Group 31

Author: Bisweswar Martha, A Kedarnath, Akshay Gupta, Arka Das

The subset C Reference Manual

This is a reference manual containing the features for the subset of C programming language(GNU C99) for which we are developing our compiler for which we are going to develop a compiler using the implementation language Python to convert it to MIPS language.

This shall contain a subset of all the features of C99 along with slight extensions to the base language. All details have been elaborated below.

---

1 Lexical Elements

This chapter describes the lexical elements that make up C source code after preprocessing. These elements are called *tokens*. There are five types of tokens: keywords, identifiers, constants, operators, and separators. White space, sometimes required to separate tokens, is also described in this chapter.

1.1 Identifiers

Identifiers are sequences of characters used for naming variables, functions, new data types, and preprocessor macros. You can include letters, decimal digits, and the underscore character '_' in identifiers. The first character of an identifier cannot be a digit. Identifiers are case sensitive.

1.2 Keywords

Keywords are special identifiers reserved for use as part of the programming language itself. You cannot use them for any other purpose.

Here is a list of keywords recognized by C:

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

1.3 Constants

A constant is a literal numeric or character value, such as `5` or `'m'`. All constants are of a particular data type; you can use type casting to explicitly specify the type of a constant.

1.3.1 Integer Constants An integer constant is a sequence of digits, with an optional prefix to denote a number base.

If the sequence of digits is preceded by `0x` or `0X`, then the constant is considered to be hexadecimal. Hexadecimal values may use the digits from 0 to 9, as well as the letters `a` to `f` and `A` to `F`.

If the first digit is 0 (zero), and the next character is not '`x`' or '`X`', then the constant is considered to be octal (base 8). Octal values may only use the digits from 0 to 7; 8 and 9 are not allowed.

In all other cases, the sequence of digits is assumed to be decimal.

There are various integer data types, for short integers, long integers, signed integers, and unsigned integers. You can force an integer constant to be of a long and/or unsigned integer type by appending a sequence of one or more letters to the end of the constant:

`u U` : Unsigned integer type.

`l L` : Long integer type.

1.3.2 Character Constants A character constant is usually a single character enclosed within single quotation marks, such as `'Q'`. A character constant is of type `int` by default.

Some characters, such as the single quotation mark character itself, cannot be represented using only one character. To represent such characters, there are several "escape sequences" that you can use:

`\\` : Backslash character.

`\?` : Question mark character.

`\'` : Single quotation mark.

`\"` : Double quotation mark.

`\n` : Newline character.

`\t` : Horizontal tab.

`\o, \oo, \ooo` : Octal number.

`\xh, \xhh, \xhhh, …` : Hexadecimal number.

To use any of these escape sequences, enclose the sequence in single quotes, and treat it as if it were any other character.

The octal number escape sequence is the backslash character followed by one, two, or three octal digits.

The hexadecimal escape sequence is the backslash character, followed by `x` and an unlimited number of hexadecimal digits. While the length of possible hexadecimal digit strings is unlimited, the number of character constants in any given character set is not.

1.3.3 Real Number Constants A real number constant is a value that represents a fractional (floating point) number. It consists of a sequence of digits which represents the integer (or "whole") part of the number, a decimal point, and a sequence of digits which represents the fractional part.

Real number constants can also be followed by `e` or `E`, and an integer exponent.The exponent can be either positive or negative.

1.3.4 String Constants A string constant is a sequence of zero or more characters, digits, and escape sequences enclosed within double quotation marks. A string constant is of type "array of characters". All string constants contain a null termination character (`\0`) as their last character. Strings are stored as arrays of characters, with no inherent size attribute. The null termination character lets string-processing functions know where the string ends.

Adjacent string constants are concatenated (combined) into one string, with the null termination character added to the end of the final concatenated string.

A string cannot contain double quotation marks, as double quotation marks are used to enclose the string. To include the double quotation mark character in a string, use the `\"` escape sequence. You can use any of the escape sequences that can be used as character constants in strings.

If a string is too long to fit on one line, you can use a backslash `\` to break it up onto separate lines.

To insert a newline character into the string, so that when the string is printed it will be printed on two different lines, you can use the newline escape sequence '`\n`'.

1.4 Operators

An operator is a special token that performs an operation, such as addition or subtraction, on either one, two, or three operands. Full coverage of operators can be found in a later chapter.

1.5 Separators

A separator separates tokens. White space (see next section) is a separator, but it is not a token. The other separators are all single-character tokens themselves:

```
( ) [ ] { } ; , . :
```

1.6 White Space

White space is the collective term used for several characters: the space character, the tab character, the newline character, the vertical tab character, and the form-feed character. White space is ignored, and is therefore optional, except when it is used to separate tokens.

Although you must use white space to separate many tokens, no white space is required between operators and operands, nor is it required between other separators and that which they separate. Furthermore, wherever one space is allowed, any amount of white space is allowed. In string constants, spaces and tabs are not ignored; rather, they are part of the string.

---

2 Data Types

2.1 Primitive Data Types

2.1.1 Integer Types The integer data types range in size from at least 8 bits to at least 64 bits. You should use integer types for storing whole number values (and the `char` data type for storing characters). The sizes and ranges listed for these types are minimums; depending on your computer platform, these sizes and ranges may be larger.

While these ranges provide a natural ordering, the standard does not require that any two types have a different range. For example, it is common for `int` and `long` to have the same range. The standard even allows `signed char` and `long` to have the same range, though such platforms are very unusual.

- `signed char`
  The 8-bit `signed char` data type can hold integer values in the range of -128 to 127.
- `unsigned char`
  The 8-bit `unsigned char` data type can hold integer values in the range of 0 to 255.

- `char`
  Depending on your system, the `char` data type is defined as having the same range as either the `signed char` or the `unsigned char` data type (they are three distinct types, however). By convention, you should use the `char` data type specifically for storing ASCII characters (such as `` `m' ``), including escape sequences (such as `` `\n' ``).
- `short int`
  The 16-bit `short int` data type can hold integer values in the range of -32,768 to 32,767.
- `unsigned short int`
  The 16-bit `unsigned short int` data type can hold integer values in the range of 0 to 65,535.
- `int`
  The 32-bit `int` data type can hold integer values in the range of -2,147,483,648 to 2,147,483,647.
- `unsigned int`
  The 32-bit `unsigned int` data type can hold integer values in the range of 0 to 4,2.2.967,295.
- `long int`
  The 32-bit `long int` data type can hold integer values in the range of at least -2,147,483,648 to 2,147,483,647. .
- `unsigned long int`
  The 32-bit `unsigned long int` data type can hold integer values in the range of at least 0 to 4,2.2.967,295.
- `long long int`
  The 64-bit `long long int` data type can hold integer values in the range of -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
  The 64-bit `unsigned long long int` data type can hold integer values in the range of at least 0 to 18,446,744,073,709,551,615.

2.1.2 Real Number Types There are three data types that represent fractional numbers.

- `float`
  The `float` data type is the smallest of the three floating point types. Its minimum value is no greater than `1e-37`. Its maximum value is no less than `1e37`.
- `double`
  The `double` data type is at least as large as the `float` type, and it may be larger. Its minimum value is stored in `DBL_MIN`, and its maximum value is stored in `DBL_MAX`.
- `long double`
  The `long double` data type is at least as large as the `float` type, and it may be larger. Its minimum value is stored in `LDBL_MIN`, and its maximum

value is stored in `LDBL_MAX`.

All floating point data types are signed; trying to use `unsigned float`, for example, will cause a compile-time error.

2.2.Structures

A structure is a programmer-defined data type made up of variables of other data types (possibly including other structure types).

2.2.1 Defining Structures You define a structure using the `struct` keyword followed by the declarations of the structure's members, enclosed in braces. You declare each member of a structure just as you would normally declare a variable—using the data type followed by one or more variable names separated by commas, and ending with a semicolon. Then end the structure definition with a semicolon after the closing brace.

You should also include a name for the structure in between the `struct` keyword and the opening brace. This is optional, but if you leave it out, you can't refer to that structure data type later on.

It is possible for a structure type to contain a field which is a pointer to the same type.

2.2.2 Declaring Structure Variables You can declare variables of a structure type when both you initially define the structure and after the definition, provided you gave the structure type a name.

2.2.2.1 Declaring Structure Variables at Definition You can declare variables of a structure type when you define the structure type by putting the variable names after the closing brace of the structure definition, but before the final semicolon. You can declare more than one such variable by separating the names with commas.

2.2.2.2 Declaring Structure Variables After Definition You can declare variables of a structure type after defining the structure by using the `struct` keyword and the name you gave the structure type, followed by variable names separated by commas.

2.2.2.3 Initializing Structure Members You can initialize the members of a structure type to have certain values when you declare structure variables.

If you do not initialize a structure variable, it's, members with integral types are initialized with 0 and pointer members are initialized to NULL.

One way to initialize a structure is to specify the values in a set of braces and separated by commas. Those values are assigned to the structure members in the same order that the members are declared in the structure in definition. Another way to initialize the members is to specify the name of the member to initialize. This way, you can initialize the members in any order you like, and even leave some of them uninitialized.

You can also initialize the structure variable's members when you declare the variable during the structure definition and can also initialize fewer than all of a structure variable's members.

2.2.3 Accessing Structure Members You can access the members of a structure variable using the member access operator (`.`) You put the name of the structure variable on the left side of the operator, and the name of the member on the right side.

2.2.4 Size of Structures The size of a structure type is equal to the sum of the size of all of its members

---

2.5 Arrays

An array is a data structure that lets you store one or more elements consecutively in memory. In C, array elements are indexed beginning at position zero, not one.

2.3.1 Declaring Arrays You declare an array by specifying the data type for its elements, its name, and the number of elements it can store.

For standard C code, the number of elements in an array must be positive.

The number of elements can be as small as zero. Zero-length arrays are useful as the last element of a structure which is really a header for a variable-length object:

C allows you to declare an array size using variables, rather than only constants.

2.3.2 Initializing Arrays You can initialize the elements in an array when you declare it by listing the initializing values, separated by commas, in a set of braces.

You don't have to explicitly initialize all of the array elements.ro:

You can initialize array elements out of order, by specifying which array indices to initialize. To do this, include the array index in brackets, and optionally the assignment operator, before the value.

If you initialize every element of an array, then you do not have to specify its size; its size is determined by the number of elements you initialize.

Alternately, if you specify which elements to initialize, then the size of the array is equal to the highest element number initialized, plus one.

2.3.3 Accessing Array Elements You can access the elements of an array by specifying the array name, followed by the element index, enclosed in brackets. Remember that the array elements are numbered starting with zero.

2.3.4 Arrays as Strings You can use an array of characters to hold a string (see String Constants). The array may be built of either signed or unsigned characters.

When you declare the array, you can specify the number of elements it will have. That number will be the maximum number of characters that should be in the string, including the null character used to end the string. If you choose this option, then you do not have to initialize the array when you declare it. Alternately, you can simply initialize the array to a value, and its size will then be exactly large enough to hold whatever string you used to initialize it.

There are two different ways to initialize the array. You can specify of comma-delimited list of characters enclosed in braces, or you can specify a string literal enclosed in double quotation marks.

The null character `\0` is included at the end of the string, even when not explicitly stated. (Note that if you initialize a string using an array of individual characters, then the null character is *not* guaranteed to be present. It might be, but such an occurrence would be one of chance, and should not be relied upon.)

After initialization, you cannot assign a new string literal to an array using the assignment operator.

However, you can change one character at a time, by accessing individual string elements as you would any other array:

It is possible for you to explicitly state the number of elements in the array, and then initialize it using a string that has more characters than there are elements in the array. This is not a good thing. The larger string will *not* override the previously specified size of the array, and you will get a compile-time warning. Since the original array size remains, any part of the string that exceeds that original size is being written to a memory location that was not allocated for it.

2.3.5 Arrays of Structures You can create an array of a structure type just as you can an array of a primitive data type.

As with initializing structures which contain structure members, the additional inner grouping braces are optional. But, if you use the additional braces, then you can partially initialize some of the structures in the array, and fully initialize others:

After initialization, you can still access the structure members in the array using the member access operator. You put the array name and element number (enclosed in brackets) to the left of the operator, and the member name to the right.

2.4 Pointers

Pointers hold memory addresses of stored constants or variables. For any data type, including both primitive types and custom types, you can create a pointer that holds the memory address of an instance of that type.

2.4.1 Declaring Pointers You declare a pointer by specifying a name for it and a data type. The data type indicates of what type of variable the pointer will hold memory addresses.

To declare a pointer, include the indirection operator (`*`) before the identifier. Here is the general form of a pointer declaration:

```
data-type * name;
```

White space is not significant around the indirection operator. When declaring multiple pointers in the same statement, you must explicitly declare each as a pointer, using the indirection operator:

2.4.2 Initializing Pointers You can initialize a pointer when you first declare it by specifying a variable address to store in it.

Note the use of the address operator, used to get the memory address of a variable. After you declare a pointer, you do *not* use the indirection operator with the pointer's name when assigning it a new address to point to. On the contrary, that would change the value of the variable that the points to, not the value of the pointer itself.

The value stored in a pointer is an integral number: a location within the computer's memory space. If you are so inclined, you can assign pointer values explicitly using literal integers, casting them to the appropriate pointer type. However, we do not recommend this practice unless you need to have extremely fine-tuned control over what is stored in memory, and you know exactly what you are doing. It would be all too easy to accidentally overwrite something that you did not intend to. Most uses of this technique are also non-portable.

It is important to note that if you do not initialize a pointer with the address of some other existing object, it points nowhere in particular and will likely make your program crash if you use it (formally, this kind of thing is called *undefined behavior*).

2.4.4 Pointers to Structures You can create a pointer to a structure type just as you can a pointer to a primitive data type.

You can access the members of a structure variable through a pointer, but you can't use the regular member access operator anymore. Instead, you have to use the indirect member access operator (`->`)

2.5 Incomplete Types

You can define structures without listing their members. Doing so results in an incomplete type. You can't declare variables of incomplete types, but you can work with pointers to those types. At some time later in your program you will want to complete the type. You do this by defining it as you usually would define a struct. This technique is commonly used to for linked lists.

2.6 Type Qualifiers

There is one type qualifier that you can prepend to your variable declarations which change how the variables may be accessed: `const`

`const` causes the variable to be read-only; after initialization, its value may not be changed.

2.7 Renaming Types

Sometimes it is convenient to give a new name to a type. You can do this using the `typedef` statement. See The typedef Statement, for more information.

---

3 Expressions and Operators

3.1 Expressions

An *expression* consists of at least one operand and zero or more operators. Operands are typed objects such as constants, variables, and function calls that return values.

An *operator* specifies an operation to be performed on its operand(s). Operators may have one, two, or three operands, depending on the operator.

3.2 Assignment Operators

Assignment operators store values in variables. C provides several variations of assignment operators. The standard assignment operator `=` simply stores the value of its right operand in the variable specified by its left operand. As with all assignment operators, the left operand cannot be a literal or constant value.

3.3 Incrementing and Decrementing

The increment operator `++` adds 1 to its operand. The operand must be a either a variable of one of the primitive data types or a pointer. You can apply the increment operator either before or after the operand. A prefix increment adds 1 before the operand is evaluated. A postfix increment adds 1 after the operand is evaluated.

Likewise, you can subtract 1 from an operand using the decrement operator `--`. The concepts of prefix and postfix application apply here as with the increment operator.

3.4 Arithmetic Operators

C provides operators for standard arithmetic operations: addition, subtraction, multiplication, and division, along with modular division and negation. (Note

that you can add and subtract memory pointers, but you cannot multiply or divide them)

Integer division of positive values truncates towards zero, so 5/3 is 1. However, if either operand is negative, the direction of rounding is implementation-defined.

You use the modulus operator `%` to obtain the remainder produced by dividing its two operands. The operands must be expressions of a primitive data type.

Modular division returns the remainder produced after performing integer division on the two operands. The operands must be of a primitive integer type.

If the operand you use with the negative operator is of an unsigned data type, then the result cannot negative, but rather is the maximum value of the unsigned data type, minus the value of the operand.

Numeric values are assumed to be positive unless explicitly made negative.

3.5 Comparison Operators

You use the comparison operators to determine how two operands relate to each other: are they equal to each other, is one larger than the other, is one smaller than the other, and so on. When you use any of the comparison operators, the result is either 1 or 0, meaning true or false, respectively.

(In the following code examples, the variables `x` and `y` stand for any two expressions of arithmetic types, or pointers.)

The equal-to operator `==` tests its two operands for equality. The result is 1 if the operands are equal, and 0 if the operands are not equal.

The not-equal-to operator `!=` tests its two operands for inequality. The result is 1 if the operands are not equal, and 0 if the operands *are* equal.

3.6 Logical Operators

Logical operators test the truth value of a pair of operands. Any nonzero expression is considered true in C, while an expression that evaluates to zero is considered false.

The logical conjunction operator `&&` tests if two expressions are both true. If the first expression is false, then the second expression is not evaluated.

The logical disjunction operator `||` tests if at least one of two expressions it true. If the first expression is true, then the second expression is not evaluated.

You can prepend a logical expression with a negation operator `!` to flip the truth value.

3.7 Bit Shifting

You use the left-shift operator `<<` to shift its first operand's bits to the left. The second operand denotes the number of bit places to shift. Bits shifted off the left side of the value are discarded; new bits added on the right side will all be 0.

Similarly, you use the right-shift operator `>>` to shift its first operand's bits to the right. Bits shifted off the right side are discarded; new bits added on the left side are *usually* 0, but if the first operand is a signed negative value, then the added bits will be either 0 *or* whatever value was previously in the leftmost bit position.

For both `<<` and `>>`, if the second operand is greater than the bit-width of the first operand, or the second operand is negative, the behavior is undefined.

3.8 Bitwise Logical Operators

C provides operators for performing bitwise conjunction, inclusive disjunction, exclusive disjunction, and negation (complement).

Bitwise conjunction examines each bit in its two operands, and when two corresponding bits are both 1, the resulting bit is 1. All other resulting bits are 0.

Bitwise inclusive disjunction examines each bit in its two operands, and when two corresponding bits are both 0, the resulting bit is 0. All other resulting bits are 1.

Bitwise exclusive disjunction examines each bit in its two operands, and when two corresponding bits are different, the resulting bit is 1. All other resulting bits are 0.

Bitwise negation reverses each bit in its operand.

3.9 Pointer Operators

You can use the address operator `&` to obtain the memory address of an object.

Function pointers and data pointers are not compatible, in the sense that you cannot expect to store the address of a function into a data pointer, and then copy that into a function pointer and call it successfully.

Given a memory address stored in a pointer, you can use the indirection operator `*` to obtain the value stored at the address. (This is called *dereferencing* the pointer.)

Avoid using dereferencing pointers that have not been initialized to a known memory location.

3.10 The sizeof Operator

You can use the `sizeof` operator to obtain the size (in bytes) of the data type of its operand. The operand may be an actual type specifier (such as `int` or `float`), as well as any valid expression. When the operand is a type name, it must be enclosed in parentheses.

The result of the `sizeof` operator is perhaps identical to `unsigned int` or `unsigned long int`; it varies from system to system.

3.11 Type Casts

You can use a type cast to explicitly cause an expression to be of a specified data type. A type cast consists of a type specifier enclosed in parentheses, followed by an expression. To ensure proper casting, you should also enclose the expression that follows the type specifier in parentheses.

3.12 Array Subscripts

You can access array elements by specifying the name of the array, and the array subscript (or index, or element number) enclosed in brackets.

3.13 Function Calls as Expressions

A call to any function which returns a value is an expression.

3.14 The Comma Operator

You use the comma operator `,` to separate two expressions. For instance, the first expression might produce a value that is used by the second expression. More commonly, the comma operator is used in `for` statements. This lets you conveniently set, monitor, and modify multiple control expressions for the `for` statement.

A comma is also used to separate function parameters; however, this is *not* the comma operator in action. In fact, if the comma operator is used as we have discussed here in a function call, then the compiler will interpret that as calling the function with an extra parameter.

3.15 Member Access Expressions

You can use the member access operator `.` to access the members of a structure. You put the name of the structure variable on the left side of the operator, and the name of the member on the right side.

You can also access the members of a structure or union variable via a pointer by using the indirect member access operator `->`. `x->y` is equivalent to `(*x).y`.

3.16 Conditional Expressions

You use the conditional operator to cause the entire conditional expression to evaluate to either its second or its third operand, based on the truth value of its first operand.

3.17 Statements and Declarations in Expressions

As a GNU C extension, you can build an expression using compound statement enclosed in parentheses. This allows you to included loops, switches, and local variables within an expression.

If you don't know the type of the operand, you can still do this, but you must use `typeof` expressions or type naming.

3.18 Operator Precedence

When an expression contains multiple operators, such as `a + b * f()`, the operators are grouped based on rules of *precedence*. For instance, the meaning of that expression is to call the function `f` with no arguments, multiply the result by `b`, then add that result to `a`. That's what the C rules of operator precedence determine for this expression.

The following is a list of types of expressions, presented in order of highest precedence first. Sometimes two or more operators have equal precedence; all those operators are applied from left to right unless stated otherwise.

1. Function calls, array subscripting, and membership access operator expressions.
2. Unary operators, including logical negation, bitwise complement, increment, decrement, unary positive, unary negative, indirection operator, address operator, type casting, and `sizeof` expressions. When several unary

operators are consecutive, the later ones are nested within the earlier ones: `!-x` means `!(-x)`.

3. Multiplication, division, and modular division expressions.
4. Addition and subtraction expressions.
5. Bitwise shifting expressions.
6. Greater-than, less-than, greater-than-or-equal-to, and less-than-or-equal-to expressions.
7. Equal-to and not-equal-to expressions.
8. Bitwise AND expressions.
9. Bitwise exclusive OR expressions.
10. Bitwise inclusive OR expressions.
11. Logical AND expressions.
12. Logical OR expressions.
13. Conditional expressions. When used as subexpressions, these are evaluated right to left.
14. All assignment expressions. When multiple assignment statements appear as subexpressions in a single larger expression, they are evaluated right to left.
15. Comma operator expressions.

---

4 Statements

You write statements to cause actions and to control flow within your programs. You can also write statements that do not do anything at all, or do things that are uselessly trivial.

4.1 Labels

Labels are allowed to be used identify a section of source code for use with a later `goto` (see The goto Statement). A label consists of an identifier followed by a colon.

Label names should not interfere with other identifier names. The ISO C standard mandates that a label must be followed by at least one statement, possibly a null statemen otherwise undefined behavior shall occur in the compiler.

4.2 Expression Statements

Any expression can be turned into a statement by adding a semicolon to the end of the expression.

4.3 The `if` Statement

`if` statement is used to conditionally execute part of your program, based on the truth value of a given expression.

If `test` evaluates to true, then `then-statement` is executed and `else-statement` is not. If `test` evaluates to false, then `else-statement` is executed and `then-statement` is not. The `else` clause is optional.

You can use a series of `if` statements to test for multiple conditions.

Unlike GNU C99, for our subset of c the statement(s) compulsorily have to be nexessarily in blocks using braces '{' at the beginning and '}' at the end for correct evaluation by our compiler.

4.4 The `switch` Statement

The `switch` statement is used to compare one expression with others, and then execute a series of sub-statements based on the result of the comparisons. Here is the general form of a `switch` statement:

```
switch (test)
  {
    case compare-1:
      if-equal-statement-1
    case compare-2:
      if-equal-statement-2
    …
    default:
      default-statement
  }
```

The `switch` statement compares `test` to each of the `compare` expressions, until it finds one that is equal to `test`. Then, the statements following the successful case are executed. All of the expressions compared must be of an integer type, and the `compare-N` expressions must be of a constant integer type (e.g., a literal integer or an expression built of literal integers).

Optionally, you can specify a default case. If `test` doesn't match any of the specific cases listed prior to the default case, then the statements for the default case are executed. Traditionally, the default case is put after the specific cases, but that isn't required.

4.5 The `while` Statement

The `while` statement is a loop statement with an exit test at the beginning of the loop. Here is the general form of the `while` statement:

```
while (test)
  {statement(s)}
```

Unlike GNU C99, for our subset of c, the statement(s) following `while(test)` compulsorily have to be nexessarily in blocks using braces '{' at the beginning and '}' at the end for correct evaluation by our compiler, just as we did above for if.

The `while` statement first evaluates `test`. If `test` evaluates to true, `statement` is executed, and then `test` is evaluated again. `statement` continues to execute repeatedly as long as `test` is true after each execution of `statement`.

A `break` statement can also cause a `while` loop to exit.

4.6 The `do` Statement

The `do` statement is a loop statement with an exit test at the end of the loop. Here is the general form of the `do` statement:

```
do
  {statement(s)}
while (test);
```

Unlike GNU C99, for our subset of c, the statement(s) following `do` compulsorily have to be nexessarily in blocks using braces '{' at the beginning and '}' at the end for correct evaluation by our compiler, just as we did above for while.

The `do` statement first executes `statement`. After that, it evaluates `test`. If `test` is true, then `statement` is executed again. `statement` continues to execute repeatedly as long as `test` is true after each execution of `statement`.

A `break` statement can also cause a `do` loop to exit.

4.7 The `for` Statement

The `for` statement is a loop statement whose structure allows easy variable initialization, expression testing, and variable modification. It is very convenient for making counter-controlled loops. Here is the general form of the `for` statement:

```
for (initialize; test; step)
  {statement}
```

Unlike GNU C99, for our subset of c, the statement(s) following `for (initialize; test; step)` compulsorily have to be nexessarily in blocks using braces '{' at the beginning and '}' at the end for correct evaluation by our compiler, just as we did above for if.

The `for` statement first evaluates the expression `initialize`. Then it evaluates the expression `test`. If `test` is false, then the loop ends and program control resumes after `statement`. Otherwise, if `test` is true, then `statement` is executed. Finally, `step` is evaluated, and the next iteration of the loop begins with evaluating `test` again.

Most often, `initialize` assigns values to one or more variables, which are generally used as counters, `test` compares those variables to a predefined expression, and `step` modifies those variables' values.

All three of the expressions in a `for` statement are optional, and any combination of the three is valid. Since the first expression is evaluated only once, it is perhaps the most commonly omitted expression.

If you leave out the `test` expression, then the `for` statement is an infinite loop (unless you put a `break` or `goto` statement somewhere in `statement`). This is like using `1` as `test`; it is never false.

Usage of the comma operator confusingly is not allowed (see The Comma Operator) for monitoring multiple variables in a `for` statement, because as usual the comma operator discards the result of its left operand.

If you need to test two conditions, you will need to use the `&&` operator.

A `break` statement can also cause a `for` loop to exit.

4.8 Blocks

A *block* is a set of zero or more statements enclosed in braces. Blocks are also known as *compound statements*. A block is used as the body of an `if` statement or a loop statement, to group statements together. You can also put blocks inside other blocks.

Declarations of variables are allowed inside a block; such variables are local to that block (details described in scope of a variable). In C89, declarations must occur before other statements, and so sometimes it is useful to introduce a block simply for this purpose:

4.9 The Null Statement

The *null statement* is merely a semicolon alone.

A null statement does not do anything. It does not store a value anywhere. It does not cause time to pass during the execution of your program. Most often, a null statement is used as the body of a loop statement, or as one or more of the expressions in a `for` statement. A null statement is also sometimes used to follow a label that would otherwise be the last thing in a block.

4.10 The `goto` Statement

Use the `goto` statement to unconditionally jump to a different place in the program.

Specifying a label to jump to is necessary; when the `goto` statement is executed, program control jumps to that label. See Labels.

The label can be anywhere in the same function as the `goto` statement that jumps to it, but a `goto` statement cannot jump to a label in a different function. **Usage of goto is not recommended in case of loops as this can cause program to not execute properly. If possible usage of for, while and do while loop constructs are recommended.**

4.11 The `break` Statement

You can use the `break` statement to terminate a `while`, `do`, `for`, or `switch` statement. If you put a `break` statement inside of a loop or `switch` statement which itself is inside of a loop or `switch` statement, the `break` only terminates the innermost loop or `switch` statement.

4.12 The `continue` Statement

Use the `continue` statement in loops to terminate an iteration of the loop and begin the next iteration. If you put a `continue` statement inside a loop which itself is inside a loop, then it affects only the innermost loop.

4.13 The `return` Statement

You can use the `return` statement to end the execution of a function and return program control to the function that called it.

`return-value` is an optional expression to return. If the function's return type is `void`, then it is invalid to return an expression. You can, however, use the `return` statement without a return value.

If the function's return type is not the same as the type of `return-value`, and automatic type conversion cannot be performed, then returning `return-value` is invalid.

If the function's return type is not `void` then return value *must* be specified.

4.14 The `typedef` Statement

Use the `typedef` statement to create new names for data types. Here is the general form of the `typedef` statement:

`old-type-name` is the existing name for the type, and may consist of more than one token (e.g., `unsigned long int`). `new-type-name` is the resulting new name for the type, and must be a single identifier. Creating this new name for the type does not cause the old name to cease to exist. Here are some examples:

To make a type definition of an array, you first provide the type of the element, and then establish the number of elements at the end of the type definition.

When selecting names for types, you should avoid ending your type names with a `_t` suffix. The compiler will allow you to do this, but the POSIX standard reserves use of the `_t` suffix for standard library type names.

---

5 Functions

You can write functions to separate parts of your program into distinct subprocedures. To write a function, you must at least create a function definition. Every program requires at least one function, called `main`. That is where the program's execution begins.

5.1 Function Declarations and Definitions

You write a function declaration **followed** by function definition to specify the name of a function, a list of parameters, the function's return type, and to specify what a function actually does.

Here is the general form of a function:

```
return-type function-name (parameter-list){
  function-body
}
```

`return-type` indicates the data type of the value returned by the function. You can declare a function that doesn't return anything by using the return type `void`.

`function-name` can be any valid identifier (see Identifiers).

`parameter-list` consists of zero or more parameters, separated by commas. A parameter consists of a data type and an name for the parameter.

The parameter names can be any identifier, and if you have more than one parameter, you can't use the same name more than once within a single decla-

ration. You can put it in a header file and use the `#include` directive to include that function declaration in any source code files that use the function.

`The function body` is a series of statements enclosed in braces.

5.2 Calling Functions

You can call a function by using its name and supplying any needed parameters. A function call can make up an entire statement, or it can be used as a subexpression. If a parameter takes more than one argument, you separate parameters with commas.

5.3 Function Parameters

Function parameters can be any expression, a literal value, a value stored in variable, an address in memory, or a more complex expression built by combining these.

Within the function body, the parameter is a local copy of the value passed into the function; you cannot change the value passed in by changing the local copy. If the value that you pass to a function is a memory address, then you can access (and change) the data stored at the memory address. This achieves an effect similar to pass-by-reference, but the memory address itself cannot be changed.

Currently we only support 1D type of array as parameter.

5.4 The `main` Function

Every program requires at least one function, called '`main`'. This is where the program begins executing. The return type for `main` is always `int`. Reaching the `}` at the end of `main` without a return, or executing a `return` statement with no value (that is, `return;`) are both equivalent. The effect of this is equivalent to `return 0;`.

You can write your `main` function that accept parameters from the command line. To accept command line parameters, you need to have two parameters in the `main` function, `int argc` followed by `char *argv[]`. They must have those data types—`int` and array of pointers to `char`. `argv[0]`, the first element in the array, is the name of the program as typed at the command line.

5.5 Recursive Functions

You can write a function that is recursive — a function that calls itself.

---

6 Program Structure and Scope

6.1 Program Structure (#program-structure .section)

A C program may exist entirely within a single source file, but more commonly, will consist of several custom header files and source files, and will also include and link with files from existing libraries. By convention, header files contain variable and function declarations, and source files contain the corresponding definitions.

6.2 Scope

A declared object can be visible only within a particular function, or within a particular file. Unless explicitly stated otherwise, declarations made at the top-level of a file (i.e., not within a function) are visible to the entire file, but are not visible outside of the file.

---

7 Implementation of Advanced Features

This section shall contain details of various advanced features apart from the above mentioned basic features we are going to implement through our compiler along with some of the code optimizations

---

7.1 Dead Code Elimination

This section deals with code optimization through the process of dead code elimination. If the compiler can obtain a section of code which can never be executed or does not modify the code results, it shall remove the section of that code thus decreasing the length of mips file and increasing the speed of execution.

7.2 Short Circuit Evaluation

This is also used to increase the speed of execution of the code. In this the later arguments are only evaluated if after the evaluation of all the arguments until now is not sufficient to evaluate the value of the entire expression.

7.3 Dynamic Memory Allocation

We allow dynamic memory allocation of various size as per requirement which is not known during compile time like `malloc, calloc` functions. Freeing of these dynamically allocated memory during runtime is done using `free` function.

Here, the malloc function dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It doesn't Initialize memory at execution time so that it has initialized each block with the default garbage value initially.

The calloc dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It initialize memory with default value zero initially. It takes two arguments the number of elements and size of each element.

7.4 File Handling

We extend the basic C99 to include file IO using functions such as `fopen, flcose, fseek, fwrite, fread` for opening, closing, pointing at a specific location in the file, writing to a file, reading from a file respectively.

7.5 Library Functions

We implement various library functions to provide additional mathematical functions, string manipulation functions such as `strchr(),strlwr(),strupr()` to provide additional functionalities.

This list is not exhaustive and further addition or elimination shall be made based on the time remaining as we go along with this project.

—-

References

https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html