is subject to the following conditions:

1. **Declared variable.** Variable <VariableName> must be declared prior to its use in the assignment statement.
2. **Type match.** <Expression> must evaluate to a type compatible with the declared type for the variable <VariableName>.

Expressions

At this point we concentrate on arithmetic expressions. A general forms of an expression are

```
<Operand> <Operation>
<Operation> <Operand>
<Operand> <Operation> <Operand>
```

C has the following operations:

Operation	Arity	Meaning	Example
+	binary	addition	5 + 6 evaluates to 11
-	binary	subtraction	6 - 5 evaluates to 1
*	binary	multiplication	4 * 3 evaluates to 12
/	binary	integer division	4/5 evaluates to 0
/	binary	division	4.0/5.0 evaluates to 0.8
%	binary	remainder	10 % 7 evaluates to 3
-	unary	unary minus	- 4 evaluates to -4
+	unary	unary plus	+4 evaluates to 4

<Operand> can be any of the following:

- constant value (2, 5.6, etc.)

- defined constant (`HUNDRED`)
- variable name (`currentStateVote`)
- function call (`log10(x)`)
- (arithmetic) expression in parentheses (`(log10(x) + 5)`)

Evaluation, operation precedence. Arithmetic expressions in C are evaluated according to the mathematical rules. Generally speaking, given an arithmetic expression in C, its evaluation proceeds as follows:

1. Using C operator precedence and associativity rules, the operation `<Op>` with the least precedence (i.e., last to be evaluated) is established.
2. The expression is represented as `<Operand1> <Op> <Operand2>`¹
3. Operand `<Operand1>` is evaluated.
4. Operand `<Operand2>` is evaluated.
5. The result of the operation `<Op>` applied to the results of evaluating `<Operand1>` and `<Operand2>` is evaluated. This result is returned as the value of the expression.

As seen, expression evaluation depends on operator precedence. The precedence rules are:

Precedence level	Operators
1. (highest)	parentheses
2.	<code>+, -</code> (unary)
3.	<code>*, /, &</code>
4. (lowest)	<code>+, -</code> (binary)

The associativity rule is:

Unary operators in the same subexpression and at the same precedence level are evaluated right-to-left (*right-associativity*).

Binary operators in the same subexpression and at the same precedence level are evaluated left-to-right (*left-associativity*).

Examples. Expression `a + b*c - (15 + a * n) + 23` has the following precedence of operations: `((a + (b*c)) - (15 + (a*n))) + 23`.

Expression `a + b + -c* --d` has the following precedence of operations: `((a+b)+ ((-c) * (-+(- d))))`.

Expression `b * c / e % f` has the following precedence of operations: `((b*c)/e)% f`.

Type of expression. Type conversion.

- **int.** If both operands (sub-expressions) in an arithmetic expression evaluate to `int` values, then the expression evaluates to an `int` value.
- **double/float.** If both operands (sub-expressions) evaluate to `float` (or `double`), the the expression evaluates to a `float` (`double`) value.
- **mix.** If one sub-expression evaluates to an `int` and the other to a `float` (`double`), then the expression evaluates to a `float` (`double`).

Note. This is how the type of the division operation is determined: if both operands are integers, integer division is applied, otherwise - floating point division.

¹In case of binary operators. For unary operators, similar intuition holds, except, only one operand gets evaluated.