Compiler Specification for *ctowasm* 1.0

1. Preface

ctowasm is a project involving the implementation of C to WebAssembly (Wasm) compiler, in Typescript, primarily for usage on the Source Academy web platform.
It supports a subset of the C17 standard, specifically ISO/IEC 9899:2017, which this document serves to specify. From this point, the "C17 standard" will specifically refer to the final draft of ISO/IEC9899:2017**.** Additionally, the term "*ctowasm subset*" shall refer to language defined as the subset of C17 supported by the *ctowasm* compiler.

This document is to meant to be read with the C17 standard as accompanying reference. All terms, definitions and symbols which are used in this document but are not defined explicitly within this document itself are defined in section 3 of the C17 standard. This document will also refer the reader to the C17 standard for more information on a feature, where appropriate.

This document is divided into two main sections. Section 2 specifies the subset of the C17 language that this compiler supports, whereas section 3 documents implementation specific behaviour/features of the *ctowasm* compiler. For conciseness, some sub sections in section 2 may contain implementation specific specifications such as the representation of types in section 2.2.

Any language feature present in the C17 standard but not in this compiler specification is defined as not within the supported language subset. Some unsupported features may be explicitly mentioned as not supported, for the purpose of clarity. Such mentions will be in *italics* and preceded by the [!] marker.

As the language supported by *ctowasm* is meant to be a subset of the C17 specification, for each language feature that is present in the subset, the reader may assume that all constraints and semantics of that particular language feature, as stated in the C17 standard, are adhered to. For brevity this document will not list these out. The reader may instead refer directly to the C17 standard.

Table of Contents

2. C Language Subset Specification

2.1 Syntax

2.1.1 Character sets

The source character sets and execution character sets (terminology defined in 5.2.1of ISO/IEC9899:2017) are supersets of the basic source and execution character sets defined in the C17 standard, respectively.

Both source and execution character sets consist of the following members:

The 26 uppercase letters of the Latin alphabet:
A B C D E F G H J K L M N O P Q R S T U V W X Y Z

The 26 lowercase letters of the Latin alphabet:
a b c d e f g h j k l m n o p q r s t u v w x y z

The 10 decimal digits:
0 1 2 3 4 5 6 7 8 9

The following 30 graphic characters:
! " # % & ′ ( ) * + , - . / : ; < = > ? [\ ] ^ _ { | } ~ @

The space character, and control characters representing horizontal tab, vertical tab, form feed and newline. (control characters are " ", "\t", "\v", "\f", "\n" respectively)

Additionally, the execution character set contains the control characters representing alert, backspace and carriage return (control characters are "\a", "\b" and "\r" respectively).

Finally, in a source C program supported by this compiler, the end of each line of text is indicated by the presence of a newline ("\n") character.

2.1.2 Lexical grammar

This section defines the lexical grammar of a C program supported by the *ctowasm* compiler.

Non-terminal symbols are in *italics.* Terminal symbols are **bolded**. Words that are present in a rule but are not actually part of the expression (merely present for descriptive purposes) will be regular text.

The "opt" subscript indicates if a given symbol is optional.

2.1.2.1 Lexical elements

token:

keyword
identifier
constant
string-literal
punctuator

2.1.2.2 Keywords

reserved-word:

keyword
unsupported-keyword

*keyword***:** one of (each terminal symbol is delimited by a 2 whitespaces to save space)

auto break case char const continue default do double else enum float for if inline int long return short signed sizeof static struct switch typedef unsigned void while

unsupported-keyword: one of

extern goto inline register restrict volatile _Alignas _Alignof _Atomic _Bool _Complex _Generic _Imaginary _Noreturn _Static_assert _Thread_local

2.1.2.3 Identifiers

identifier:

nondigit
identifier nondigit
identifier digit

nondigit: one of

_ a b c e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V X Y Z

digit: one of

0 1 2 3 4 5 6 7 8 9

2.1.2.4 Constants

constant:
    integer-constant
    floating-constant
    enumeration-constant
    character-constant

integer-constant:
    decimal-constant integer-suffix$_{opt}$
    octal-constant integer-suffix$_{opt}$
    hexadecimal-constant integer-suffix$_{opt}$

decimal-constant:
    nonzero-digit
    decimal-constant digit

octal-constant:
    0
    octal-constant octal-digit

hexadecimal-constant:
    hexadecimal-prefix hexadecimal-digit
    hexadecimal-constant hexadecimal-digit

hexadecimal-prefix: one of
    0x 0X

nonzero-digit: one of
    1 2 3 4 5 6 7 8 9

octal-digit: one of
    0 1 2 3 4 5 6 7

hexadecimal-digit: one of
    0 1 2 3 4 5 6 7 8 9
    a b c d e f A B C D E F

integer-suffix:
    unsigned-suffix long-suffix$_{opt}$
    unsigned-suffix long-long-suffix
    long-suffix unsigned-suffix$_{opt}$
    long-long-suffix unsigned-suffix$_{opt}$

unsigned-suffix: one of
    u U

long-suffix: one of
l L

long-long-suffix: one of
ll LL

floating-constant:
decimal-floating-constant
hexadecimal-floating-constant

decimal-floating-constant:
fractional-constant exponent-part$_{opt}$ *floating-suffix*$_{opt}$
digit-sequence exponent-part floating-suffix$_{opt}$

fractional-constant:
digit-sequence$_{opt}$ **.** *digit-sequence*
digit-sequence **.**

exponent-part:
e *sign*$_{opt}$ *digit-sequence*
E *sign*$_{opt}$ *digit-sequence*

sign: one of
+ -

digit-sequence:
digit
digit-sequence digit

hexadecimal-digit-sequence:
hexadecimal-digit
hexadecimal-digit-sequence hexadecimal-digit

floating-suffix: one of
f l F L

enumeration-constant:
identifier

character-constant:
' *c-char-sequence* **'**

c-char-sequence:
c-char
c-char-sequence c-char

c-char:

any member of the source character set (2.1) except the single quote ', backslash \ or newline character.
escape-sequence

escape-sequence:

simple-escape-sequence
octal-escape-sequence
hexadecimal-escape-sequence

simple-escape-sequence: one of

\' \" \? \\ \a \b \f \n \r \t \v

octal-escape-sequence:

\ *octal-digit*
\ *octal-digit octal-digit*
\ *octal-digit octal-digit octal-digit*

hexadecimal-escape-sequence:

\x *hexadecimal-digit*
hexadecimal-escape-sequence hexadecimal-digit

2.1.2.5 String Literals

string-literal:

" *s-char-sequence*$_{opt}$ **"**

s-char-sequence:

s-char
s-char-sequence s-char

s-char:

any member of the source character set except the double-quote ", backslash \, or newline character
escape-sequence

2.1.2.6 Punctuators

punctuator: one of

[ ] ( ) { } . -> ++ -- & * + - ~ ! / % << >> < > <= >= == != ^ | && ||
? : ; ... = *= /= %= += -= <<= >>= &= ^= |= , # ##
<: :> <% %> %: %:%:

2.1.3 Phrase structure grammar

A C program written for the *ctowasm* compiler is defined by the following phrase structure grammar. The grammar largely resembles C17 grammar, with simplifications in rules where excluded language features are removed.

ctowasm 1.0 only supports compilation of a single C program (in other words, 1 source "file"). A single C program is defined to begin with the start symbol "*c-program*". Furthermore, preprocessor directives are not supported. However the symbol "includes-directive" resembles the "#include" preprocessor directive present in the standard (6.10). Its function is conceptually similar but fundamentally different in how it functions. This is described in greater detail in section XXX.

Non-terminal symbols are in *italics.* Terminal symbols are **bolded**. Words that are present in a rule but are not actually part of the expression (merely present for descriptive purposes) will be regular text.

2.1.3.1 C Program

c-program:
includes-directive-list$_{opt}$ *translation-unit*

includes-directive-list:
includes-directive
includes-directive-list includes-directive

includes-directive:
*include* **<** *external-library-name* **>**

external-library-name:
identifier

translation-unit:
external-declaration
translation-unit external-declaration

external-declaration:
function-definition
declaration

function-definition:
declaration-specifiers declarator declaration-list$_{opt}$ *compound-statement*

declaration-list:
declaration
declaration-list declaration

2.1.3.2 Expressions

primary-expression:
identifier
constant
string-literal
(*expression* **)**

postfix-expression:
primary-expression

postfix-expression **[** *expression* **]**
postfix-expression **(** *argument-expression-list*$_{opt}$ **)**
postfix-expression **.** *identifier*
postfix-expression **->** *identifier*
postfix-expression **++**
postfix-expression **--**

argument-expression-list:
assignment-expression
argument-expression-list **,** *assignment-expression*

unary-expression:
postfix-expression
++ *unary-expression*
-- *unary-expression*
unary-operator unary-expression
sizeof *unary-expression*
sizeof (*type-name* **)**

unary-operator: one of
& ***** **+** **-** **~** **!**

multiplicative-expression:
multiplicative-expression ***** *unary-expression*
multiplicative-expression **/** *unary-expression*
multiplicative-expression **%** *unary-expression*

additive-expression:
multiplicative-expression
additive-expression **+** *multiplicative-expression*
additive-expression **-** *multiplicative-expression*

shift-expression:
additive-expression
shift-expression **<<** *additive-expression*
shift-expression **>>** *additive-expression*

relational-expression:
shift-expression
relational-expression **<** *shift-expression*
relational-expression **>** *shift-expression*
relational-expression **<=** *shift-expression*
relational-expression **>=** *shift-expression*

equality-expression:
relational-expression
equality-expression **==** *relational-expression*
equality-expression **!=** *relational-expression*

and-expression:

equality-expression
and-expression **&** *equality-expression*

exclusive-or-expression:
and-expression
exclusive-or-expression ^ *and-expression*

inclusive-or-expression:
exclusive-or-expression
inclusive-or-expression | *exclusive-or-expression*

logical-and-expression:
inclusive-or-expression
logical-and-expression **&&** *inclusive-or-expression*

logical-or-expression:
logical-and-expression
logical-or-expression || *logical-and-expression*

conditional-expression:
logical-or-expression
logical-or-expression **?** *expression* **:** *conditional-expression*

assignment-expression:
conditional-expression
unary-expression assignment-operator assignment-expression

assignment-operator: one of
= *= /= %= += -= <<= >>= &= ^= |=

expression:
assignment-expression
expression **,** *assignment-expression*

constant-expression:
conditional-expression

2.1.3.3 Declarations

declaration:
declaration-specifiers init-declarator-list$_{\text{opt}}$ **;**

declaration-specifiers:
storage-class-specifier declaration-specifiers$_{opt}$
type-specifier declaration-specifiers$_{opt}$
type-qualifier declaration-specifiers$_{opt}$

init-declarator-list:
init-declarator
init-declarator-list **,** *init-declarator*

init-declarator:
    declarator
    declarator = *initializer*

storage-class-specifier:
    typedef
    auto
    static

type-specifier:
    void
    char
    short
    int
    long
    float
    double
    signed
    unsigned
    struct-specifier
    enum-specifier
    typedef-name

struct-specifier:
    struct *identifier*$_{opt}$ **{** *struct-declaration-list* **}**
    struct *struct-or-union identifier*

struct-declaration-list:
    struct-declaration
    struct-declaration-list struct-declaration

struct-declaration:
    specifier-qualifier-list struct-declarator-list$_{opt}$

specifier-qualifier-list:
    type-specifier specifier-qualifier-list$_{opt}$
    type-qualifier specifier-qualifier-list$_{opt}$

struct-declarator-list:
    struct-declarator
    struct-declarator-list **,** *struct-declarator*

struct-declarator:
    declarator

enum-specifier:
    enum *identifier*$_{opt}$ ***{*** *enumerator-list* ***}***
    enum *identifier*$_{opt}$ ***{*** *enumerator-list* **,** ***}***
    enum *identifier*

enumerator-list:
    enumerator
    enumerator-list **,** *enumerator*

enumerator:
    enumeration-constant
    enumeration-constant = *constant-expression*

type-qualifier:
    const

declarator:
    pointer$_{opt}$ *direct-declarator*

direct-declarator:
    identifier
    (*declarator* **)**
    direct-declarator **[** *assignment-expression*$_{opt}$ **]**
    direct-declarator **(** *parameter-list* **)**

pointer:
    ***** *type-qualifier-list*$_{opt}$
    ***** *type-qualifier-list*$_{opt}$ *pointer*

type-qualifier-list:
    type-qualifier
    type-qualifier-list type-qualifier

parameter-list:
    parameter-declaration
    parameter-list **,** *parameter-declaration*

parameter-declaration:
    declaration-specifiers declarator
    declaration-specifiers abstract-declarator$_{opt}$

identifier-list:
    identifier
    identifier-list **,** *identifier*

type-name:
    specifier-qualifier-list abstract-declarator$_{opt}$

abstract-declarator:
    pointer
    pointer$_{opt}$ *direct-abstract-declarator*

direct-abstract-declarator:
    (*abstract-declarator* **)**
    direct-abstract-declarator$_{opt}$ **[** *assignment-expression*$_{opt}$ **]**

direct-abstract-declarator$_{opt}$ **(** *parameter-type-list*$_{opt}$ **)**

typedef-name:
identifier

initializer:
assignment-expression
{ *initializer-list* **}**
{ *initializer-list* **,** **}**

initializer-list:
initializer
initializer-list **,** *initializer*

2.1.3.4 Statements

statement:
compound-statement
expression-statement
selection-statement
iteration-statement
jump-statement

compound-statement:
{ *block-item-list*$_{opt}$ **}**

block-item-list:
block-item
block-item-list block-item

block-item:
declaration
statement

expression-statement:
expression$_{opt}$ **;**

selection-statement:
if (*expression* **)** *statement*
if (*expression* **)** *statement* **else** *statement*
switch (*expression* **) {** *switch-case-list*$_{opt}$ *switch-default-case*$_{opt}$ **}**
switch (*expression* **)** *statement*

switch-case-list:
switch-case
switch-case-list switch-case

switch-case:
case *constant-expression* **:** *statement*

switch-default-case:
default **:** *statement*

iteration-statement:
while (*expression* **)** *statement*
do *statement* **while (** *expression* **) ;**
for ($expression_{opt}$ **;** $expression_{opt}$ **;** $expression_{opt}$ **)** *statement*
for (*declaration* $expression_{opt}$ **;** $expression_{opt}$ **)** *statement*

jump-statement:
continue ;
break ;
return $expression_{opt}$ **;**

2.2 Types

This section specifies the subset of types that are supported by this compiler, along with all supported type-specifiers and qualifiers. Their representations in memory (as specified in the memory model section) are also specified.

2.2.1 Arithmetic types

This section documents all supported arithmetic types.

2.2.1.1 Integer types

Basic Integer Types

The *ctowasm* subset contains all integer types defined in C17. Figure XX shows the list of integer types, along with their sizes in memory, as defined by the compiler implementation.

Integer type	*Memory size in bytes*
char	1
short int	2
int	4
long int	8
long long int	8

As per the C17 standard, the **char** type is able to contain all characters of the execution character set specified in section 2.1.1.

Both **unsigned** and **signed** integer type variants are supported. Signed integer types are represented in memory using 2's complement. It suffices to state the length specifier only for **short int, long int**, **long long int** without the accompanying **int**.

Enumerated types

Enumerated types (as defined by the **enum** keyword). In this compiler implementation, all enumerated types are compatible with **signed int**.

2.2.1.2 Floating types

ctowasm subset contains all the real floating types defined in the C17 subset. Figure XX shows all real floating types along with their size in memory in number of bytes as per the *ctowasm* compiler.

Real floating type	*Memory size in bytes*
float	4
double	8
long double	8

All floating types are represented in memory using the IEEE 754 standard defined for the number bits that are used to represent them in memory.

[!] *Complex floating types are not supported.*

2.2.2 Void type

The definition of void type is the same as specified in 6.2.5/19 of C17 standard.

2.2.4 Derived types

2.2.2.1 Array Type

The definition of a an *array type* is the same as specified in 6.2.5/20 of the C17 standard. In the *ctowasm* subset, the number of elements of the array must be specified using an expression that can be evaluated at compile-time, consisting of only *integer-constant*(s). e.g. "int arr[10 + 20]" is a valid array declaration, but "int arr[x]" where x is a declared variable is not.

2.2.2.2 Structure type

The definition of *structure type* (defined using **struct** keyword) is the same as specified in 6.2.5.20 of the C17 standard.

2.2.2.3 Function type

The definition of *function type* is the same as specified in 6.2.5/20 of the C17 standard.

2.2.2.4 Pointer type

The definition of a *pointer type* adheres to all specifications on *pointer type* specified in 6.2.5 of the C17 standard.

ctowasm compiler implementation of pointers

While the C17 standard only specifies that pointers are a reference to a function or object type, *ctowasm* compiler defines the actual implementation of pointers to object types and pointers to function types.

Pointers to object types are equivalent to an **unsigned int** representing the address of the object in underlying linear memory buffer (see section 3.1).

Pointers to function types are equivalent to an **unsigned int** representing the index of the underlying function. All functions that are included or defined within the C program are given a unique index in the order of definition. This is the value that is stored by the pointer to function type.

2.2.3 Type conversions

All implicit conversions of types supported by the *ctowasm* subset are supported. This includes: integer promotion (6.3.1.1/2 of C17 standard)

- usual arithmetic conversions (6.3.1.8 of C17 standard)
- lvalue conversion (6.3.2.1/2 of the C17 standard)
- array to pointer type conversion (6.3.2.1/3 of the C17 standard)
- function to pointer type conversion (6.3.2.1/4 of the C17 standard)
- void pointer type conversions (6.3.2.3/1 of the C17 standard)

2.3 Subset features

All language features that are covered in the phase structure grammar in section XX are, unless stated otherwise, fully supported. This refers to the complete adherence to constraints and semantics of each feature as stated in the C17 standard, where applicable. (constraints and semantics which refer to features outside of the subset are not included. An example would be constraints relating to unsupported types like Complex numeric types)

This section details any language features which are only partially supported in the *ctowasm* subset, as well as the missing sub-features or additional limitations of the feature.

2.3.1 Switch statements

As labeled and goto statements are unsupported, switch statements are implemented differently in *ctowasm* as compared to traditional implementations. As such there are additional limitations on switch statements in comparison to the C17 standard. This section lists these limitations, as well as the relevant grammar.

selection-statement:
if (*expression* **)** *statement*
if (*expression* **)** *statement* **else** *statement*
switch (*expression* **) {** *switch-case-list*$_{opt}$ *switch-default-case*$_{opt}$ **}** // rule 1
switch (*expression* **)** *statement*

switch-case-list:
switch-case
switch-case-list switch-case

switch-case:
case *constant-expression* **:** *statement*

switch-default-case:
default **:** *statement*

The limitations are listed here:

1. A switch statement followed by a brace-enclosed block (rule 1) cannot have any *statement* immediately at the start of the brace-enclosed block. Statements within the brace-enclosed block must be part of a *switch-case* or *switch-default-case*.

2.3.2 Sizeof operator

As variable length arrays are not supported, **sizeof** expressions are evaluated strictly at compile-time to be an integer constant, equivalent to the number of bytes of the operand.

3. Compiler implementation specification

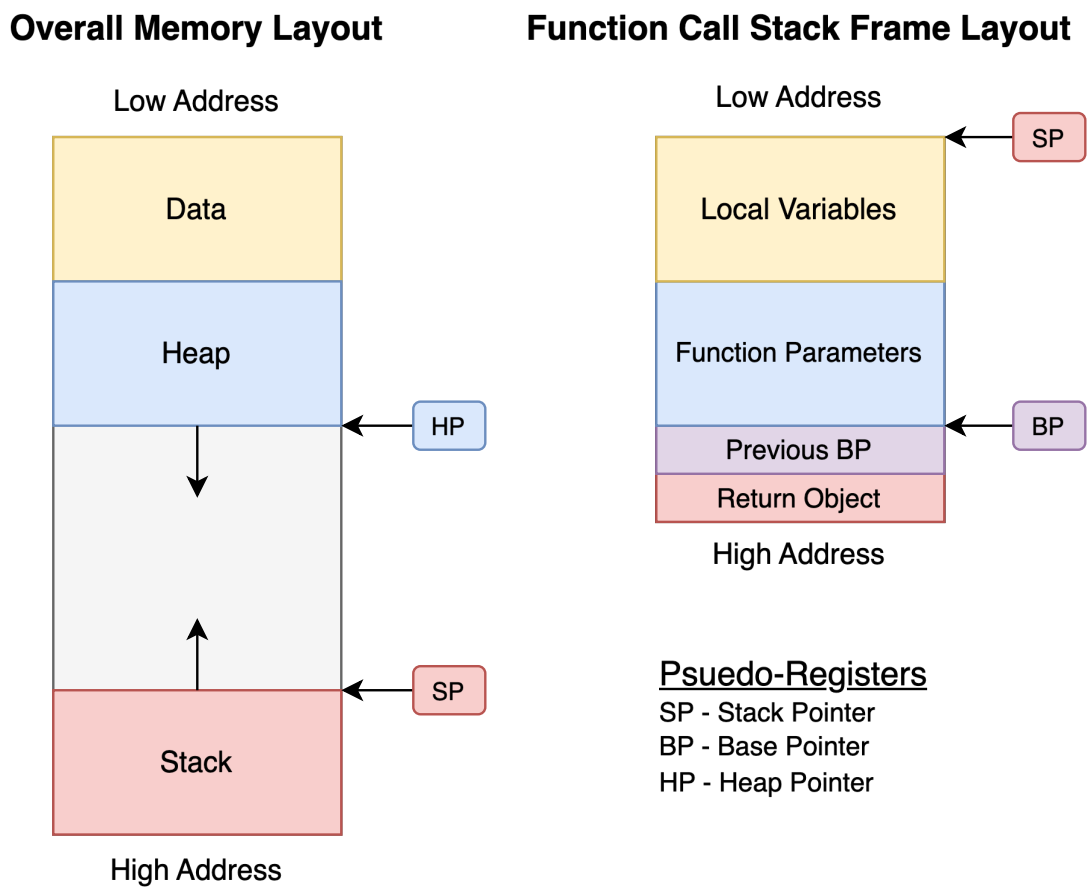
3.1 Memory model

The *ctowasm* compiler compiles C code assuming a memory model which consists of a linear array of bytes, which can be expanded when it is full and more memory is required. This directly corresponds to an JavaScript ArrayBuffer. However, the frontend of the compiler is designed such that any suitable backend which provides a linear array of bytes can be used to implement the memory model.

Figure XX depicts the memory model, showing the 3 memory segments in the linear array buffer that *ctowasm* "creates". The memory model is intentionally designed to mimic a traditional "textbook" memory model.

Global variables (variables declared outside of any function scope), *static* storage class specified variables, as well as string literals are stored in the "data" segment. Dynamically allocated variables (using malloc) are stored in the "heap" segment. Function call stack frames (also depicted) are stored in the "stack" segment. Arrows in the diagram depict growth of the memory segments when the heap and stack frames are expanded.

Finally, also depicted in Figure XX, are "psuedo-registers" which are simple WASM global variables which play a similar role to registers in typical architectures, being a reference point for accessing objects in memory.



3.2 Implementation-defined behaviour

As the *ctowasm* compiler is limited to compiling a single C translation unit in the browser environment, many of the implementation-defined behaviour defined in Annex J.3 of the C17 standard are not applicable. Implementation-defined behaviour relating to unsupported C features are also not applicable. This section documents applicable implementation-defined behaviour, in the order that they are stated in Annex J.3 of the C17 standard.

3.2.1 Diagnostic messages

The *ctowasm* compiler will report an error message(s) during compilation if the compiled translation unit contains a violation of any syntax rule or constraint that lies within the *ctowasm subset.*

As peggy.js is used for parsing, syntax errors will which are detected by peggy.js syntax errors will be propagated up to the compiler and presented to the user.

```
Compilation failed with the following error(s):

Error: Expected "(", ",", ";", "=", "[", or block but "{" found.
3 | {
```

During compilation, if any constraint violations are detected, the compiler will stop compilation and emit an error specifying the details of the violated constraint.

```
Compilation failed with the following error(s):

Error: incompatible types when initializing type 'signed int' using type 'struct X'
5 | int y = x;
```

3.2.2 Characters

- A byte is defined to be 8 bits.
- The values of each member in the execution character set, including escape sequences, is their value in ASCII.
- As a **char** is defined to be equivalent in range, representation and behaviour as a **signed char.**
- A character other than a member of the basic execution character set cannot be stored in a **char** object the source character set is a subset of the execution character set. In the event an integer type value is stored in a **char**, it obeys the same integer overflow rules as **signed char,** as they are equivalent.
- Members of the source character set are mapped to their equivalent character in the execution character set. e.g. 'x' will be mapped to 'x'.

3.2.3 Integers

- Signed integer types are represented using two's complement.
- If an attempt to convert an integer to a signed integer type when the value cannot be represented in an object of that type is made, the result will be the least significant *n* bits of the two's complement bit representation of the integer being converted, where *n* is the number of bits of the signed integer type.

- For a binary expression E1 << E2, which consists of a bitwise right shift using the ">>" operator, if E1 has a signed integer type and has negative value, then the result is that the bits of the two's complement binary representation of E1 is right shifted by E2 bit positions, with the most significant bit being set to 1, while all other vacated bits are set to 0.

3.2.4 Floating point

- Floating point types are implemented using the f32 and f64 WASM data types. As such, they correspond to the respective binary floating-point representations (single and double), defined by the IEEE 754 standard. The reader may refer to the specification for the implementation-defined behaviour pertaining to floating point types.

3.2.5 Arrays and pointers

- Pointers are equivalent to an **unsigned int**. As such, the result of converting a pointer to an integer or vice versa is equivalent to that of converting between an **unsigned int** and that integer.
- The result of subtracting two pointers to elements of the same array has type **ptrdiff_t**, which is defined by the *ctowasm* compiler to be equivalent to **signed int**. As such, the size of the result is 4 bytes.

3.2.6 Structures and enumerations

- Non-bit-fields members of structures (bit-fields are not supported in the *ctowasm* subset) have 1 byte alignment. As such, they are ordered contiguously after each other within the memory allocated for the containing struct.
- Enumerated types are compatible with **signed int**.

3.2.7 Structures and enumerations

- Non-bit-fields members of structures (bit-fields are not supported in the *ctowasm* subset) have 1 byte alignment. As such, they are ordered contiguously after each other within the memory allocated for the containing struct.
- Enumerated types are compatible with **signed int**.

3.2.8 Architecture

- The result of the **sizeof** operator has the type **size_t**, which is defined by the *ctowasm* compiler to be equivalent to **unsigned int**. As such, the, the result is a positive integer value, 4 bytes in size, which equates to the number of bytes of the operand.

4. Custom standard library

This section details the custom standard libraries that are included with the *ctowasm* compiler.

Some of the functions involve interacting with functions provided to the compiler as part of configuration parameters when it is used in a project. This is in the form of a "ModulesGlobalConfig" object provided to the exported *compileAndRun()* function. Refer to the *index.ts* file in the *ctowasm* project codebase for more details. (link)

Some methods are specified to be only callable in a browser environment.

4.1 source_stdlib

Description

Provides I/O and dynamic memory management functions.

How to include

Add the line "#include <source_stdlib>" at the top of the C program.

Functions

void print_int(int x)
Prints the *signed integer* x using the print function provided during compiler configuration.

void print_int_unsigned(unsigned int x)
Prints the *unsigned integer* x using the print function provided during compiler configuration.

void print_char(char x)
Prints the character representation (ASCII representation) of the *char* x using the print function provided during compiler configuration.

void print_long(long int x)
Prints the *signed long int* x using the print function provided during compiler configuration.

void print_unsigned_long(unsigned long int x)
Prints the *unsigned long int* x using the print function provided during compiler configuration.

void print_float(float x)
Prints the *float* x using the print function provided during compiler configuration.

void print_double(double x)
Prints the *double* x using the print function provided during compiler configuration.

void print_string(const char *p)
Prints the null-terminated string starting at the address pointed to by the *char* pointer p using the print function provided during compiler configuration.

void print_address(void *p)
Prints the address in memory pointed to by the pointer p using the print function provided during compiler configuration.

void * malloc(size_t x)
Dynamically allocates memory with size = x; Returns a pointer to the start of the allocated memory.

void free(void *p)
Frees the dynamically allocated memory at address pointed to be p. The pointer p must have been previously returned by a call to *malloc(size_t x)*. This must only be called once per unique pointer to avoid undefined behaviour.

int prompt_int()
Initiates a browser prompt, returning the *signed int* that was entered by the user.

long int prompt_long()
Initiates a browser prompt, returning the *signed long int* that was entered by the user.

float prompt_float()
Initiates a browser prompt, returning the *float* that was entered by the user.

double prompt_double()
Initiates a browser prompt, returning the *double* that was entered by the user.

void prompt_string(char *p)
Initiates a browser prompt, and places a null-terminated string that was entered by the user at the address in memory pointed to be p. Enough memory must have been allocated beforehand at the location pointed to by p for the null-terminated string. Overflow of this buffer results in undefined behaviour.

4.2 math

Description

Provides a subset of mathematical functions of the "math.h" C standard library, that are identical in functionality.

How to include

Add the line "#include <math>" at the top of the C program.

Functions

All functions declared in this module are exactly identical to their identically named counterpart in the "math.h" C standard library. The reader is directed to refer to the C17 standard for descriptions of the functions. This section will simply list all available functions.

double acos(double x)
double asin(double x)
double atan(double x)
double cos(double x)
double cosh(double x)
double sin(double x)
double sinh(double x)
double tan(double x)
double tanh(double x)
double exp(double x)
double log(double x)
double log10(double x)
double pow(double base, double exponent)
double sqrt(double x)
double ceil(double x)
double floor(double x)

4.3 utility

Description

Provides various functions which are a subset of the "stdlib.h" C standard library, that are identical in functionality.

How to include

Add the line "#include <utility>" at the top of the C program.

Functions

All functions declared in this module are exactly identical to their identically named counterpart in the "stdlib" C17 standard library. The reader is directed to refer to the C17 standard for descriptions of the functions. This section will simply list the available functions.

double atof(const char *str)
int atoi(const char *str)
long int atol(const char *str)
int abs(int x)
long int labs(long int x)
int rand()
void srand(unsigned int seed)
void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void*))

5. Appendix

5.1 Major excluded features

The readers can determine features that are excluded from the *ctowasm* subset specification by comparing the C17 standard to the ctowasm subset specification in section 2. However for the reader's convenience, this section lists some major C language features, not in any specific order, that are excluded from the subset.

1. Complex floating types
2. Compound literals
3. Hexadecimal floating constants
4. Type casting
5. Labeled statements and goto (excluding switch cases)
6. Union type
7. Bit fields
8. Variadic arguments
9. Designated list initializers
10. Variable length arrays
11. Preprocessing directives
12. Universal character names
13. String and character-constant encoding prefixes

5.2 Miscellaneous user notes

This section is a collection of miscellaneous notes intended to help users use the compiler more easily.

- While typecasting is not supported, implicit conversions between supported types are. Thus for cases where you traditionally need to typecast a value with a given data type, and the value's data type may be implicitly converted to the target data type, you can use the value in an expression that would trigger implicit conversion, or use the value as an initializer to initialize a new variable with the target data type.

  An example of the former would be casting of a integer to a double. Instead of "***(double) x***" you may instead use ***"x * 1.0***" which will trigger implicit conversion.

  An example of the latter would be when working with void pointers, which are implicitly convertible to any other pointer. Instead of using "***(int *) p***" you may first initialize a new pointer with "***int *p2 = p***" before where you originally intended to use the typecasted *p* expression.