COMS W4115: Programming Languages and Translators Spring 2021

C ♭ Language Reference Manual

Submitted to: Professor Stephen A. Edwards Submitted by: Wednesday, February 24, 2021

1. Introduction and Manual Overview

C ♭ is a computer language designed for musical composition. The language is fundamentally built upon and inspired by rudimentary elements of music theory. C ♭ is a novel, but also chimeric language with many new features, explained extensively in the following sections. This manual describes many data types and methods that are especially unique to the musicality of C ♭. However, users with experience programming in different computer languages — namely, Python, Java, and C — will also recognize many syntactical, organizational, and functional aspects of C ♭ from other computer language as well. The C ♭ Language Reference Manual offers examples, explanations, and additional insight and clarity about the key types, terms, uses, conventions, procedures, and routines that make up the inner workings of the C ♭ language. (This C ♭ Language Reference Manual itself was largely based off of the C Reference Manual; full credit to Brian W. Kernighan and Dennis M. Ritchie is acknowledged in full.)

2. Lexical Conventions

2.1. Comments

Any sequence of characters that begin with the characters "`(:`" and end with the characters "`:)`" is considered a comment. Comments may extend over any number of lines in the code.

2.2. Identifiers

Identifiers are simply sequences of characters. As a rule, the first character of an identifier must be alphabetic or the "_" character in order to be a valid identifier. Identifiers are also case-sensitive. The purpose of an identifier in a C ♭ program is to identify a function or some constant.

2.3. Keywords

All sequences of characters that are shown below are reserved with a specified significance in the C ♭ language, and may never be used as identifiers.

`A`	`B`	`C`	`D`	`E`	`F`	`G`
`note`	`measure`	`float`	`bool`	`int`	`void`	`none`
`continue`	`return`	`break`	`while`	`else`	`true`	`false`
`for`	`def`	`do`	`in`	`if`	`R`	`_DEFAULT_BEATS_`

2.4. Literals

2.4.1. Integer Literals

Integer literals are sequences of numeric symbols treated as decimal digits. Integers in C ♭ are 2 bytes. Both positive and negative integers are allowed from -32,768 to 32,767. Integer literals are defined by the following regular expression: `['0' - '9']+`.

2.4.2. Floating Point Literals

Floating point literals consist of an integer part, a decimal character, and a fractional part. The integer part and fractional parts are observed as sequences of numeric symbols, like integers. No part of the floating point literal may be exempt from declaration. Floating point literals are always computed single-precision, meaning 4 bytes are used, and are defined by the following regular expression: `['0' - '9']+ '.' ['0' - '9']+`.

2.4.3. Character and String Literals

Character literals are recognized in the C ♭ language, enclosed in single quotation marks "`'`". Generally, character literals are single, 2 byte ASCII characters, represented by their printable form. However, character literals may also be two characters — in which case the first character must be a back-slash "\". String literals are recognized in the C ♭ language, enclosed in double quotation marks "`"`". String literals may be of any length, meaning they may contain any number of characters. Many common-practice escape keys are also supported in C ♭, including the following: backspace is accomplished as "`\b`", a newline is "`\n`", tab is "`\t`", back-slash itself is "`\\`", "`\r`" is a carriage return, "`\0`" is a null character, and the double quotation mark itself is "`"`".

2.4.4. Boolean Literals

The two boolean literal constants as supported and represented in C ♭ are "`false`" and "`true`" — and nothing else.

3. Manual Notation

The current section describes fonts and notation used in this manual, titled "C ♭ Language Reference Manual". In this manual, sections are named in bold and numbered in dotted-decimal format for convenient referencing. Furthermore, nested sections generally refer to corresponding belonging and topical relevance. Throughout the manual, the `Courier New` font is used to refer to specific / explicit sequences of characters that will be used in the C ♭ language, in respective scenarios. Double or single quotation marks around the `Courier New` text simply gives added emphasis on the actual spelling-out of each character itself.

The *italics* font is used to describe more generic syntactic categories. This is especially relevant when explanations benefit from more general references to concepts, and not explicitly the exact term used to describe a specific concept in the parser itself.

4. Types

In the C♭ language, two main characteristics of any identifier are to be understood at all times — that is, the scope of the identifier and the type of the identifier. Type must be stated explicitly by the user upon the first initialization / declaration. As a rule, identifiers must be either initialized or declared exactly once before assignment or any other use.

There are eight types supported by C♭ are as follows: integer, float, character, string, boolean, note, measure, and array. Out of the eight types, five — integer, float, character, string, and boolean — are well-established in the programming world. Note and measure are novel, compound, built-in data types in C♭. They are explained at great lengths in this section, along with array, which are a composite data type. In addition, two more special types — none and void — are described in this section as well.

4.1. *Simple Types*

These five basic data types — integer, float, character, string, and boolean — are all alike in that identifiers of these data types may only be assigned to a value from their respective set of constant literals, as described in section 2.4. They are also the only non-composite data types. As such, these five data types are referred to as the *simple types*. This is because they all follow conventional assignment syntax. (By contrast, *note* and *measure* do not follow conventional assignment syntax.)

Specific keywords are used to refer to these *simple types* — `int`, `float`, `char`, `string`, and `bool` refer to integer, float, character, string, boolean respectively.

4.2. Note

In the C♭ language, every note is defined by its three attributes — tone, octave, and rhythm (the duration of the pitch). The value of each note's tone, octave, and rhythm must be explicitly stated during the initialization of every note. Upon declaration (see `note x;` below), the note takes on default values for tone, octave, and rhythm. Notes are a composite data type, in that they can hold more than one value — in this case, of different types. The following lines show examples of how notes are initialized and declared in C♭:

```
(: syntax: note <note_identifier> = (<tone> <octave> <rhythm>); :)
note a = (C- 0 q);        (: Declaring a note with a flat :)
note b = (C+ 0 q);        (: Declaring a note with a sharp :)
note c = (C. 0 q);        (: Declaring a natural note :)
note c = (C 0 q);         (: Also declaring a natural note (same as above) :)
note cc = (C +1 q);       (: Declaring same note as above, but one octave higher :)
note ccc = (C -2 q);      (: Declaring same note as above, but two octaves lower :)
note d = (R 0 q);         (: Declaring a rest :)
note x;                   (: This note defaults to: note x = (C- 0 q); :)
```

4.2.1. Tone

There are 12 tones in a single Western musical octave, represented by a letter (C, D, E, F, G, A, B) and an accidental (♮♭♯). The 12 tones evenly divide an octave and are each separated by a half-step. The C♭ language obeys this convention. To represent a tone, letter names are modified by an accidental. The ♮ (natural symbol) indicates the natural pitch of the letter. The ♯ (sharp symbol) raises the natural letter pitch by a half-step, and the

♭ (flat symbol) lowers the natural letter pitch by a half-step. The C♭ language utilizes capital letters A through G to represent tones, and “`+`” to represent a sharp, “`-`” to represent a flat, and “`.`” to represent a natural. The absence of any accidental assumes a natural, unless a sharp/flat has been explicitly stated before it, for the same note in the same measure.

A note is a data type. The tone of a note is a specific attribute of the note data type. Tones of notes may only be of type string, and naturally, all legal operations that may be performed on strings may also be performed on the tone of a note — for now.

4.2.2. Octave

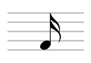
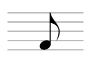
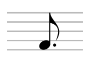
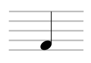
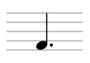
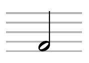
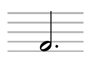
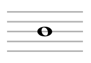
The octave value acknowledges which octave a tone belongs to, which is important because there are theoretically infinitely many notes for each of the 12 tones. The C♭ language defines the octave value of middle C as `0`. The first octave above middle C has an octave value of `+1`. The second octave above would have an octave value of `+2`. And the first octave below middle C would have an octave value of `-1`, and the second octave below would have an octave value of `-2`. This pattern continues in both directions.

The octave of a note is a specific attribute of the note data type. Octaves of notes may only be of type int, and naturally, all legal operations that may be performed on an int literal may also be performed on the octave of a note — for now.

4.2.3. Rhythm

The rhythm of a note is a specific attribute of the note data type. Rhythms of notes may only be of type float, and naturally, all legal operations that may be performed on a float literal may also be performed on the rhythm of a note. Rhythms of notes must be positive floats — for now.

A note requires a rhythm value to indicate the amount of time a note is held. The C♭ language strays from conventional music theory in that it does not regard traditional time signatures in the same way. (Instead, the programmer may be liberal with measure “size” and tempo. See section 3.3.) In C♭, users are able to assign the rhythm of a note from a theoretically infinite set of positive floats, as there is an infinite set of legitimate positive single-precision floats. However, in music composition, composers often resort to a far smaller set of rhythmic values for notes, whose rhythmic floating point values are shown in the “Number of Beats” row in the table below. Strictly speaking, since rhythms of notes are of type float, the rhythmic value of a note must be explicitly stated as a float. (However, a standard library may be used such that a specific, legal string may be used to refer to a specific number of beats instead — strictly for the convenience of the programmer. Each string, shown in the “Letter Syntax” row below, is associated with a set number of beats.)

Rhythm								
Number of Beats	`0.25`	`0.5`	`0.75`	`1.0`	`1.5`	`2.0`	`3.0`	`4.0`
Letter Syntax	`s`	`e`	`e.`	`q`	`q.`	`h`	`h.`	`w`

4.2.4. Rest

Rests are special notes whose rhythm attribute is specified in the same way as any note. The rest tone is represented as `R`. Applying an accidental `+/-/.` does not affect the rest. Moreover, the octave of a rest is `0` by convention, but this also does not affect the rest.

A rest is simply the absence of a played tone for some beats. Rests mean silence.

4.2.5. Methods for Notes

Additionally, the note data-type in the C♭ language possesses built-in methods which are shown below. These methods may be used to access or reassign attributes of a note. Let `a` be the default note — `note a = (C- 0 q);` — for which the following methods are executed to obtain the following return values:

Get Methods (accesses and returns)

Method	Return Value	Return Type
`a.getTone()`	`C-`	*Type string*
`a.getOct()`	`0`	*Type int*
`a.getRthm()`	`q`	*Type float*

Set Methods (sets and does NOT return)

Method	Subsequent Expected Values
`a.setTone(B-)`	`A.getTone() == B-` *Type string*
`a.setOct(-1)`	`A.getOct() == -1` *Type int*
`a.setRthm(s.)`	`A.getRthm() == s.` *Type float*

(For the Set methods, note that each method takes in type char, int, or float, and corresponding Get methods return the same type. However, as mentioned before, `.getRthm` and `.setRthm` may also take in type string, if the standard library is used.)

4.3. Measure

A measure is a data type that contains notes, but they are inherently different from arrays. Measures have two attributes — beats and voices. The only way to create a measure is for the user to type out every note in it, or declare a default measure of four quarter-note rests. Before declaring or initializing a measure, the `_DEFAULT_BEATS_` must have already been set.

```
(: syntax: _DEFAULT_BEATS_ <int>; measure <measure_identifier> = [<note_1, ...,
note_n>]; :) _DEFAULT_BEATS 4;
(: Declaring a simple 4/4 measure with 4 quarter notes :)
measure M = [(C- 0 q), (C- 0 q), (C- 0 q), (C- 0 q)];
(: The following measure defaults to 4 beats of quarter note rests :)
```

```
measure X;
```

4.3.1. Beats

Beats is of type integer, and must be a positive integer. The beats attribute of a measure must have already been initialized before the measure is created. (The beats attribute is initialized using the `_DEFAULT_BEATS_` keyword.) Beats is akin to a measure's "size"; it constrains the total number of beats allowed in the measure. In other words, the sum of all the rhythms of the notes in a measure — these are floats — must add up to the beats of a measure. However, if the inputted sum of rhythms in a measure is less than the measure's time signature, the remaining beats are filled with rests by default. Also, if the sum of all the rhythms of the notes attempted to be added to a measure is greater than the beats of the measure, this will also trigger an error.

4.3.2. `_DEFAULT_BEATS_`

An example of using the `_DEFAULT_BEATS_` keyword is shown below. It must be declared globally, outside of any function, and followed by a positive integer, which becomes the beats value of all subsequent measures. Before any declaration or initialization of a measure, the `_DEFAULT_BEATS_` must have already been set. The `_DEFAULT_BEATS_` is used to set the beats of all measures declared, initialized, assigned, or modified after the statement with `_DEFAULT_BEATS_`. The only way to override the default beats determined by `_DEFAULT_BEATS_` is to write another complete `_DEFAULT_BEATS_` statement, or to use the `.setBeats` method for measures.

```
(: syntax: "_DEFAULT_BEATS_ <beats>;" :)
_DEFAULT_BEATS 3;         (: Declaring the default beat as 3 :)
measure A;                (: Defaults to a 3/4 measure of 3 quarter note rests :)
_DEFAULT_BEATS 2;         (: Declaring the default beat as 3 :)
measure B;                (: Defaults to a 2/4 measure of 2 quarter note rests :)
```

4.3.3. Voice

Voices are an optional attribute of measures, but completely necessary in the case that a user wants to play more than one not at one time, at any point. They are of type `measure[]` . The voices of a measure allow for multiple notes to be played in a measure simultaneously, enabling musicality such as harmonies. Voices abide by the same rules as measure themselves, and their usage is dictated by the following methods. Users may utilize the `.addVoice` or `.setVoice` method to add/assign a voice to a measure at a certain index. By default, the initial content of a measure is the first voice. Note that voices are not a unique data-type, only another (more extensive / scalable) attribute of a measure.

4.3.4. Methods for Measures

As a rule, before measures may be reassigned, measures must have been declared or initialized. Users may add, edit, or delete initialized notes from a measure using built-in methods for the measure data type. If the user attempts to add, edit, or delete absent notes in a measure, an error will be triggered.

The measure data-type in the C ♭ language possesses built-in methods which are shown at the top of the following page. Let `M` be the `measure M` from page 5, above:

Method	Return Type
`M.setBeats(int b)`	returns None (in place modifier sets beats to b)
`M.setNote(int n, string a, ??? x)`	*returns set note: type note
`M.getNote(int i)`	returns i-th note: type note
`M.setVoice(int j, measure m)`	returns None (in place modifier sets the j-th voice to m)
`M.getVoice(int k)`	returns k-th voice: type measure
`M.addVoice(measure m)`	returns None (in place modifier adds a new default voice)
`M.numVoices()`	returns number of voices in measure: type int
`M.numBeats()`	returns time signature of measure: type int

(*The starred method `.setNote` takes in three parameters. The first is of type integer in order to specify the, in this case, n-th note in the measure. Then, the next parameter must be of type string and equivalent to either `"tone"`, `"octave"`, or `"rhythm"` in order to specify the attribute of the note at index. Then, the final parameter must be of type `string` if the string `a` was equivalent to `"tone"`, type `int` if the string `a` was equivalent to `"octave"`, and type `float` if the string `a` was equivalent to `"rhythm"`.)

4.4. Array

The array data type is another composite / compound data type. It stores an ordered and indexed sequence of any number of objects, as long as all the objects are all of the same type. Depending on the type of the objects inside the array, the declaration of an array is different. For example, using an arbitrary literal `A`, the following are the only valid array declarations in C ♭ — “`int[] A;`”, “`float[] A;`”, “`char[] A;`”, “`string[] A;`”, “`bool[] A;`”, “`note[] A;`”, and “`measure[] A;`”.

4.4.1. Methods for Arrays

The measure data-type in the C ♭ language possesses built-in methods which are shown below. For example, let `A` be some array `float[] A = [0.0, 0.0];`.

Method	Return Type
`A.setSize(3)`	`A.getSize() == 3` *`Type int`*
`A.getSize()`	`A == [0.0, 0.0, 0.0]` *`Type float[]`*

4.5. None

Although none is a legitimate data type, only `none` can be of the none data type. Thus, all do not have to be declared before use. None can be a type that is returned by a function.

4.6. Void

Although void is a legitimate data type, only `void` can be of the void data type. Thus, all do not have to be declared before use. Void is specified as the return type of a function if the function has no *return statement* in its *body* (*statement list*).

4.7. Conclusion and Summary of Types

The following table summarizes many important characteristics of the above discussed data types in a way that compares many of their commonalities and differences.

“Array-able” means this data type can be put into an array. “Return-able” means this data type can be returned by a function. “Declare-able” means this data type can be declared, and the `Courier New` text demonstrates declaration. The comment shows the default value that the declared data type takes on. This default value is especially important, because this value will fill a `.setSize()`’ed array of this data type, and the value taken upon declaration without initialization. The “Composite / Compound” column contains additional information regarding the items that made the data type compound, shown in parentheses. Non-formatted words are simply adjectives to give further insight on the compounded items. Italicized words describe the data type of the compounded items. Underlined words delineate the names of compounded attribute items of the data type, if they are named.

Summary of Built-In Data Types in C ♭

	Declare-able	Simple	Composite / Compound	Return-able	Array-able

Integers	✔
	`int i;`
	`(: 0 :)`
		✔	--	✔	✔

Floats	✔
	`float f;`
	`(: 0.0 :)`
		✔	--	✔	✔

Characters	✔
	`char c;`
	`(: '\0' :)`
		✔	--	✔	✔

Strings	✔
	`string s;`
	`(: "\0" :)`
		--	✔
			(indexable *char*’s)
				✔	✔

Booleans	✔
	`bool b;`
	`(: false :)`
		✔	--	✔	✔

Note	✔
	`note n;`
	`(: (C 0 0.25) :)`
		--	✔
			(*string* <u>tone</u>, *int* <u>octave</u>, *float* <u>rhythm</u>)
				✔	✔

***Measure**	✔
	`_SET_DEFAULT_ 4;`
	`measure m;`
	`(: [R, R, R, R]`
	`:)`
		--	✔
			(indexable *notes*, *int* <u>beats</u>, *measure[]* <u>voices</u>)
				✔	✔

****Array**	✔
	`???[] a;`
	`(: [] OR [???] :)`
		--	✔
			(indexable *???*’s)
				✔	--

None	--	✔	--	✔	--
Void	--	✔	--	✔	--

(*The measure data type requires that a `_SET_DEFAULT_` statement took place prior to the measure declaration. Additionally, the default value of a measure is `[R, R, R, R]` where `R` is equal to the note `note R = [R 0 4];`.) (**The default values for array types correspond to the specific data type that the array itself contains. An array of the default values of the array elements' data types populate the array if the `.setSize` method is called upon the array. If the array has only been declared and has no size, the array is simply equal to `[]`.)

5. Expressions

5.1. Primary Expressions

Primary expressions consist of identifiers, constants, strings, note expressions, measure expressions, parenthesized expressions, and function expressions.

5.1.1. Identifiers

An identifier (of the token *ID*) is a primary expression, and its type is specified by its declaration.

5.1.2. Constants

The literals mentioned in 2.4 are primary expressions, which the grammar calls *LITERAL*, *FLIT*, *CHARLIT*, *STRLIT*, and *BLIT*.

5.1.3. Arrays

An array literal is a primary expression that the grammar calls *array_expr*. The expressions that array literals contain must be of the same type.

5.1.4. Notes

A note literal is a primary expression that the grammar calls *note_expr* and is of the form (*tone octave rhythm*).

5.1.5. Measures

A measure literal is a primary expression that the grammar calls *measure_expr* and is of the form [*note_expr*, *note_expr*, ...].

5.1.6. (*expression*)

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

5.1.7. Functions

A function expression is a primary expression that the grammar calls *func_expr* and is of the form *ID*(*args_opt)*.

5.1.8. Built-in functions

A built-in function expression is a primary expression that the grammar calls *built_in_func_expr* and is of the form *ID.ID(args_opt)*.

5.2. Unary Operators

C-flat has four unary operators: `-`, `!`, `++`, `--`. The negation operator `-` will negate the `int` or `float` that comes after it. The postfix increment operator `++` adds 1 to its operand; the postfix decrement operator `--` subtracts 1. These postfix operators can be applied to `int`, `float`, and `char` types.

5.3. Binary Operators

The two operands of a binary operation must have the same type. Expressions with binary operators have the form *expression* bin-op *expression*.

5.3.1. Arithmetic Operators

The multiplication operator `*`, division operator `/`, and mod `%` operator have the same precedence and are all of higher precedence than the addition operator `+`, and subtraction operator `-`. Arithmetic operators have higher precedence than relational operators and are left-associative.

5.3.2. Relational Operators

The relational operators `<`, `>`, `<=`, and `>=` return either `true` or `false`. Relational operators have higher precedence than equality operators and are left-associative.

5.3.3. Equality Operators

The equality operators `==` and `!=` are analogous to the relational operators and return either `true` or `false`. Equality operators have higher precedence than boolean operators and are left-associative.

5.3.4. Boolean Operators

The `&&` operator returns `true` if both its operands are `true`, and `false` otherwise. The `||` operator returns `true` if one of its operands are `true`, and `false` otherwise. The `&&` operator has higher precedence than the `||` operator, which has higher precedence than the assignment operator.

5.4. Assignment Expression

The assignment operator `=` groups right-to-left and requires an lvalue as its left operand. The value of an assignment expression is the value stored in the left operand after the assignment.

6. Program Structure

A C♭ programs essentially consists of a list of declarations, among which there must be a function declaration that declares a function named `main`, and `main` must return either an `int` or `void`. Any variable declared inside functions belong to the scope of that function, while variables declared outside functions are treated as global variables. Rules regarding scopes are further discussed in section 9.

The syntax for both variable and function declarations are elaborated in the following section.

7. Declarations

7.1. Variable Declaration

Variable declarations are used to specify the interpretation given to each variable identifier. Variable declarations have the form

type_specifier declarator `;`

We will discuss both of these components in the following subsections.

7.1.1. Type Specifiers

The type specifiers include

```
note
measure
int
```

`float`

`cha`

`bool`

`string`

All of their corresponding types are discussed in detail in section 4.

In addition, to declare an array variable, the type specifier would have the form

element_typ `[]`

where element_typ is the data type of elements contained in the array.

7.1.2. Variable Declarators

The variable declarator has the form

identifier

where identifier is the programmer-defined name of the variable being declared.

Identifiers must belong to the regular set that belongs to the regular expression `['A'-'G' 'R'] ['a'-'z' 'A'-'Z' '0'-'9' '_']+ | ['a'-'z' 'H'-'Q' 'S'-'Z' '_'] ['a'-'z' 'A'-'Z' '0'-'9' '_']*.`

7.1.3. Variable Initialization

C ♭ allows programmers to declare a variable and initialize its value in the same line following the form

type_specifiers declarator `=` expr `;`

where expr must belong to the type indicated by type-specifier.

7.2. Function Declaration

Function declarations in C ♭ follow the form

`def` return_type identifier `(` formals$_{opt}$ `)` `{` body_list `}`

where return_type specified the type of the function’s return value (types that can be a return_type are listed in section 4.7); identifier is the programmer-defined name of the function; formals is the optional list of parameters passed into the function; body_list is a list of statements to be executed in the scope of this function.

8. Statements

8.1. Expression Statement

Most statements are expression statements, which have the form

expression `;`

Usually expression statements are assignments or function calls.

8.2. Conditional Statement

The two forms of the conditional statement are

`if (` expression `)` statement

`if (` expression `)` statement `else` statement

In both cases the expression is evaluated and if it is true, the first substatement is executed. In the second case, the second substatement is executed if the expression is false.

8.3. While Statement

The `while` statement has the form

`while (` expression `)` statement

The substatement is executed repeatedly as long as the value of the expression remains true. The test takes place before each execution of the statement.

8.4. Do Statement

The `do` statement has the form

`do` statement `while (` expression `) ;`

The substatement is executed repeatedly until the value of the expression becomes false. The test takes place after each execution of the statement.

8.5. For Statement

The `for` statement has the form

`for (` expression-1$_{opt}$`;` expression-2$_{opt}$`;` expression-3$_{opt}$ `)` statement

The first expression specifies initialization for the loop; the second specifies a test, made before each iteration such that the loop is exited when the expression becomes false; the third expression typically specifies an incrementation which is performed after each iteration.

8.6. Break Statement

The statement

`break ;`

causes the termination of the smallest enclosing `while`, `do`, or `for` statement; control passes to the statement following the terminated statement.

8.7. Continue Statement

The statement

```
continue ;
```

causes control to pass to the loop-continuation portion (i.e. the end of the loop) of the smallest enclosing `while`, `do`, or `for` statement.

8.8. Return Statement

A function returns to its caller by means of the return statement, which has one of the forms

```
return ;
return expression ;
```

In the first case, no value is returned. In the second case, the value of the expression is returned to the caller of the function.

9. Scope Rules

Variables declared in a function are visible within that function. Variables can also be declared in `do`, `while`, `for`, and conditional statements and have scope within that statement. Moreover, a variable overrides another variable with the same identifier if it has a smaller scope. A variable with wider scope can be accessed from a block with a lower scope level, given that its identifier is unique to that level.

The storage class is never explicitly state by the user; no keywords in the C ♭ language pertain to storage class or scope of an identifier.

Elaborating first on storage classes, the break-down of storage classes and variable lifetimes in C ♭ is extremely simple. A variable is either global to the entire program, or local to only the function in which it is declared, after its declaration. The location in the program of such a declaration immediately assumes this natural differentiation of scope, and storage classes are determined naturally to the identifier’s initialization.

10. Built-in Functions

10.1. The `print` function

The `print` function takes in any type as an argument and prints it to standard output.

10.2. The `play` function

The `play` function takes in *`note`*, *`measure`*, and `measure[]` types as an argument and generates a playable .WAV file. This function renders the music at a default tempo of 120 BPM.

10.3. The `bplay` function

The `bplay` function takes in *`note`*, *`measure`*, and `measure[]` types as the first argument, an `int` specifying the tempo as the second argument, and generates a playable .WAV file at the specified tempo.