Purpose/Features of Beats++:

* Musical Notation has always been written by hand, and still is done so even in the age of computers
* Representing Musical Information in a way which is both intuitive to the user and the machine has never been done before
* Beats++ aims to represent music and Musical Notation in an efficient way, thus opening opportunities for developers and musicians to write better programs more efficiently than by using libraries written in existing languages
* The main purpose of Beats++ is for users to code musical pieces in an operation-friendly way, which can be modified and played back in ways convenient to the user
* One of the few languages which works with file formats like .midi, which are designed for music production, and provides file handling with said formats.
* Since the language is written specifically for music operations, it removes unnecessary programs out of the equation, as opposed to using a library in an existing language
* The basic data types are pitch, duration and integer
* Derived Data Types like Measures, Chords, Arpeggios, and are created using the basic data types
* Includes methods inherent to music theory like Transpose, Arpeggiate and “Chordify?”, thus simplifying the process of composing
* Includes implicit operations like multiplication of the duration of a note, and incrementing and decrementing the pitch of a note.
* Syntactic similarity to C to keep the learning curve in check for newcomers.
* File extension .bpm

Do’s and Dont’s:

* Statically typed language
* Compiled not interpreted to allow for quick runtime execution. This is important because we believe the average “sheet” will be “played” a lot more than “composed”.
* Light, clean syntax that is fast to write
* Efficiency favoured over conciseness as the language targets a specific use case
* Possible multithreaded support for simultaneous playback of multiple instruments
* No explicit scoping
* No global variables
* JavaScript module type imports
* Hard check for redundant (unused) variables and imports at compile time.
* Strict prevention on any uninitialized variables at compilation.
* Functions support return of multiple variables; both derived and primitive data types are passed and returned by value.

**Syntax**

Sheet

Measure//bar (time signature)=> Adjacency list kinda thing. Later

Note\_Units

Notes => Note is a tuple of (pitch,duration)

Minimum Time = 32nd note

Pitch is int

Name is char/string

Duration is time

Later, add something to say how many seconds one duration corresponds to.

import io from ‘io’;

func start[None] returns [None] {

pitch sample = 2;

duration sample\_duration = 4;

<--Initializing the note-->

Note sample\_note1 = (sample,sample\_duration);

<--Another form of initialization-->

Note sample\_note2 = (-10,3);

<-- Simple operations on notes -->

Sample\_note2 ++;

<-- Playing the note through computer audio-->

io.play[sample\_note1];

io.play[sample\_note2];

return [ ];  
}

func initialise\_bar[] returns [Measure M]  
{

Note note1 = (-10,4);

Note note2 = (5,7);

Note note3, note4 = (6,36);

note3 = (69,420);

return [];  
}

func start[None] returns [None] {

Note note1 = (10, 2);

Note note2 = (7, 3);

Note note3 = (9,4);

Note note4 = (12, 1);

<-- A measure is a fixed duration, containing a group of notes , which can overlap with each other -->

<-- Measure takes the total duration as an argument-->

Measure measure1 = new Measure[6];

<-- Can add notes to a measure at different starting durations-->

measure1.add[note1,0];

measure1.add[note2,1];

measure1.add[note3,2];

measure1.add[note4,5];

io.play[measure1];

io.midi\_out[measure1];

return [];

}

Types of io:

* io.play(): for output sound
* io.midi\_out(): writes to a midi file
* io.print(): print to console/text files
* io.scan(): take text input from console/text files

**Stuff for examples/later**

* Operators and precedence[Soumi]
* Logical handling?[Agraj and Nandz]
* Keywords[Agraj and Nandz]
* Casting
* Validity of identifiers[Argaj and Nandz]
* No recursion
* Our loop(python syntax or C) and if/else syntax
* Concatenation [Span, Tarun]
* Addition -> Transpose(Note + note; Note + integer transpose)
  + Note + Note should be checking if duration is same
* Multiplication(Note \* duration(Why tf does this data type exist) or Note \* int)? [Soumi]
* Changing the note pitch to actual frequencies??(language evolution)[Collective decision gotta be taken after 1st set of examples]
* Define the “new”, “till” keyword
* For and conditional =>
* For loops with python syntax
* Appendix section for all the weird operators and their overloading

**Examples ideas**

* Non conforming : Wrong variable names[Agraj]
* More IO library stuff??
* One with functions
* Read a MIDI -> Transpose(Showcase operators) -> Write to MIDI
* Depict functions -> Chordify; Arpeggiate;Scale[Tarun, Spandan]
* Somehow show off the multiplication thing too
* Some kind of sticking to scale thing[Soumi?]

Language Evolution Plox:

Add all “Oh shit, we should have done that earlier” over here:

1. Guide on how to write loops correctly
2. Repetition of bars
3. Sound Synthesis Language?
4. Live Playback?
5. Degree of scales (chord progressions)
6. Pitch to float instead of int for more perfect sounds
7. Predefined notion of a time signature

Roles:  
  
1. Span and Tarun: Functions and similar syntax

2. Nandita: Logical/Keywords

3. Agraj: Identifiers/Keywords

4. Soumi: Operators/Precedence

5. Arsalan:

Appendix:

Intro  
  
Lexical Conventions - Agraj

* Comments
* Tokens
* Identifiers
* Keywords

Constants - Agraj

Meaning of identifiers - Tarun

* Storage (make a keyword called heap; if not used then dynamically allocated)
* Precision/ Bit Allocation

Conversions - Tarun

* Frequency vs. Relative Pitch
* Superimposition, concatenation, etc (measures waala stuff)
* None data type

Expressions - Everyone

* Compiled vs. Interpreted
* Write every fuckall operator you've ever seen

Declarations - Everyone(Tarun - Function declaration)

* Our legendary syntax
* Functions
* Variables
* Function declaration and definition isda same
* Class/Struct Declaration

Statements, External Declarations - Soumi

Scope Linkage - Agraj

* How the import works
* Javascript Style
* When the linking happens (during compilation)

Preprocessing - Soumi

* Macros (#replace)

Grammar?

& some shit from Appendix 1B

Add something about pass by reference/value??

Stuff for tutorial

* Hello world (strings)/ Maybe a jingle?
* One with operators
* For/while loop
* One with #replace
* One with an explicit function
* I/O. Console and/ or MIDI
* Scope example

Introduction

C is a general-purpose programming language. It has been closely associated with the UNIX operating system where it was developed, since both the system and most of the programs that run on it are written in C.

The language, however, is not tied to any one operating system or machine; and although it has been called a ‘‘system programming language’’ because it is useful for writing compilers and operating systems, it has been used equally well to write major programs in many different domains.

Many of the important ideas of C stem from the language BCPL, developed by Martin Richards. The influence of BCPL on

C proceeded indirectly through the language B, which was written by Ken Thompson in 1970 for the first UNIX system on

the DEC PDP-7.

BCPL and B are ‘‘typeless’’ languages.

By contrast, C provides a variety of data types. The fundamental types are characters, and integers and floating point numbers of several sizes. In addition, there is a hierarchy of derived data types created with pointers, arrays, structures and unions. Expressions are formed from operators and operands; any expression, including an assignment or a function call, can be a statement.

Pointers provide for machine-independent address arithmetic. C provides the fundamental control-flow constructions required for well-structured programs: statement grouping, decision

making (if-else), selecting one of a set of possible values (switch), looping with the termination test at the top (while,

for) or at the bottom (do), and early loop exit (break).

Functions may return values of basic types, structures, unions, or pointers. Any function may be called recursively. Local

variables are typically ‘‘automatic’’, or created anew with each invocation. Function definitions may not be nested but

variables may be declared in a block-structured fashion. The functions of a C program may exist in separate source files that

are compiled separately. Variables may be internal to a function, external but known only within a single source file, or

visible to the entire program.

A preprocessing step performs macro substitution on program text, inclusion of other source files, and conditional

compilation.

C is a relatively ‘‘low-level’’ language. This characterization is not pejorative; it simply means that C deals with the same

sort of objects that most computers do, namely characters, numbers, and addresses. These may be combined and moved about

with the arithmetic and logical operators implemented by real machines.

C provides no operations to deal directly with composite objects such as character strings, sets, lists or arrays. There are no

operations that manipulate an entire array or string, although structures may be copied as a unit. The language does not define

any storage allocation facility other than static definition and the stack discipline provided by the local variables of functions;

there is no heap or garbage collection. Finally, C itself provides no input/output facilities; there are no READ or WRITE

statements, and no built-in file access methods. All of these higher-level mechanisms must be provided by explicitly called

functions. Most C implementations have included a reasonably standard collection of such functions.

Similarly, C offers only straightforward, single-thread control flow: tests, loops, grouping, and subprograms, but not

multiprogramming, parallel operations, synchronization, or coroutines.

Although the absence of some of these features may seem like a grave deficiency, (‘‘You mean I have to call a function to

compare two character strings?’’), keeping the language down to modest size has real benefits. Since C is relatively small, it

can be described in small space, and learned quickly. A programmer can reasonably expect to know and understand and

indeed regularly use the entire language.

For many years, the definition of C was the reference manual in the first edition of The C Programming Language. In 1983,

the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of

C. The resulting definition, the ANSI standard, or ‘‘ANSI C’’, was completed in late 1988. Most of the features of the

standard are already supported by modern compilers.

The standard is based on the original reference manual. The language is relatively little changed; one of the goals of the

standard was to make sure that most existing programs would remain valid, or, failing that, that compilers could produce

warnings of new behavior.

For most programmers, the most important change is the new syntax for declaring and defining functions. A function

declaration can now include a description of the arguments of the function; the definition syntax changes to match. This extra

information makes it much easier for compilers to detect errors caused by mismatched arguments; in our experience, it is a

very useful addition to the language.

There are other small-scale language changes. Structure assignment and enumerations, which had been widely available, are

now officially part of the language. Floating-point computations may now be done in single precision. The properties of

arithmetic, especially for unsigned types, are clarified. The preprocessor is more elaborate. Most of these changes will have

only minor effects on most programmers.

A second significant contribution of the standard is the definition of a library to accompany C. It specifies functions for

accessing the operating system (for instance, to read and write files), formatted input and output, memory allocation, string

manipulation, and the like. A collection of standard headers provides uniform access to declarations of functions in data

types. Programs that use this library to interact with a host system are assured of compatible behavior. Most of the library is

closely modeled on the ‘‘standard I/O library’’ of the UNIX system. This library was described in the first edition, and has

been widely used on other systems as well. Again, most programmers will not see much change.

Because the data types and control structures provided by C are supported directly by most computers, the run-time library

required to implement self-contained programs is tiny. The standard library functions are only called explicitly, so they can

be avoided if they are not needed. Most can be written in C, and except for the operating system details they conceal, are

themselves portable.

Although C matches the capabilities of many computers, it is independent of any particular machine architecture. With a little

care it is easy to write portable programs, that is, programs that can be run without change on a variety of hardware. The

standard makes portability issues explicit, and prescribes a set of constants that characterize the machine on which the

program is run.

C is not a strongly-typed language, but as it has evolved, its type-checking has been strengthened. The original definition of

C frowned on, but permitted, the interchange of pointers and integers; this has long since been eliminated, and the standard

now requires the proper declarations and explicit conversions that had already been enforced by good compilers. The new

function declarations are another step in this direction. Compilers will warn of most type errors, and there is no automatic

conversion of incompatible data types. Nevertheless, C retains the basic philosophy that programmers know what they are

doing; it only requires that they state their intentions explicitly.

C, like any other language, has its blemishes. Some of the operators have the wrong precedence; some parts of the syntax

could be better. Nonetheless, C has proven to ben an extremely effective and expressive language for a wide variety of

programming applications.

The book is organized as follows. Chapter 1 is a tutorial on the central part of C. The purpose is to get the reader started as

quickly as possible, since we believe strongly that the way to learn a new language is to write programs in it. The tutorial

does assume a working knowledge of the basic elements of programming; there is no explanation of computers, of

compilation, nor of the meaning of an expression like n=n+1. Although we have tried where possible to show useful

programming techniques, the book is not intended to be a reference work on data structures and algorithms; when forced to

make a choice, we have concentrated on the language.

Chapters 2 through 6 discuss various aspects of C in more detail, and rather more formally, than does Chapter 1, although the

emphasis is still on examples of complete programs, rather than isolated fragments. Chapter 2 deals with the basic data types,

operators and expressions. Chapter 3 threats control flow: if-else, switch, while, for, etc. Chapter 4 covers functions and

program structure - external variables, scope rules, multiple source files, and so on - and also touches on the preprocessor.

Chapter 5 discusses pointers and address arithmetic. Chapter 6 covers structures and unions.

Chapter 7 describes the standard library, which provides a common interface to the operating system. This library is defined

by the ANSI standard and is meant to be supported on all machines that support C, so programs that use it for input, output,

and other operating system access can be moved from one system to another without change.

Chapter 8 describes an interface between C programs and the UNIX operating system, concentrating on input/output, the file

system, and storage allocation. Although some of this chapter is specific to UNIX systems, programmers who use other

systems should still find useful material here, including some insight into how one version of the standard library is

implemented, and suggestions on portability.

Appendix A contains a language reference manual. The official statement of the syntax and semantics of the C language is

the ANSI standard itself. That document, however, is intended foremost for compiler writers. The reference manual here

conveys the definition of the language more concisely and without the same legalistic style. Appendix B is a summary of the

standard library, again for users rather than implementers. Appendix C is a short summary of changes from the original

language. In cases of doubt, however, the standard and one’s own compiler remain the final authorities on the language.