

4-23-2004

Computer-Assisted Music Composition in the 32-bar Jazz Standard Form

Chad Hogg

Follow this and additional works at: https://digitalcommons.ursinus.edu/comp_hon

Digital Commons is part of the Composition Commons, Computer Sciences Commons, and the Music Theory Commons. [Click here to let us know how access to this document benefits you.](#)

Digital Commons Network

Logo

Computer-Assisted Music Composition in the 32-bar Jazz

Standard Form

Chad Hogg

April 23, 2004

Submitted to the faculty of Ursinus College in fulfillment of the
requirements for Honors in Computer Science and Music

Computer Science Advisor:

Nancy Hagelgans
Dr. Nancy Hagelgans

Music Advisor:

John French
Dr. John French

Committee Members:

Nancy Hagelgans
Dr. Nancy Hagelgans

John French
Dr. John French

Dr. Chris Cellucci
Dr. Christopher Cellucci

Approved:

Roger Coleman
Dr. Roger Coleman
Dept. of Math and CS Chair

Table of Contents

Abstract	4
A Brief History Of Research In Computer Music	6
Basic Music And Jazz Theory	10
Evolution Of The Project: CHChordBase.dll	17
Evolution Of The Project: CHRandomChords.exe	21
Evolution Of The Project: CHChordsSearch.exe	22
Conclusions	30
Notes For Reading And Compiling The Code	33
Bibliography	35
Table of Experimental Results	Appendix A
Lead Sheets	Appendix B
Source Code for the Project	Appendix C

Abstract

This paper is presented as an explanation of research performed by Chad Hogg during the Fall 2003 and Spring 2004 semesters at Ursinus College. The goal of this research project was to explore the potential for teaching a computer to compose jazz music. Specifically, the hope was to develop a method in which every decision could be made based on information about the entire system. If this problem were solved, the ancillary goal of allowing the user to set any arbitrary constraints that have global effects would be a trivial addition.

The first section of this paper is a brief summary of the work that has already been completed in the field of relating computation and music. The purpose of this section is twofold: to show that the relationship between computation and music is not unreasonable and to demonstrate that prior solutions to the problem have depended on localization of decisions.

Following this section is a broad overview of music theory and its specific applications to jazz. The goal here is to familiarize the reader with terminology and concepts that will be discussed later as the impetus for certain programming decisions. Additionally, this section presents an accurate depiction of how a lead sheet, which is the output of the project, is used by a group of jazz musicians during performance.

The remaining section of the paper is a thorough discussion of the methods used in programming a solution and the problems encountered. This section assumes some familiarity with programming concepts and the general workings of the modern

computer. Additionally, it will be much more useful if read in conjunction with the appendix, where the source code and inline documentation may be found.

Finally the paper concludes with a discussion of results obtained from listeners and further work to be completed. A complete listing of the source code for the solution presented and several pages of sample outputs are appended to the end.

A Brief History of Research in Computer Music

The study of Western music must begin with a thorough understanding of the theories of early Greek philosophers and mathematicians. Writings of the followers of Pythagoras record that his interest extended far beyond the geometric principles for which he is commonly remembered; Pythagoras also formed the basis for the music theory of classical society through the early Renaissance. Legend holds that Pythagoras noticed the different pitches that resulted from the clanging of hammers in a blacksmith shop, and that further research demonstrated a mathematical relationship between the size of the hammers and the tones produced. Specifically, he discovered that simple size ratios such as 2:1, 3:2, and 4:3 produced the consonant intervals known today as an octave, a perfect fifth, and a perfect fourth respectively. More complex ratios such as 1024:729, a diminished fifth, generally produced highly dissonant intervals. This mathematical approach to music provided the foundation for early monophonic and polyphonic ideas. (Wood 50-51, 182)

In modern society music is often viewed as a pure art, divorced from anything but a trivial connection to science or mathematics and requiring a spark of human creativity to be of any value. Attempts to reduce music to its mathematical foundations are often scorned by musicians as removing the human aspect that makes music something more than noise. Nevertheless, some progressive musicians and computer scientists have recently attempted to construct music based upon computational algorithms. Much work has been done in the field of classical music, but relatively little research has been performed with computer-generated jazz.

Work in the broad field of computer music is generally separated into two concentrations: composition and performance. Performance applications have ranged from the development of the commercially successful Musical Instrument Digital Interface (MIDI) and synthesizers to more esoteric uses. Such uses include manipulation of digital signals to produce timbres unlike anything ever created by acoustic instruments and programs that use artificial intelligence to simulate the expressive nature of human performance.

One of the more controversial developments in performance-oriented research has been a number of computer programs designed to interpret dancing or mouse movement as instructions that are combined with rules known to the system to produce music. Laurie Spiegel, creator of one of the pioneering efforts in the field, believes that her program Music Mouse emancipates “musicality” from “sheer physical coordination” and “the ability to deal with and manipulate symbolic notation”, which “have nothing to do with musical ability”. (Jacobs 16) Obviously, this notion is going to bother some people who have spent lives refining their coordination and knowledge of proper notation. Jacobs describes his experiences with Music Mouse in a rather different light: “When I use Music Mouse I am not making music – or at least that’s how it feels to me; rather, I’m offering the computer some extremely simple prompts which allow it to make music, and only the kind of music it knows how to make.” Furthermore, he equates the user’s efforts to create music as being as valid as a video game player’s athletic accomplishments. (Jacobs 17)

Compositional systems have existed from near the first days of the computer. The first major breakthrough was the Illiac Suite, a set of compositions generated by Lejaren Hiller, Leonard Isaccson, and the ILLIAC I computer in the late 1950s. Hiller and Isaccson's program used Markov chains to pseudorandomly produce streams of notes and then passed the results through a set of heuristics. Any note that failed to meet the requirements of the heuristics was regenerated until an acceptable solution had been found. (Arcos 44)

One of the first attempts to apply computers to the problem of jazz music was by John Rothgeb in 1969. He developed a system that would take a bass line in the style of a jazz bassist and attempt to guess the chord structure in which it fit. Rothgeb's program used a series of harmonic rules to generate its guesses. (Arcos 45)

In the early 1970s a number of methods that avoided use of probability propagated through the industry. In 1972 James Moorer was able to generate simple melodies using only a list of heuristics. Gary Rader emphasized a rules-based artificial intelligence system in 1974. The Rader system used a set of applicability rules, which determined what options are possible, and a set of weighting rules that chose the best of several possible alternatives. (Arcos 44)

Evan David Jones's more recent Counterpoint Assistant (CPA) takes a different approach. This computer program does not contain any musical information itself; it is only a system that will enforce certain constraints set by the user. The system uses these rules to produce a harmony – melody matrix for classical

counterpoint. Rather than returning a single best result, CPA returns every musical possibility that matches all the user's constraints. (Jones 33)

Each of these methods has been a step forward in the understanding of a computer's usefulness in composition, but each has had its flaws. The early probabilistic methods are fast and efficient, but they only use a small amount of information to determine each step. These methods are quite acceptable for serial composition and other techniques that do not rely on a strong sense of key relationships, but they are inappropriate for jazz composition. Every good jazz standard contains patterns, repetition, and very deliberate relationships among chords in one section and another.

The rules-based systems have precisely the opposite problem. These approaches can take the entire composition into perspective when making decisions, but doing so can require exhaustive amounts of resources. It is also difficult to select a set of rules that is restrictive enough to prevent fallacies but not so restrictive as to stifle any sense of creativity. As Arcos explains, "Purely rules-based harmonization usually fails because, in general, the rules don't make the music; it is the music that makes the rules." (Arcos 45)

Basic Music and Jazz Theory

Sound, as perceived by the human ear, exists as waves produced by vibrations of air or other substances. Such a sound may be quantified by four different metrics – volume, timbre, duration, and pitch. Volume is related to the amplitude of the resultant wave. Variations in volume are used for effect in nearly all music, but in jazz considerations of volume are generally determined by the performers, rather than the composer. Thus volume considerations are outside the scope of this project.

The timbre of a sound is determined by the shape of its wave. A difference in timbre can be experienced as the distinction between a piano and a trumpet that are otherwise producing the same note. Such a thing is quite difficult to explicate, but the property is immediately obvious to anyone who has heard the two sounds. Jazz musicians have used instruments that produce timbres anywhere from the extremely simple wave of the flute to the complex waveform of the distorted electric guitar. Thus considerations of timbre are not generally decided by a jazz composer either.

Duration is simply the amount of time during which a sound is produced. The durations of a series of sounds produces what is called rhythm in music. Although performers typically alter the rhythms of a song to express themselves more clearly or to change the style of a song, general control of rhythm is certainly important for a jazz composer.

The final attribute of sound, pitch, is much more interesting to the jazz composer. The pitch of a sound is the period of its wave. A sound with a long period has a small frequency, which is its inverse, and sounds low to the ear. A sound with a short period has a high frequency and sounds high to the ear. However, to

characterize the sound produced by a musical instrument as having a single pitch is an oversimplification. When a musician plays the note known as A (440 Hz), the sound clearly contains a wave with a period corresponding to 440 Hz. This is the most prominent component of the wave, and it is known as the fundamental. In most cases there are many smaller waves occurring at the same time as the fundamental. The combination of all these pitches is heard together as a single sound. Differences in the prominence of the many different layers of pitch in the sound produce the differences in shape of the resultant wave, and thus the aspect known as timbre.

Music may be defined as a set of sounds being produced concurrently and sequentially. The fundamental properties of music are harmony, melody, and rhythm. Each of these properties of music involves the variation in a certain property of sound across the sequence of sounds of which the music is composed.

Certain ratios of pitch frequencies are recognized as known intervals. For example, if the ratio of the frequencies of two pitches is 3:2 the pitches are in an interval of a perfect fifth. Two pitches with a ratio of frequencies of 2:1 are in the interval of a perfect octave, and these pitches are considered fundamentally similar and assigned the same letter names.

When sounds of different pitches are produced concurrently, the result is known as harmony. Notes in a particular interval that are sounded simultaneously produce a similar kind of sound, regardless of the actual pitches. When several notes in particular intervals are sounded together, they produce a chord. A multitude of such combinations exist and could be used, but chord theory generally begins with simple triads, sets of three different pitches played together. The two predominant

triads are the major and the minor. Each contains the root and a perfect fifth above that root. The major triad also contains a major third above the root, while the minor triad includes a minor third above the root. Although these triads provide a basic distinction between chord sounds, they are usually extended in jazz to include other notes. For the purposes of this program, three full chord qualities are used. The major seventh chord is a major triad that contains an additional major seventh above the root. The dominant seventh chord is a major triad that contains an additional minor seventh above the root. The minor seventh chord is a minor triad with an additional minor seventh above the root.

When sounds of different pitches are produced consecutively, the result is known as melody. Notes in a particular interval that are sounded sequentially produce a similar kind of sound, regardless of the actual pitches. The melody of a composition is what is commonly known in jazz as the head, and is what generally makes a song recognizable.

Rhythm is the organization of sounds with respect to time. This organization involves meter, melodic rhythm, and harmonic rhythm. The meter of a song is the way individual units of time are grouped together. For example, the great majority of jazz compositions are written in the meter of 4/4. This means that the beat, or fundamental unit of time, is represented as a quarter note and that these quarter notes are organized into measures of four beats. Many other meters have been used in music and even in jazz, though rarely. Melodic rhythm is the pattern of how many beats will be allocated to each note of the melody. Harmonic rhythm is the pattern of how beats are allocated to each chord.

The jazz composer's work is to create a progression of chords and a melody to played at the same time. The development of a harmonic progression is a very complex process. First, the composer must choose the structure of the composition: its length and form. A jazz composition may be any length, but 32 measures has become the standard and all compositions generated as part of this project comprise 32 measures. Form is the pattern of repetition, if any, that appears in the piece. The standard jazz form is AABA, where the first 8 measures of the song are very similar to the second 8 measures and the fourth 8 measures. The third 8 measures are different and known as the bridge. All compositions generated in this project are in the AABA form.

The composer must also consider the harmonic rhythm that he would like the piece to exhibit. Early works of jazz used varied harmonic rhythms, but modern compositions tend to change chords either at every half measure or to hold the same chord for many measures. Whatever system the composer decides to use, he usually alters the harmonic rhythm at certain places, such as the end of the composition, to create an effective cadence.

Finally, the composer must choose the actual chords that comprise the progression. There is a large number of rules that govern how chords may be arranged to produce a particular effect. In jazz especially, these rules are often broken. However, they are generally useful when forming sequences of chords. An example of such a rule is that dominant chords produce a sound that is full of tension and often described as unpleasant. However, the resolution of this tension to a chord whose root is either a perfect fifth or a minor second below the root of the dominant

chord produces a highly enjoyable sound or feeling of the release of tension. This is quite difficult to describe but simple to demonstrate. Thus the use of dominant chords must be carefully planned. This resolution would normally be timed at certain moments, such as the end of a phrase or across a barline. A dominant chord that does not resolve to an expected chord sounds inappropriate and destroys any sense of logical motion in the progression.

The process of devising a melody is equally complex. As different chords produce different musical effects, so does the use of different notes of the melody against a chord. All the notes that comprise a chord sound fine played against it, but other notes range from mildly consonant to painfully dissonant. The composer must know how to use these different sounds to induce the desired feeling. The different intervals between consecutive notes in a melody are also important. Traditional jazz compositions favor small intervals between consecutive notes to produce a smoothly flowing line. Modern bebop compositions have taken the opposite approach, using many large leaps from one note to another in a more angular, disjointed line. Melodic rhythm must be considered as well, from the overarching decision between a swing, Latin, or ballad feel to the use of specific rhythms to organize the melody into phrases.

Much confusion exists regarding the nature of a complete jazz composition and its performance. A composition consists of merely the melody written in standard musical notation with symbols above the staff that represent the chord progression. For example, a major chord with C as its root might be notated "CMa7". A piece of music notated in this style is known as a lead sheet. Often it also contains

a general indication of appropriate style and tempo, and perhaps some extra markings to denote dynamic changes or note articulations.

This lead sheet is provided to each member of a jazz combo, which generally includes a drummer, a bassist, and pianist, and one or more horn players. The band begins by playing exactly what is written on the lead sheet. This is known as the head. Then the band continues playing through the chord progressions multiple additional times while different members of the group improvise their own melodies based on the chord progression. Each repetition of the entire chord sequence is known as a chorus. Finally the band plays the head again to close the piece.

Of all the members of the band, the drummer is the least constrained by the composer. He plays some sort of rhythm that he feels is consistent with the style the composer desired, but this varies greatly from one drummer to another. During the head, the drummer may support the phrasing of the melody by filling the space between one phrase and the next. During other members' solos he generally ignores the lead sheet and instead bases his playing on what he hears the soloist doing.

The bass player uses the chord symbols to determine what he plays. The goal of the bassist is to outline the chords in the progression and to help the drummer keep steady time. He chooses notes that are either in the chord or sound good with it, and he plays one of them at a time to invoke the particular sound of that chord. The rhythm of the bass part is dependent on the style of the composition. In a swing piece the bassist uses quarter notes almost exclusively, while he uses dotted-quarter – eighth rhythms for a Latin song. If the bassist is given the opportunity to play a solo

through a chorus, he often plays the instrument in a melodic fashion and ignores these rules.

The pianist is responsible for voicing the chords so that the melody or improvisations of others is heard within that harmonic framework. The piano part is largely improvised but usually consists of fragments of the current chord played in varying rhythmic patterns. This is known as comping. When the pianist is the soloist, he often combines this comping with one hand with playing a melodic line with the other.

The remaining members of a band generally play wind instruments such as the saxophone, trumpet, or trombone. These players are responsible for the melody. During the head, they play the melody exactly as it is written on the lead sheet. When given the opportunity to play a solo, they create other melodies based on the same progression of chords. During other people's solos they may improvise short background passages to add excitement to the solo or they may simply play nothing.

The confusion arises because large jazz ensembles known as big bands operate on a fundamentally different principle. In most cases a big band plays a specific arrangement of a composition. This means that someone has taken a lead sheet and written out specific parts for each of the instruments that are used in performance. This process of arranging is necessary because there are simply too many people in a big band to perform in the improvised manner of a jazz combo. The goal of this project is not to produce arrangements, but only compositions.

Evolution Of The Project: CHChordBase.dll

The initial goal of this project was to develop an engine that could generate a complete jazz composition based on user-provided constraints. The initial step towards this goal was to create a general music library that could be used to simplify code for any type of compositional algorithm. This library was designed to incorporate all information needed for storing, retrieving, and calculating information about the components of a composition.

The first part of this library was a document that defined musical properties in a numerical notation that would be both comprehensible to the reader of the program and simple to manipulate with the constructs of the general-purpose programming language C++. The initial step was to use macros with English-like names that were mapped to specific integers. For example, each of the possible pitches in the scale was assigned an integer, 1 through 12. The pitch C was accessed by the macro `PITCH_C` and had a value of 1. The advantages of this system were readability and ease of iterating through the different pitches. A similar system was used for the various intervals and qualities of chords. At a later time these macros were converted to actual enumerated types. While this requires some additional conversion code at places, it promotes readability and safety of types.

The next addition to this library was a class `CCHChordData` that performs simple calculations based on this data. Two of its functions, `CCHChordData::GetChordName()` and `CCHChordData::GetPitchName()`, return a string containing a textual representation of the chord type or pitch, respectively. The functions `CCHChordData::GetChordFirstInterval()`,

CCHChordData::GetChordSecondInterval(), and

CCHChordData::GetChordThirdInterval() take a quality of chord and return their respective interval within that chord. All of these functions look up information in a hash table that is initialized with correct values in the constructor. Two additional functions exist to perform calculations. The first,

CCHChordData::AddIntervalToPitch(), takes an interval and a pitch as parameters and returns the pitch that is above the specified pitch by the specified interval. The other, CCHChordData::GetInterval(), takes two pitches and returns the interval between them. Together these functions can be combined to retrieve any necessary information about a chord. For example, to print out the second note of a C major chord, the appropriate interval must first be determined by the function

CCHChordData::GetChordFirstInterval(). Then that interval is added to the root, which is C, by CCHChordData::AddIntervalToPitch(). Finally, the resulting note is passed to CCHChordData::GetPitchName() and its result is printed to the screen.

The rest of the classes in this library were added much later, when it became clear that duplicate instances of similar code in other places could be effectively combined. The CCHChord class was developed to contain all the information and actions regarding a single chord. The two attributes of a chord are its root and its quality or type. The CCHChord class contains accessor functions to set and retrieve each of these properties. Additionally, there are functions for retrieving each of the notes that comprise the chord and for printing a complete chord symbol.

The internal representation of information in the CCHChord class was quite simple initially: there was a member variable of the PITCHES type to hold the root

and a member variable of the `CHORD_TYPES` type to hold the quality. This system provides maximal readability and simplicity, but is quite an inefficient use of memory. Continued work on the project brought situations where 36^4 arrays of 4 `CCHChord` objects needed to be resident in memory for extended periods of time. An enumerated type is stored in 4 bytes of memory, so the structure size of a `CCHChord` object is 8 bytes. The resultant memory usage required constant shifting of information in and out of virtual memory and slowed program execution unacceptably. A partial solution to this problem is to store both properties of the chord in a single piece of data: a 1-byte unsigned character. The type of the chord is stored in the 4 high bits, while the root of the chord is stored in the 4 low bits. For this reason, the enumerators of `CHORD_TYPES` are given values such as 16 and 32 rather than 1 and 2. Extracting and setting the values of the two properties uses low-level instructions that are highly optimized, but obtuse to the reader. Fortunately, this code is entirely internal to the `CCHChord` class.

The next important class was `CCHProgression`, which represents a progression of 64 chords. The internal data is simply an array of 64 `CCHChord` objects. The class consists mainly of accessor functions for the properties of individual chords in the progression and functions to print the entire progression. Additionally, there is a function `CCHProgression::AppendShort()`, to copy information from a `CCHShortProgression`, which will be discussed shortly, into a particular position in the array.

The `CCHShortProgression` class is the second half of a solution to the memory use problem. Although only progressions of 4 chords needed to be stored for a large

part of the algorithm, the existing data structure, which was 16 times larger than necessary, was naively used. The reasons for this misuse of memory will be explained later. The CCHShortProgression class is therefore nearly identical to the CCHProgression class, except that its array is only of length 4.

The final portion of this library was supposed to be the routines that write the internal data into the industry standard for storing musical information: a standard MIDI (Musical Instrument Digital Interface) file. Unfortunately, time constraints have prevented the completion of this section of the project. This aesthetic enhancement is left to future work.

Evolution Of The Project: CHRandomChords.exe

The next step in this project was to test the library and to generate some examples of what an uninformed algorithm would produce for later comparison. The CHRandomChords program was created to fill these needs. This program simply chooses each of the chords in a progression of 64 chords randomly. The entire progression is then printed to the console output. Below this is a list of the individual notes that comprise the first 8 chords, to demonstrate the library's knowledge of this information. Much of the code that was originally written in CHRandomChords was later moved to the CCHChord and CCHProgression classes to prevent code duplication. What is left is a very simple usage of the library functions.

Evolution Of The Project: CHChordsSearch.exe

The core of the project is to design and implement the algorithms that will be used to generate compositions. Most previous approaches that have been used for computer music have made decisions only on a local basis. If a system could be devised such that the choice of each chord and note in the composition depended on every other piece, the result would be much more cohesive. The initial step was a system that generates the chord progression of a composition, so that a melody could later be added to it.

The first attempt at an algorithm was to recast the generation problem into a searching problem. The algorithm is simple: start from the root of a tree and add a child node for each possibility. Continue until all possibilities have been generated. Use some metric to evaluate all the leaf nodes of the tree, and choose the optimal leaf. The field of artificial intelligence provides numerous examples of this approach to problem-solving as well as methods for pruning the tree to reduce the number of necessary calculations. A complete solution using this method would fit the initial goal, because no real decision would be made until all possible data had been considered. Additionally, the user could set constraints at any point that would merely be added to the evaluation function that is run at the end. These constraints would be truly shaping the composition, because they would have more than a local effect.

The problem with this algorithm did not become apparent until the testing phase. This seems like a feasible solution until available computational resources are considered. However, the initial data structure for a chord used 8 bytes of memory.

A full progression of 64 chords was thus 512 bytes in length. At each position in the progression there were 12 different root pitches and 3 different chord qualities to choose from, which could be combined in all of the 36 possible ways. Thus a tree that searched only to a length of 4 chords would require $(512 \text{ bytes / node}) * (36 \text{ children / node}) * (36 \text{ children / node}) * (36 \text{ children / node}) * (36 \text{ children / node}) =$ approximately 820 megabytes of storage. For a full tree to 64 chords, the number of nodes would be 36^{64} , or approximately $2 * 10^{96}$ megabytes of storage. Even if the problem of storage is ignored, the creation of this many nodes would be far too computationally expensive for the resources available.

Various schemes for pruning were considered, but none were satisfactory. To be effective, the storage needs would have to decrease from 10^{96} megabytes to no more than 10^3 megabytes. Such severe pruning would destroy the concept of globalization in decision-making. Additionally, the overhead of evaluating each node merely slowed the process even further. This concept has unfortunately been abandoned, but it may be reopened in future research. Multi-threading and storing temporary results to disk for later use may yet make an approximation of this method feasible.

A seemingly workable compromise was to use the existing method to generate every possible progression of 4 chords, and then choose the optimal progression. While this algorithm was not scalable, it did provide a start. In the altered method, each of the possible 4-chord progressions was stored. Then an optimal progression was chosen for the first 4 chords of the result. Then each 4-chord progression was appended to the result and evaluated. The best of these then became the result. Thus,

an 8-chord progression could be generated that would be in some ways optimal. It was easy to extend this idea to continue appending 4-bar progressions to what had been chosen so far until the entire 64-chord progression had been chosen.

This method did not completely meet the original goal, because choices were still based on highly localized data. The choice of the first and second progressions will ultimately affect the others, yet they are chosen when very little information is available. However, the choice of blocks of four chords is somewhat less localized than other methods, which choose a single chord at a time based on only the previous chord.

Although this continues to be the method used for generating chord progressions, it has evolved through a lengthy cycle of optimizations and other changes. It was during this time that the CCHChord and CCHProgression classes were created in the music library, and the memory usage optimizations made. Additionally, the evaluation function that is used to choose among the many possible progressions has been in a continual process of testing, fine-tuning, and retesting to ensure that the various rules in it are balanced. As this function grew in size, it became necessary to reduce the portion of the progression that is evaluated at a given time. Another optimization converted several evaluation rules that encouraged an AABA form into code that simply copied the A section to different parts of the progression, which greatly reduced running time.

The evaluation function for chord progressions has included many metrics through its lifetime. The first deals with dominant chords. The sound of a dominant chord is volatile, and the listener will strongly feel a need to resolve its tension. This

resolution occurs properly if the root of the following chord is either a perfect fifth or a minor second below the root of the dominant chord. Because the repetitive creation and release of tension provides some of the forward momentum of jazz tunes, this use of dominant chords is strongly favored by the algorithm. Dominant chords that do not resolve properly invoke a tension that is never resolved when another unrelated chord begins. This type of sequence is usually avoided by the program because it is rarely used by composers.

The feeling of resolution from a properly used dominant chord is known as a cadence. A stronger and more complete cadence involves placing a minor chord before the dominant that is a major second above the target chord. This cadence is fundamental to jazz and is often used to mark the ends of sections. Additionally, both the A section and the B section are centered around a specific key: a grouping of related chords and scales. A minor chord that is both a part of the key and used in a cadence seems appropriate, while a minor chord that has neither of these properties will seem out of place. Similarly, certain major chords belong to a key, others are related to the key, and others are unrelated.

For the purposes of storage, every chord gets exactly two beats of a measure. However, it is possible for the same chord to be repeated, in which case it would be considered to just be continuing. This is natural within a measure but awkward across measures. The ends of 8-measure phrases are encouraged by the algorithm to end with a cadence to the next phrase, and the beginnings of each section are encouraged to have a clear indication of the key. Other metrics that have been

removed include discouraging repetition of the same chord often and encouraging the AABA structure.

When the generated chord progressions reached a satisfactory quality work could be done to fit melodies over them. The original intention had been to use a similar process to what I hoped to do with chord progressions. Unfortunately, the searching procedure was even more unsuitable for melodies; it would require a search through 12^{256} different nodes!

After attempts at several unsatisfactory methods, it became obviously necessary to consider the more localized approach of many older projects. The model that was finally chosen is of a Markov Chain from probability theory. A Markov Chain is a stochastic process in which each possible state of the system has a specific probability of being chosen as the next state, where these probabilities are based on the current state of the system. Another way to think of such a system would be as a weighted, directed graph where the sum of the weights of all edges leaving a node is 1, or 100 percent.

There are several things to consider when trying to compose a melody: the position of the note within the current chord, the relationship between each note and the notes surrounding it, and phrasing. To handle as many of these considerations as possible, a state of the system or node of the graph was defined as being a position (upbeat or downbeat) and the interval between the note and the chord over which it is played. A bit of a trick is used storing this internally: the weights are determined by looking at the previous note in context of the chord that will be used for the next note.

For example, consider the progression D-7 G7. The last note on the D-7 chord is a C. Within the chord D-7, it is the interval of a minor seventh above the root. But when generating the first note for the G7 chord, the note is treated as belonging to that chord. Thus the note C is a perfect fourth above the root of a G7 chord. The most likely next notes are B (the major third of a G7 chord) or D (the perfect fifth of a G7 chord), because the note should fit the chord and usually be close to the previous note in the scale. Additionally, it is known that position must alternate. Thus the probability of any node that is an upbeat is 0 after a node that is an upbeat. Internally, values are stored not as intervals but as pitches that are the interval above a C. This requires a small amount of extra computation to convert values when looking up probabilities, but it is much more convenient for the person determining the probabilities.

This system is represented by a 4-dimensional matrix, with dimensions chord type (3) x position (2) x previous note relative to a C (12) x note that is being chosen relative to a C (12). The contents of the matrix were filled simply by the programmer's knowledge of melodic rules. Given more time, it would be quite appropriate to instead base this entire matrix on a quantitative analysis of existing compositions. Unfortunately, a complete sample space would probably require around 100 compositions, or 25,600 notes to analyze.

The probabilities in this matrix are based on several criteria. First, the note to be chosen must fit the chord. All the pitches that are actually used in a chord sound fine, but other pitches may also be effective. For example, the note that is a perfect fourth above the root of the chord sounds terrible on a major or dominant chord,

because it clashes with the major third. On a minor chord, however, the perfect fourth yields a pleasant sound. These rules must be stringently enforced when notes fall on a downbeat. On an upbeat, the note is played only for a brief time, and may be a note that sounds dissonant played against a chord, but works well for making a transition between strong downbeats.

The chosen note must also relate to the note before it. A melody that is constantly jumping from one note to another far removed from it and back will be difficult to play and sound disjointed to the ear. While this effect is certainly used, especially in bebop, a generic approach to jazz composition must focus on movement by small intervals that produces a more flowing line. Finally, the willingness to step outside these general guidelines is necessary to produce genuinely creative work. Thus the probability of any legal notes being chosen is at no time less than $1/100$.

There are two significant drawbacks of this system. First, it precludes any type of a constraint system. Because all the notes that fall before a constrained one are generated based on only previous notes, they may have no relationship with the constrained note. The larger flaw is that there is very little control over phrasing. A good melody will rise and fall and provide rests for the performers between phrases. When each note only knows about the previous one, there can be no concept of long phrases. The only control the program has over this is in the distinction between downbeats and upbeats. However, this solution is computationally easy and produces passable results.

This section of the paper contains only a global explanation of purpose and methodology because a function-level description would be unnecessarily verbose.

The intricate details of implementation are heavily documented in the appendix, which is a complete listing of the source code.

Conclusions

This project has demonstrated the difficulties inherent in the initial objectives. Making decisions based on a global knowledge of the system is far too computationally expensive to be feasible with current algorithms and available hardware. The project has resulted in a program that does generate complete jazz compositions as output, but is the program actually performing creative work? The level of artificial intelligence extends only to allowing the computer to make choices based on the same criteria as its programmer would manually. The program may not generate its own rules, nor does it have any way of knowing what sounds "good", because "sounding good" cannot be quantitatively described.

The general method for testing the success of an artificial intelligence application is the Turing Test. Historically this test has been used specifically with applications that are designed to simulate conversation. An impartial user converses with the application and a human through the same interface. If he is able to determine which of the conversants is a program the application fails the test. Otherwise, it passes.

Simulation of this type of test required a set of sample compositions – some generated by the program and others taken from the jazz canon. Additionally, these compositions needed to be presented in a way that the common man could understand them. Six compositions were chosen for the test: three that had been generated by the program, "Shaw 'Nuff" by Charlie Parker and Dizzy Gillespie, "Early Autumn" by Woody Herman, and "Line For Lyons" by Gerry Mulligan. Two of these tunes are written in the style of a ballad, two are swing charts, and two are examples of bebop.

A small jazz combo consisting of Ursinus students and faculty members recorded very simple arrangements of the compositions so that the tunes themselves could be compared.

The test was ran in two different settings. The first was a group of 19 students who were near completion of a course in either Harmony (MUS222) or Jazz History (MUS207). Of these students, one correctly identified all of the generated compositions, 16 identified two of the generated compositions, one identified one of the generated compositions, and one student was entirely incorrect. The second setting was a group of 11 students in the Operating Systems (CS474) course. Of these students, three correctly identified all of the generated compositions, four identified two of them, and four identified one of them.

These results are quite interesting. A random selection of three compositions from the six would result in an average of 1.5 correct choices, while the experimental average was 1.9 correct choices. This difference is significant, but small enough to indicate that the similarities between the generated and written compositions are sufficient to confuse even a careful listener. Further interesting results are that the incorrect guesses were fairly well distributed among the compositions and that non-music students correctly identified an average of 1.909 compositions while music students only identified 1.895 correctly.

The project has proved more difficult and time-intensive than anticipated, so several parts remain to be done as future work. First among these is implementing an engine to convert the output to a MIDI file format. Currently the output is a textual representation of notes and chords, which must be tediously entered into a musical

notation program by the user. The probability matrix for generating melodies is the next priority. This should be based on actual data collected from a representative sample of existing jazz compositions. The remaining goal is to continue experimenting with optimization and new algorithms that would allow a closer approximation of the original objectives, which include a constraint-based choice.

Notes For Reading And Compiling The Code

All the source code used in this project has been written in the C++ language, using the libraries of the Microsoft Foundation Classes (MFC). Because this system is designed strictly for Windows users, the program is only compilable under a Windows system running Microsoft Visual Studio 6.0. However, the bulk of the code is not concerned with a user-interface and is not heavily tied to its libraries. A competent programmer could rewrite the code to run on a UNIX or Macintosh system by writing his own string, list, and hash table classes and altering the sections that set up the system for console input and output.

Building programs that will link to MFC, and especially to other Dynamic Link Libraries (DLLs) requires work that is unrelated to the project itself. Most of this information is stored in the .dsp files and several source files. The contents of these files are irrelevant and have thus not been appended to this paper. However, they are available in digital form so the reader may compile his own version.

There are three types of documentation that are used in the following code. The first type uses the C++ style `/* ... */` markers to denote the beginning and end of the comments. All documentation that is found in this format is used by Doxygen, an external documentation engine. The Doxygen documentation is primarily used to determine the purpose of a class or function. The strength of this documentation system is its searchability and ease of linking related documentation together. These advantages do not translate well to paper, so the actual Doxygen output is only available in digital form.

There are also documentation sections surrounded by double / symbols.

These sections provide a general overview of an algorithm or purpose of a block of code. Most of the code in the project is written in a two-column format. The first column contains executable code while the second column contains documentation that describes exactly what that line of code does and why. This type of documentation is useful for determining how a particular function works. In places where documentation is unnecessary and lines of source code are long, the code has been allowed to fill both columns.

Bibliography

- Arcos, Josep Lluís and deMantaras, Ramon Lopez. "AI and Music From Composition to Expressive Performance." AI Magazine. 23:3. Fall 2002: 43.
- Biletskyy, Andriy. "Doctor Webern: A Visual Environment for Computer-Assisted Composition Based on Linear Thematism." Computer Music Journal. 24:3. Fall 2000: 34.
- Blaszczak, Mike. Professional MFC with Visual C++ 6. Canada: Wrox, 1999.
- Budon, Osvaldo. "Composing with Objects, Networks, and Time Scales: An Interview with Horatio Vaggione." Computer Music Journal. 24:3. Fall 2000: 9.
- Chadabe, Joel. "Remarks On Computer Music Culture." Computer Music Journal. 24:4. Winter 2000: 9.
- Heil, Martha. "Millennium Watch." Discover. 21:3. March 2000: 24.
- Jacobs, Alan. "Computer Control: Who's In Charge?" Books & Culture. 8:3. May 2002: 16.
- Jones, David Evan. "A Computational Composer's Assistant for Atonal Counterpoint." Computer Music Journal. 24:4. Winter 2000: 33.
- Pozzati, Guillermo. "Gen: A Lisp Music Environment." Computer Music Journal. 24:3. Fall 2000: 42.
- Sher, Chuck, ed. The New Real Book. Petaluma: Sher Music Co., 1988.
- Supper, Martin. "A Few Remarks On Algorithmic Composition." Computer Music Journal. 25:1. Spring 2001: 47.

Vaggione, Horacio. "Some Ontological Remarks about Music Composition Processes." Computer Music Journal. 25:1. Spring 2001: 54.

Additionally, this project is indebted to the following people.

Professors who have provided musical or programming guidance:

Dr. Norman David

Dr. John French

Dr. Holly Gaines

Dr. Nancy Hagelgans

Dr. April Kontostathis

Members of the jazz combo that recorded some compositions:

Josh Kistner, Drums

Dr. Holly Gaines, Piano

Chad Hogg, Bass

Ryan Michaluk, Alto Saxophone

Adam Gaines, Trumpet

Peter Bregman, Recording Engineer

Appendix A

The following page is a table of results from the experiment of asking students to listen to six recorded jazz compositions, three of which were generated by a computer program as part of this project, and to attempt to choose which of the tracks were computer generated. In the experiment, students were made aware that exactly three of the songs were computer generated and that exactly three were composed by jazz masters. Thus a random selection of three recordings would have 1.5 correct guesses on average.

In the table, the first column records which test subject the row of information came from. Columns 2-7 represent respectively the three compositions generated by the program and the three human compositions, Early Autumn, Line For Lyons, and Shaw 'Nuff. A '1' in this column means that the listener believed it was one of the human compositions.

Listener	Gen1	Gen2	Gen3	Ballad	Swing	Bebop	Correct
MUS student 1	0	0	1	1	1	0	2
MUS student 2	1	0	0	0	1	1	2
MUS student 3	0	1	0	1	0	1	2
MUS student 4	0	0	1	1	1	0	2
MUS student 5	0	1	0	0	1	1	2
MUS student 6	0	1	0	1	1	0	2
MUS student 7	0	1	1	0	0	1	1
MUS student 8	1	0	0	1	1	0	2
MUS student 9	0	1	0	1	1	0	2
MUS student 10	1	1	1	0	0	0	0
MUS student 11	0	1	0	1	1	0	2
MUS student 12	0	0	1	0	1	1	2
MUS student 13	0	0	1	0	1	1	2
MUS student 14	0	0	1	0	1	1	2
MUS student 15	0	1	0	1	0	1	2
MUS student 16	0	0	0	1	1	1	3
MUS student 17	0	0	1	1	1	0	2
MUS student 18	1	0	0	0	1	1	2
MUS student 19	0	1	0	1	0	1	2
MUS student average	0.211	0.474	0.421	0.579	0.737	0.579	1.895
CS student 1	0	1	0	1	0	1	2
CS student 2	0	0	1	1	0	1	2
CS student 3	0	1	0	0	1	1	2
CS student 4	1	0	1	0	1	0	1
CS student 5	0	0	0	1	1	1	3
CS student 6	0	1	1	0	1	0	1
CS student 7	0	0	0	1	1	1	3
CS student 8	1	1	0	0	0	1	1
CS student 9	1	0	0	0	1	1	2
CS student 10	0	0	0	1	1	1	3
CS student 11	0	1	1	0	1	0	1
CS student average	0.273	0.455	0.364	0.455	0.727	0.727	1.909
Total average	0.233	0.467	0.400	0.533	0.733	0.633	1.900

Table of Trial Results

Appendix B

The following six pages contain the lead sheets used for recording the tracks used for the previously described experiment. They are presented in the same order as the columns in the table on the previous page. Each lead sheet was created using Finale 2003. The lead sheets for the three human compositions are based on transcriptions in The New Real Book by the Sher Music Company.

GENERATED COMPOSITION 1

MEDIUM SWING

Chord progression for the first system: C^M7 F7 B^bM^A7 D^bM^A7 G^M7 C7 F7 B^bM^A7

Chord progression for the second system: C^M7 B7 B^bM^A7 D^M7 A7 D^M7 G7

Chord progression for the third system: C^M7 F7 B^bM^A7 D^bM^A7 G^M7 C7 F7 B^bM^A7

Chord progression for the fourth system: C^M7 B7 B^bM^A7 C^M7 F7 B^bM^A7 A7

Chord progression for the fifth system: D^M7 G7 C7 F^M7 E^b7 D^M7 G7 C^M7

Chord progression for the sixth system: D^M7 G7 C^M7 F^M7 E^M7 A^M7 D7 G7

Chord progression for the seventh system: C^M7 F7 B^bM^A7 D^bM^A7 G^M7 C7 F7 B^bM^A7

Chord progression for the eighth system: C^M7 B7 B^bM^A7 C^M7 F7 B^bM^A7

GENERATED COMPOSITION 2

BEBOB

Handwritten musical score for "BEBOB". The score consists of eight staves of music, each with a key signature of one flat (Bb) and a common time signature (C). The music is written in a single melodic line on a treble clef. Above each staff, there are handwritten chord symbols indicating the harmonic structure. The staves are numbered 1, 5, 9, 13, 17, 21, 25, and 29.

Chord symbols above the staves:

- Staff 1: EbMA7 E7 EbMA7 Gm7 Fm7 Bb7 EbMA7
- Staff 5: AbMA7 Fm7 Bb7 EbMA7 Gm7 C7 Fm7 Bb7
- Staff 9: EbMA7 E7 EbMA7 Gm7 Fm7 Bb7 EbMA7
- Staff 13: AbMA7 Fm7 Bb7 EbMA7 Gm7 EbMA7 Cm7 B7
- Staff 17: BbMA7 Bb7 EbMA7 BbMA7 Cm7 F7 BbMA7 AbMA7
- Staff 21: BbMA7 F7 BbMA7 EbMA7 BbMA7 B7 BbMA7 Bb7
- Staff 25: EbMA7 E7 EbMA7 Gm7 Fm7 Bb7 EbMA7
- Staff 29: AbMA7 Fm7 Bb7 EbMA7 F7 Bb7 EbMA7

GENERATED COMPOSITION 3

BALLAD

Handwritten musical notation for a ballad, consisting of eight staves of music. The notation includes treble clefs, a common time signature (C), and various chords and notes. The chords are written above the staves, and the notes are written on the staves.

Chords and notes are as follows:

- Staff 1: $BbMA7$ $Bb7$ $EbMA7$ $CMA7$ $B7$ $BbMA7$ $EbMA7$
- Staff 2: $BbMA7$ $F7$ $BbMA7$ $DM7$ $Db7$ $C7$ $F7$
- Staff 3: $BbMA7$ $Bb7$ $EbMA7$ $CMA7$ $B7$ $BbMA7$ $EbMA7$
- Staff 4: $BbMA7$ $F7$ $BbMA7$ $GM7$ $B7$ $BbMA7$ $Eb7$
- Staff 5: $AbMA7$ $BMA7$ $AbMA7$ $BbM7$ $Eb7$ $AbMA7$
- Staff 6: $FMA7$ $Bb7$ $Eb7$ $AbMA7$ $DbMA7$ $CMA7$ $F7$
- Staff 7: $BbMA7$ $Bb7$ $EbMA7$ $CMA7$ $B7$ $BbMA7$ $EbMA7$
- Staff 8: $BbMA7$ $F7$ $BbMA7$ $EbMA7$ $F7$ $BbMA7$

EARLY AUTUMN

BALLAD

BURNS & HERMAN

Handwritten musical score for the ballad "Early Autumn" by Burns & Herman. The score is written on a single staff in treble clef, 4/4 time, with a key signature of one flat (B-flat). The melody is accompanied by a series of chords written above the staff. The score is divided into measures by bar lines, with measure numbers 5, 9, 13, 17, 21, 25, and 29 indicated at the start of their respective lines.

Chords and Measure Numbers:

- Measures 1-4: C^{MA7}, B⁷, B^bMA⁷, A⁷
- Measures 5-8: A^bMA⁷, G⁷, C^{MA7}, D^{M7}, G⁷
- Measures 9-12: C^{MA7}, B⁷, B^bMA⁷, A⁷
- Measures 13-16: A^bMA⁷, G⁷, C⁶
- Measures 17-20: D^{M7}, G¹³, C^{MA7}, D¹³, D^{M7}, G¹³, C^{MA7}
- Measures 21-24: C^{M7}, F¹³, B^bMA⁷, E^b13, E⁷, E^b7, A^b7, G⁹
- Measures 25-28: C^{MA7}, B⁷, B^bMA⁷, A⁷
- Measures 29-32: A^bMA⁷, G⁷, C⁶, D^{M7}, G⁷

LINE FOR LYONS

MEDIUM SWING

GERRY MULLIGAN

Handwritten musical score for "Line for Lyons" by Gerry Mulligan. The score is written in treble clef with a key signature of one sharp (F#) and a 4/4 time signature. The tempo/style is "MEDIUM SWING". The score consists of eight staves of music, each with a measure number (1, 5, 9, 13, 17, 21, 25, 29) and a series of chords written above the staff.

Staff 1 (Measures 1-4): Chords: G6, G7(b9), CM7, F7, BM7, E7, AM7, D7.

Staff 2 (Measures 5-8): Chords: GMA7, E7, AM7, D7, G6, E7, AM7, D7.

Staff 3 (Measures 9-12): Chords: G6, G7(b9), CM7, F7, BM7, E7, AM7, D7.

Staff 4 (Measures 13-16): Chords: GMA7, E7, AM7, D7, G6, G7.

Staff 5 (Measures 17-20): Chords: CMA7, C#M7(b5), F#7(b9), BM7, E7(b9).

Staff 6 (Measures 21-24): Chords: AM7, D7(b9), BM7, E7, AM7, D7.

Staff 7 (Measures 25-28): Chords: G6, G7(b9), CM7, F7, BM7, E7, AM7, D7.

Staff 8 (Measures 29-32): Chords: GMA7, E7, AM7, D7, G6.

SHAW 'NUFF

BEBOP

PARKER & GILLESPIE

$Bb6$ $G7$ $C7$ $F7$ $Bb6$ $G7(\#5)$ $C7$ $F7(\#5)$

$Bb7$ $Eb6$ $E^{\circ}7$ $Bb6$ $G7$ $C7$ $F7$

$Bb6$ $G7$ $C7$ $F7$ $Bb6$ $G7(\#5)$ $C7$ $F7(\#5)$

$Bb7$ $Eb6$ $E^{\circ}7$ $Bb6$ $F7$ $Bb6$

$D7(b9)$ $G7(b9)$

$C13$ $F7$

$Bb6$ $G7$ $C7$ $F7$ $Bb6$ $G7(\#5)$ $C7$ $F7(\#5)$

$Bb7$ $Eb6$ $E^{\circ}7$ $Bb6$ $F7$ $Bb6$

Appendix C

The rest of this document consists of all relevant portions of the source code for programs written as part of this project. The accompanying electronic media contains a complete copy of the source code, which may be browsed, compiled, run, and modified at the reader's discretion.

```

/*! \file CHCS491\CHChordBase\ChordDefs.h
 * \brief Contains representations of various musical concepts
 * and relationships as numeric macros.
 *
 * Documented in the format of Doxygen.
 */

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// This is a header file that will be #included by other header //
// files. To prevent multiple declarations of the same things, //
// we must make sure the actual code is only ever added once //
// for each compilation. This check is performed by the use of //
// this macro, which will be defined when the contents of this //
// file are added during a compilation. When the macro is //
// defined, the contents of this file do not include any code. //
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

#ifndef _CHCS491_CHCHORDBASE_CHORDDEFS_H
#define _CHCS491_CHCHORDBASE_CHORDDEFS_H

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// The functions, classes, etc that are implented in files in //
// this project are designed to be used elsewhere. Thus, they //
// must be exported so that other projects may instantiate them //
// by linking to this object file. Before each function, class //
// or or global variable that needs to be accessible from //
// outside the .DLL file we insert a storage-class //
// specification that makes it part of the .DLL interface. //
// The macro CHCHORDBASE is defined by the project file whenever //
// you are building the CHCS491\CHChordBase\CHChordBase.dsp //
// file. It will not be defined when you are building in any //
// other project. //
// When this project is built, it needs to know to export the //
// items so they will be available for other projects. //
// When a different project is built, it will need to know that //
// can import the items from this project. //
// Thus we define macros that will have a different definition //
// depending on the project they are being accessed from. The //
// three macros CHCHORDBASECLASS, CHCHORDBASEFUNC, and //
// CHCHORDBASEDATA are used to put classes, functions, and //
// global variables into the .DLL interface respectively. //
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

#ifdef CHCHORDBASE
#define CHCHORDBASECLASS AFX_CLASS_EXPORT
#define CHCHORDBASEFUNC AFX_API_EXPORT
#define CHCHORDBASEDATA AFX_DATA_EXPORT
#else
#define CHCHORDBASECLASS AFX_CLASS_IMPORT
#define CHCHORDBASEFUNC AFX_API_IMPORT
#define CHCHORDBASEDATA AFX_DATA_IMPORT
#endif

/*! \brief Types of chords.
 *
 * These have rather large values so that they can occupy the high-order
 * bits in a byte. A pitch would be stored in the low-order bits.
 */
enum CHORD_TYPES
{
    /*! \brief An uninitialized CHORD_TYPES value. */
    CHORD_TYPE_NONE = 0,

    /*! \brief An initial value for iterating through CHORD_TYPES. */
    CHORD_TYPE_FIRST = 32,

    /*! \brief A Major7 chord.
     *

```

```

* INTERVALS::INTERVAL MAJOR3 +
* INTERVALS::INTERVAL PERFECT5 +
* INTERVALS::INTERVAL_MAJOR7
*/
CHORD_TYPE_MAJOR7 = 32,

/*! \brief A Minor7 chord.
*
* INTERVALS::INTERVAL MINOR3 +
* INTERVALS::INTERVAL PERFECT5 +
* INTERVALS::INTERVAL_MINOR7
*/
CHORD_TYPE_MINOR7 = 64,

/*! \brief A Dominant7 chord.
*
* INTERVALS::INTERVAL MAJOR3 +
* INTERVALS::INTERVAL PERFECT5 +
* INTERVALS::INTERVAL_MINOR7
*/
CHORD_TYPE_DOMINANT7 = 96,

/*! \brief A final value for iterating through CHORD_TYPES. */
CHORD_TYPE_LAST = 96
};

/*! \brief Types of closed intervals.
*
* A closed interval is one that does not span more than one octave.
* Any open interval is merely a combination of closed intervals.
* <br>The actual value of the interval is the number of half-steps
* contained in it.
*/
enum INTERVALS
{
    /*! \brief An initial value for iterating through INTERVALS. */
    INTERVAL_FIRST = 0,

    /*! \brief A unison: two notes of the same pitch. */
    INTERVAL_UNISON = 0,

    /*! \brief A minor second, two notes half a step apart. */
    INTERVAL_MINOR2 = 1,

    /*! \brief A major second, two notes a whole step apart. */
    INTERVAL_MAJOR2 = 2,

    /*! \brief A minor third, two notes a step and a half apart. */
    INTERVAL_MINOR3 = 3,

    /*! \brief A major third, two notes two steps apart. */
    INTERVAL_MAJOR3 = 4,

    /*! \brief A perfect fourth, two notes two and a half steps apart. */
    INTERVAL_PERFECT4 = 5,

    /*! \brief A tritone, two notes three steps apart.
    *
    * This is also known as an augmented fourth or a diminished fifth.
    */
    INTERVAL_TRITONE = 6,

    /*! \brief A perfect fifth, two notes three and a half steps apart. */
    INTERVAL_PERFECT5 = 7,

    /*! \brief A minor sixth, two notes four steps apart. */
    INTERVAL_MINOR6 = 8,

    /*! \brief A major sixth, two notes four and a half steps apart. */
    INTERVAL_MAJOR6 = 9,

    /*! \brief A minor seventh, two notes five steps apart. */
    INTERVAL_MINOR7 = 10,

```

```

/*! \brief A major seventh, two notes five and a half steps apart. */
INTERVAL_MAJOR7 = 11,

/*! \brief A perfect octave, two notes six steps apart.
 *
 * Note that this is essentially the same as
 * INTERVALS::INTERVAL_UNISON.
 */
INTERVAL_OCTAVE = 12,

/*! \brief A final value for iterating through INTERVALS. */
INTERVAL_LAST = 12
};

/*! \brief The different pitches in Western music.
 */
enum PITCHES
{
    /*! \brief An uninitialized PITCHES value. */
    PITCH_NONE = 0,

    /*! \brief An initial value for iterating through PITCHES. */
    PITCH_FIRST = 1,

    /*! \brief The pitch C or B#. */
    PITCH_C = 1,

    /*! \brief The pitch Db or C#. */
    PITCH_Db = 2,

    /*! \brief The pitch D. */
    PITCH_D = 3,

    /*! \brief The pitch Eb or D#. */
    PITCH_Eb = 4,

    /*! \brief The pitch E or Fb. */
    PITCH_E = 5,

    /*! \brief The pitch F or E#. */
    PITCH_F = 6,

    /*! \brief The pitch Gb or F#. */
    PITCH_Gb = 7,

    /*! \brief The pitch G. */
    PITCH_G = 8,

    /*! \brief The pitch Ab or G#. */
    PITCH_Ab = 9,

    /*! \brief The pitch A. */
    PITCH_A = 10,

    /*! \brief The pitch Bb or A#. */
    PITCH_Bb = 11,

    /*! \brief The pitch B or Cb. */
    PITCH_B = 12,

    /*! \brief A final valule for iterating through PITCHES. */
    PITCH_LAST = 12
};

```

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Everything between this and the corresponding #ifndef
// CHCS491_CHOCHORDBASE_CHORDDEFS_H will only be included once. //

```


////////////////////////////////////

#endif

```

/*! \file CHCS491\CHChordBase\ChordData.h
 * \brief Defines a class for looking up musical information.
 *
 * Documented in the format of Doxygen.
 */

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// This is a header file that will be #included by other header //
// files. To prevent multiple declarations of the same things, //
// we must make sure the actual code is only ever added once //
// for each compilation. This check is performed by the use of //
// this macro, which will be defined when the contents of this //
// file are added during a compilation. When the macro is //
// defined, the contents of this file do not include any code. //
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

#ifndef CHCS491_CHCHORDBASE_CHORDDATA_H
#define CHCS491_CHCHORDBASE_CHORDDATA_H

#include "ChordDefs.h" // Include the definitions
                        // of musical values that
                        // this class will work
                        // with.
#include "afxtempl.h" // Include the MFC
                     // collection classes.

/*! \brief A class to perform calculations on musical values.
 *
 * \nosubgrouping
 */
class CHCHORDBASECLASS CCHChordData
{
public:
    /*! \brief Initialize the maps.
     */
    CCHChordData();

    /*! \brief Return a string containing the name of the chord.
     *
     * \param p_eChord The type of chord you would like.
     *
     * Looks up its answer in CCHChordData::m_mChordNames.
     */
    CString GetChordName( CHORD_TYPES p_eChord ) const;

    /*! \brief Return the interval of the first chord tone from the root.
     *
     * \param p_eChord The type of chord you are looking up.
     *
     * Looks up its answer in CCHChordData::m_mChordFirstInterval.
     */
    INTERVALS GetChordFirstInterval( CHORD_TYPES p_eChord ) const;

    /*! \brief Return the interval of the second chord tone from the root.
     *
     * \param p_eChord The type of chord you are looking up.
     *
     * Looks up its answer in CCHChordData::m_mChordSecondInterval.
     */
    INTERVALS GetChordSecondInterval( CHORD_TYPES p_eChord ) const;

    /*! \brief Return the interval of the third chord tone from the root.
     *
     * \param p_eChord The type of chord you are looking up.
     *
     * Looks up its answer in CCHChordData::m_mChordThirdInterval.
     */
    INTERVALS GetChordThirdInterval( CHORD_TYPES p_eChord ) const;

```

```

    /*! \brief Return a string containing the name of the pitch.
    *
    * \param p_ePitch      The pitch who's name you are looking up.
    *
    * Looks up its answer in CCHChordData::m_mPitchNames.
    */
    CString GetPitchName( PITCHES p_ePitch ) const;

    /*! \brief Return the sum of a pitch and an interval.
    *
    * \param p_ePitch      The base pitch to start from.
    * \param p_eInterval   The interval to add to it.
    *
    * This function takes care of the fact that values must wrap around
    * in the sequence Ab, A, Bb, B, C, Db, D, Eb, E, F, Gb, G, Ab, A,
    * Bb, B, etc.
    * For example, a C + a perfect fifth results in a G.
    * A G + a perfect fifth results in a D.
    */
    PITCHES AddIntervalToPitch( PITCHES p_ePitch,
                                INTERVALS p_eInterval ) const;

    /*! \brief Return the interval between two pitches.
    *
    * \param p_eFirst      The "lower" note in the interval.
    * \param p_eSecond     The "higher" note in the interval.
    *
    * This function takes care of the fact that compound intervals are
    * just a standard interval plus one or more octaves.
    * For example, the interval between C and G is a perfect fifth.
    * The interval between a D and a G is also a perfect fifth.
    */
    INTERVALS GetInterval( PITCHES p_eFirst, PITCHES p_eSecond ) const;

private:

    /*! \brief A hash table associating chord types with their names.
    */
    CMap< CHORD_TYPES, CHORD_TYPES, CString, LPCTSTR > m_mChordNames;

    /*! \brief A hash table of chord types to their first interval.
    *
    * This is going to be the interval between the root and first tone
    * of the chord.
    * For example, the first interval in a CHORD_TYPE_MAJOR7 is a
    * INTERVAL_MAJOR3.
    */
    CMap< CHORD_TYPES, CHORD_TYPES, INTERVALS, INTERVALS >
        m_mChordFirstInterval;

    /*! \brief A hash table of chord types to their second interval.
    */
    CMap< CHORD_TYPES, CHORD_TYPES, INTERVALS, INTERVALS >
        m_mChordSecondInterval;

    /*! \brief A hash table of chord types to their third interval.
    */
    CMap< CHORD_TYPES, CHORD_TYPES, INTERVALS, INTERVALS >
        m_mChordThirdInterval;

    /*! \brief A hash table associating pitches with their names.
    */
    CMap< PITCHES, PITCHES, CString, LPCTSTR > m_mPitchNames;
};

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Everything between this and the corresponding #ifndef
// CHCS491 CHOCHORDBASE CHORDDATA H will only be included once. //
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

#endif

```

```

/*! \file CHCS491\CHChordBase\Progression.h
*   \brief Defines a class for a chord and a progression of chords.
*
*   Documented in the format of Doxygen.
*/

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// This is a header file that will be #included by other header //
// files. To prevent multiple declarations of the same things, //
// we must make sure the actual code is only ever added once //
// for each compilation. This check is performed by the use of //
// this macro, which will be defined when the contents of this //
// file are added during a compilation. When the macro is //
// defined, the contents of this file do not include any code. //
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

#ifdef __CHCS491_CHCHORDBASE_PROGRESSION_H
#define __CHCS491_CHCHORDBASE_PROGRESSION_H

#include "ChordData.h" // Include the definitions
                        // and the class to use
                        // for looking things up.

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// For the sake of efficient memory usage, both a chord type and //
// a pitch are going to be stored in an 8-bit character. This //
// is acceptable because the values of pitches range from //
// 1 - 12 and the values of chord types are 32, 64, or 96. //
// An 8-bit number in binary has 8 digits. The pitch will be //
// stored in the first 4 digits, while the chord type will be //
// stored in the sixth and seventh digits. Thus they can both //
// be stored in the same small data structure without //
// overlapping. These masks are used so you can edit one part //
// of the number and not the other. //
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

#define TYPE_MASK ( 32 | 64 ) // In binary this is
                               // 01100000.
#define ROOT_MASK ( 1 | 2 | 4 | 8 ) // In binary this is
                                     // 00001111.

/*! \brief A class to represent a single chord.
*
*   \nosubgrouping
*/
class CHCHORDBASECLASS CCHChord
{
public:
    CCHChord();

    /*! \brief Set the root of the chord.
    *
    *   \param p_eRoot      The new root value.
    *
    *   Returns the pitch so they can be chained.
    */
    PITCHES SetRoot( PITCHES p_eRoot );

    /*! \brief Set the type of the chord.
    *
    *   \param p_eType      The new type value.
    *
    *   Returns the type so they can be chained.
    */
    CHORD_TYPES SetType( CHORD_TYPES p_eType );

    /*! \brief Set the root and type of the chord.
    *
    *   \param p_eRoot      The new root value.

```

```

    * \param p_eType          The new type value.
    */
void SetChord( PITCHES p_eRoot,
               CHORD_TYPES p_eType );

/*! \brief Return the root of the chord.
    */
PITCHES GetRoot() const;

/*! \brief Return the type of the chord.
    */
CHORD_TYPES GetType() const;

/*! \brief Return the "third" of the chord.
    */
PITCHES GetSecondNote() const;

/*! \brief Return the "fifth" of the chord.
    */
PITCHES GetThirdNote() const;

/*! \brief Return the "seventh" of the chord.
    */
PITCHES GetFourthNote() const;

/*! \brief Print a representation of the chord to a string.
    */
CString Print() const;
protected:

/*! \brief The actual chord information.
    *
    * To save space, bits 0 through 3 hold the root pitch and
    * bits 5 and 6 hold the type of chord. Bitwise arithmetic
    * is used to extract these individual parts.
    */
    unsigned char m_cData;
};

class CCHShortProgression; // Forward declaration.

/*! \brief A class to represent a 64-chord progression.
    *
    * \nosubgrouping
    */
class CHCHORDBASECLASS CCHProgression
{
public:

    /*! \brief Set the root of a specific chord.
        *
        * \param p_iIndex      The index of the chord to be adjusted.
        * \param p_eRoot      The new value.
        */
    PITCHES SetRoot( int p_iIndex,
                     PITCHES p_eRoot );

    /*! \brief Set the type of a specific chord.
        *
        * \param p_iIndex      The index of the chord to be adjusted.
        * \param p_eType      The new value.
        */
    CHORD_TYPES SetType( int p_iIndex,
                         CHORD_TYPES p_eType );

    /*! \brief Set the root and type of a specific chord.
        *
        * \param p_iIndex      The index of the chord to be adjusted.
        * \param p_eRoot      The new root value.
        * \param p_eType      The new type value.
        */
    void SetBoth( int p_iIndex,
                  PITCHES p_eRoot, CHORD_TYPES p_eType );

```



```

    /*! \brief Get the root of a specific chord.
    *
    * \param p_iIndex      The index of the chord to be read.
    */
    PITCHES GetRoot( int p_iIndex ) const;

    /*! \brief Get the type of a specific chord.
    *
    * \param p_iIndex      The index of the chord to be read.
    */
    CHORD_TYPES GetType( int p_iIndex ) const;

    /*! \brief Get a copy of a specific chord.
    *
    * \param p_iIndex      The index of the chord to be read.
    *
    * The CCHChord that is returned may not be altered.
    */
    const CCHChord GetChord( int p_iIndex ) const;

    /*! \brief Print the progression to a string.
    */
    CString Print() const;

    /*! \brief Print the composition of the first 8 chords to a string.
    */
    CString PrintNotes() const;

    /*! \brief Replace the four chords at p_iIndex with the short progression.
    *
    * \param p_cProg        The 4-chord progression to copy from.
    * \param p_iIndex      The index of the first chord to be replaced.
    */
    void AppendShort( CCHShortProgression* p_cProg, int p_iIndex );
protected:

    /*! \brief The actual array of chords.
    */
    CCHChord m_cChords[ 64 ];
};

    /*! \brief A class to represent a 4-chord progression.
    *
    * \nosubgrouping
    */
    class CHCHORDBASECLASS CCHShortProgression
    {
    public:

        /*! \brief Set the root of a specific chord.
        *
        * \param p_iIndex      The index of the chord to be adjusted.
        * \param p_eRoot      The new value.
        */
        PITCHES SetRoot( int p_iIndex,
            PITCHES p_eRoot );

        /*! \brief Set the type of a specific chord.
        *
        * \param p_iIndex      The index of the chord to be adjusted.
        * \param p_eType      The new value.
        */
        CHORD_TYPES SetType( int p_iIndex,
            CHORD_TYPES p_eType );

        /*! \brief Set the root and type of a specific chord.
        *
        * \param p_iIndex      The index of the chord to be adjusted.
        * \param p_eRoot      The new root value.
        * \param p_eType      The new type value.
        */
        void SetBoth( int p_iIndex,

```

```

    PITCHES p_eRoot, CHORD_TYPES p_eType );

/*! \brief Get the root of a specific chord.
 *
 * \param p_iIndex      The index of the chord to be read.
 */
PITCHES GetRoot( int p_iIndex ) const;

/*! \brief Get the type of a specific chord.
 *
 * \param p_iIndex      The index of the chord to be read.
 */
CHORD_TYPES GetType( int p_iIndex ) const;

/*! \brief Get a copy of a specific chord.
 *
 * \param p_iIndex      The index of the chord to be read.
 *
 * The CCHChord that is returned may not be altered.
 */
const CCHChord GetChord( int p_iIndex ) const;

protected:

    /*! \brief The actual array of chords.
     */
    CCHChord m_cChords[ 4 ];
};

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Everything between this and the corresponding #ifndef          //
// CHCS491_CHOCHORDBASE_PROGRESSION_H will only be included      //
// once.                                                          //
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

#endif

```

```

/*! \file CHCS491\CHChordBase\ChordData.cpp
* \brief Defines a class for looking up musical information.
*
* Documented in the format of Doxygen.
* See also CCHChordData.
*/

#include "StdAfx.h" // This include is necessary

// for all cpps in an MFC
// project.
#include "ChordData.h" // This include provides the
// definition of this
// class.

////////////////////////////////////
// The block of preprocessor directives below assists the
// compiler in producing useful debugging information.
// Specifically, it allows you to track where exactly any
// dynamic memory allocations occur and which ones do not get
// freed properly.
////////////////////////////////////

#ifdef _DEBUG // This macro is defined in
// the project file for
// all debug builds. For
// release builds, the
// rest of this will be
// ignored.

#define new DEBUG_NEW // Replace all instances of
// "new" with "DEBUG_NEW",
// which is defined by
// MFC to allocate memory
// and store useful
// information with it.

#undef THIS_FILE // Make sure the symbol
// THIS_FILE does not
// have any existing
// meaning.

static char THIS_FILE[] = __FILE__; // The __FILE__ macro is
// automatically defined
// to be the name of the
// current source file.
// This value is stored in
// a string THIS_FILE,
// which has scope limited
// to this file.

#endif // End the preceding #ifdef.

////////////////////////////////////
// The functions of this class are primarily used for looking up
// attributes of a specific chord. The lookup process is
// expedited by a series of hash tables ( maps ) that relate a
// musical entity with some attribute of it. The constructor
// merely fills all of these hash tables with the appropriate
// data so it may be later retrieved. In the case of name
// attributes, the names have been padded to fit all possible
// entries and leave reasonable space after them.
////////////////////////////////////

CCHChordData::CCHChordData()
{
    m_mChordNames.SetAt( // When there is no chord,
        CHORD_TYPE_NONE, // you just want padding.
        " " );
    m_mChordNames.SetAt( // A major chord is denoted
        CHORD_TYPE_MAJOR7, // by "Ma7".
        "Ma7 " );
    m_mChordNames.SetAt( // A minor chord is denoted

```

```

    CHORD_TYPE MINOR7,
    "-7" );
mChordNames.SetAt(
    CHORD_TYPE DOMINANT7,
    "7" );

mChordFirstInterval.SetAt(
    CHORD_TYPE MAJOR7,
    INTERVAL MAJOR3 );
mChordFirstInterval.SetAt(
    CHORD_TYPE MINOR7,
    INTERVAL MINOR3 );
mChordFirstInterval.SetAt(
    CHORD_TYPE DOMINANT7,
    INTERVAL MAJOR3 );

mChordSecondInterval.SetAt(
    CHORD_TYPE MAJOR7,
    INTERVAL PERFECT5 );
mChordSecondInterval.SetAt(
    CHORD_TYPE MINOR7,
    INTERVAL PERFECT5 );
mChordSecondInterval.SetAt(
    CHORD_TYPE DOMINANT7,
    INTERVAL PERFECT5 );

mChordThirdInterval.SetAt(
    CHORD_TYPE MAJOR7,
    INTERVAL MAJOR7 );
mChordThirdInterval.SetAt(
    CHORD_TYPE MINOR7,
    INTERVAL MINOR7 );
mChordThirdInterval.SetAt(
    CHORD_TYPE DOMINANT7,
    INTERVAL MINOR7 );

mPitchNames.SetAt( PITCH NONE, " " );
mPitchNames.SetAt( PITCH C, "C" );
mPitchNames.SetAt( PITCH Db, "Db" );
mPitchNames.SetAt( PITCH D, "D" );
mPitchNames.SetAt( PITCH Eb, "Eb" );
mPitchNames.SetAt( PITCH E, "E" );
mPitchNames.SetAt( PITCH F, "F" );
mPitchNames.SetAt( PITCH Gb, "Gb" );
mPitchNames.SetAt( PITCH G, "G" );
mPitchNames.SetAt( PITCH Ab, "Ab" );
mPitchNames.SetAt( PITCH A, "A" );
mPitchNames.SetAt( PITCH Bb, "Bb" );
mPitchNames.SetAt( PITCH B, "B" );
}

```

```

// by "-7".
// A dominant chord is
// denoted by "7".

// The first interval in a
// major7 chord is a
// major third.
// The first interval in a
// minor7 chord is a
// minor third.
// The first interval in a
// dominant chord is a
// major third.

// The second interval in a
// major7 chord is a
// perfect fifth.
// The second interval in a
// minor7 chord is a
// perfect fifth.
// The second interval in a
// dominant chord is a
// perfect fifth.

// The third interval in a
// major7 chord is a
// major seventh.
// The third interval in a
// minor 7 chord is a
// minor seventh.
// The third interval in a
// dominant chord is a
// minor seventh.

```

```

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// This function takes a chord type and returns a string that
// contains the suffix that will be used for chords of that
// type when displaying them.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

```

```

CString CCHChordData::GetChordName(
    CHORD_TYPES p_eChord ) const
{
    CString l_sTemp;
    if( !mChordNames.Lookup(
        p_eChord,
        l_sTemp ) )
    {
        ASSERT( FALSE );
    }
    return l_sTemp;
}

```

```

// A string to return.
// Attempt to retrieve the
// string value from the
// hash table.

// If this fires, either
// the table is corrupt,
// or p_eChord was an
// invalid value.

// Return the string.

```



```

}

/////////////////////////////////////////////////////////////////
// This function takes a chord type and returns the interval from //
// the root of the chord to the second note. //
/////////////////////////////////////////////////////////////////

INTERVALS CCHChordData::GetChordFirstInterval(
    CHORD_TYPES p_eChord ) const
{
    INTERVALS l_eTemp; // An interval to return.
    if( !m_mChordFirstInterval.Lookup( // Attempt to retrieve the
        p_eChord, // interval value from
        l_eTemp ) ) // the hash table.
    {
        ASSERT( FALSE ); // If this fires, either
                        // the table is corrupt,
                        // or p_eChord was an
                        // invalid value.
    }
    return l_eTemp; // Return the interval.
}

/////////////////////////////////////////////////////////////////
// This function takes a chord type and returns the interval from //
// the root of the chord to the third note. //
/////////////////////////////////////////////////////////////////

INTERVALS CCHChordData::GetChordSecondInterval(
    CHORD_TYPES p_eChord ) const
{
    INTERVALS l_eTemp; // An interval to return.
    if( !m_mChordSecondInterval.Lookup( // Attempt to retrieve the
        p_eChord, // interval value from
        l_eTemp ) ) // the hash table.
    {
        ASSERT( FALSE ); // If this fires, either
                        // the table is corrupt,
                        // or p_eChord was an
                        // invalid value.
    }
    return l_eTemp; // Return the interval.
}

/////////////////////////////////////////////////////////////////
// This function takes a chord type and returns the interval from //
// the root of the chord to the fourth note. //
/////////////////////////////////////////////////////////////////

INTERVALS CCHChordData::GetChordThirdInterval(
    CHORD_TYPES p_eChord ) const
{
    INTERVALS l_eTemp; // An interval to return.
    if( !m_mChordThirdInterval.Lookup( // Attempt to retrieve the
        p_eChord, // interval value from
        l_eTemp ) ) // the hash table.
    {
        ASSERT( FALSE ); // If this fires, either
                        // the table is corrupt,
                        // or p_eChord was an
                        // invalid value.
    }
    return l_eTemp; // Return the interval.
}

/////////////////////////////////////////////////////////////////

```



```

// This function takes a pitch and returns a representation of it //
// as a string for printing to the screen. //
/////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////

CString CCHChordData::GetPitchName(
    PITCHES p_ePitch ) const
{
    CString l_sTemp;
    if( !m_mPitchNames.Lookup(
        p_ePitch,
        l_sTemp ) )
    {
        ASSERT( FALSE );
    }
    return l_sTemp;
}

/////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////
// This function takes an initial pitch and an interval and //
// the pitch that is higher than the initial pitch by that //
// interval. Adjustments are made for when the pitch would //
// wrap back around to the beginning. //
/////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////

PITCHES CCHChordData::AddIntervalToPitch(
    PITCHES p_ePitch,
    INTERVALS p_eInterval ) const
{
    int l_iTemp = (int)p_ePitch;
    l_iTemp += (int)p_eInterval;

    if( l_iTemp > PITCH_LAST )

    {
        l_iTemp -= PITCH_LAST;
    }
    return (PITCHES)l_iTemp;
}

/////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////
// This function takes an initial pitch and an additional pitch //
// that should be higher than it. It then returns the simple //
// interval between those pitches. Pitches that result in a //
// compound interval are simplified by octave. //
/////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////

INTERVALS CCHChordData::GetInterval(
    PITCHES p_eFirst,
    PITCHES p_eSecond ) const

```

```

{
    int l_iInterval = (int)p_eSecond;
    l_iInterval -= (int)p_eFirst;
    if( l_iInterval < INTERVAL_FIRST )
    {
        l_iInterval += (int)INTERVAL_LAST;
    }
    return (INTERVALS)l_iInterval;
}

// Convert the higher pitch
// to a simple integer.
// Subtract the lower pitch.

// This may result in a
// negative interval. For
// example, if the first
// pitch is G and the
// second is C, this will
// currently be 1-8 = -7.
// This value is adjusted
// by adding an octave to
// it. So -7 becomes 5,
// which is a perfect
// fourth.

// Add an octave to the
// interval.

// Cast this adjusted value
// back to an interval and
// return it.

```

```

/*! \file CHCS491\CHChordBase\Progression.cpp
*   \brief Defines a class for a chord and a progression of chords.
*
*   Documented in the format of Doxygen.
*/

#include "StdAfx.h"                                // This include is necessary

                                                    // for all cpps in an MFC
                                                    // project.
#include "Progression.h"                            // This is where my classes
                                                    // are declared.

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// The block of preprocessor directives below assists the
// compiler in producing useful debugging information.
// Specifically, it allows you to track where exactly any
// dynamic memory allocations occur and which ones do not get
// freed properly.
// See this section in CHCS491\CHChordBase\ChordData.cpp for a
// more detailed explanation of how it works.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

static CCHChordData g_cData;                        // An object to look up
                                                    // musical data. This
                                                    // must be a static global
                                                    // variable because static
                                                    // members are not allowed
                                                    // on things that get
                                                    // exported, and it would
                                                    // be inefficient for each
                                                    // chord to have its own.

CCHChord::CCHChord()
{
    m_cData = NULL;
}

PITCHES CCHChord::SetRoot(
    PITCHES p_eRoot )
{
    m_cData = ( m_cData & TYPE_MASK )           // Save the old type and
    | ( p_eRoot & ROOT_MASK );                  // replace the root.
    return p_eRoot;                             // Send the value back.
}

CHORD_TYPES CCHChord::SetType(
    CHORD_TYPES p_eType )
{
    m_cData = ( m_cData & ROOT_MASK )           // Save the old root and
    | ( p_eType & TYPE_MASK );                  // replace the type.
    return p_eType;                             // Send the value back.
}

void CCHChord::SetChord(
    PITCHES p_eRoot,
    CHORD_TYPES p_eType )
{
    m_cData = ( p_eType & TYPE_MASK )           // Replace both halves.
    | ( p_eRoot & ROOT_MASK );
}

PITCHES CCHChord::GetRoot() const
{

```

```

    return (PITCHES)( m_cData & ROOT_MASK );    // Return the value.
}

CHORD_TYPES CCHChord::GetType() const
{
    return (CHORD_TYPES)( m_cData & TYPE_MASK ); // Return the value.
}

PITCHES CCHChord::GetSecondNote() const
{
    return g_cData.AddIntervalToPitch(           // Return the sum of the
        GetRoot(),                               // root and the interval
        g_cData.GetChordFirstInterval(          // to the second note of
            GetType() ) );                       // the chord.
}

PITCHES CCHChord::GetThirdNote() const
{
    return g_cData.AddIntervalToPitch(           // Return the sum of the
        GetRoot(),                               // root and the interval
        g_cData.GetChordSecondInterval(         // to the third note of
            GetType() ) );                       // the chord.
}

PITCHES CCHChord::GetFourthNote() const
{
    return g_cData.AddIntervalToPitch(           // Return the sum of the
        GetRoot(),                               // root and the interval
        g_cData.GetChordThirdInterval(          // to the fourth note of
            GetType() ) );                       // the chord.
}

CString CCHChord::Print() const
{
    CString x =                                  // Return a string of
        g_cData.GetPitchName( GetRoot() )       // the concatenation of
        + g_cData.GetChordName( GetType() );    // the root and suffix.
    return x;
}

PITCHES CCHProgression::SetRoot(
    int p_iIndex,
    PITCHES p_eRoot )
{
    if( p_iIndex >= 0 && p_iIndex < 64 )        // Do nothing if the index
                                                // is not valid.
    {
        m_cChords[ p_iIndex ].SetRoot(         // Set the root of the
            p_eRoot );                          // appropriate chord.
    }
    return p_eRoot;                             // Return the root.
}

CHORD_TYPES CCHProgression::SetType(
    int p_iIndex,
    CHORD_TYPES p_eType )
{
    if( p_iIndex >= 0 && p_iIndex < 64 )        // Do nothing if the index
                                                // is not valid.
    {
        m_cChords[ p_iIndex ].SetType(         // Set the type of the
            p_eType );                          // appropriate chord.
    }
    return p_eType;                             // Return the root.
}

void CCHProgression::SetBoth(
    int p_iIndex,
    PITCHES p_eRoot,

```

```

    CHORD_TYPES p_eType )
{
    if( p_iIndex >= 0 && p_iIndex < 64 )           // Do nothing if the index
                                                    // is not valid.
    {
        m_cChords[ p_iIndex ].SetRoot(           // Set the root of the
            p_eRoot );                           // appropriate chord.
        m_cChords[ p_iIndex ].SetType(           // Set the type of the
            p_eType );                           // appropriate chord.
    }
}

PITCHES CCHProgression::GetRoot(
    int p_iIndex ) const
{
    if( p_iIndex >= 0 && p_iIndex < 64 )           // Fail if index is invalid.
    {
        return m_cChords[ p_iIndex ].GetRoot();   // Return the root of the
                                                    // appropriate chord.
    }
    return PITCH_NONE;                           // If index was invalid,
                                                    // return an uninitialized
                                                    // pitch.
}

CHORD_TYPES CCHProgression::GetType(
    int p_iIndex ) const
{
    if( p_iIndex >= 0 && p_iIndex < 64 )           // Fail if index is invalid.
    {
        return m_cChords[ p_iIndex ].GetType();   // Return the type of the
                                                    // appropriate chord.
    }
    return CHORD_TYPE_NONE;                       // If index was invalid,
                                                    // return an uninitialized
                                                    // type.
}

const CCHChord CCHProgression::GetChord(
    int p_iIndex ) const
{
    if( p_iIndex >= 0 && p_iIndex < 64 )           // Fail if index is invalid.
    {
        return m_cChords[ p_iIndex ];             // Return a copy of the
                                                    // appropriate chord.
    }
    return CCHChord();                           // If index was invalid,
                                                    // return an uninitialized
                                                    // chord.
}

CString CCHProgression::Print() const
{
    CString ret;                                  // A string to return.

    for( int i = 0; i < 8; i++ )                 // There will be 8 staves.
    {
        for( int j = 0; j < 8; j++ )             // Each staff will hold 8
                                                    // chord positions, or 4
                                                    // bars.
        {
            ret += m_cChords                     // Print the current chord
                [ j + ( 8 * i ) ].Print();        // into the string.

            if( j % 2 == 1 )                      // If this is an
                                                    // odd-numbered position:
            {
                ret += " | ";                    // Insert a barline.
            }
            else
            {
                ret += " ";                      // Otherwise:
            }
            ret += " ";                          // Insert spaces.
        }
    }
}

```



```

    }
    ret += "\n";
}

return ret;
}

CString CCHProgression::PrintNotes() const
{
    CString ret;
    CCHProgression l_cOther;

    for( int i = 0; i < 8; i++ )
    {
        if( m_cChords[ i ].GetRoot() )
        {
            l_cOther.SetRoot(
                i,
                m_cChords[ i ].GetFourthNote() );

            l_cOther.SetRoot(
                i + 8,
                m_cChords[ i ].GetThirdNote() );

            l_cOther.SetRoot(
                i + 16,
                m_cChords[ i ].GetSecondNote() );

            l_cOther.SetRoot(
                i + 24,
                m_cChords[ i ].GetRoot() );
        }
    }

    for( int k = 0; k < 4; k++ )
    {
        for( int j = 0; j < 8; j++ )

        {
            ret += l_cOther.m_cChords
                [ j + ( 8 * k ) ].Print();

            if( j % 2 == 1 )
            {
                ret += " | ";
            }
            else
            {
                ret += "   ";
            }
        }
        ret += "\n";
    }

    return ret;
}

void CCHProgression::AppendShort(
    CCHShortProgression* p_cProg,
    int p_iIndex )
{
    if( p_iIndex >= 0 && p_iIndex <= 60 )

```

```

// Move to the next line for
// the next staff.

```

```

// Return the string.

```

```

// A string to return.
// A temporary progression
// that will actually
// contain notes rather
// than chords.

```

```

// Loop through each of the
// chords to expand.

```

```

// Only need new chords.

```

```

// The first 8 roots in the
// temporary progression
// are the sevenths of the
// first 8 chords.

```

```

// The next 8 roots in the
// temporary progression
// are the fifths of the
// first 8 chords.

```

```

// The next 8 roots in the
// temporary progression
// are the thirds of the
// first 8 chords.

```

```

// The next 8 roots in the
// temporary progression
// are the roots of the
// first 8 chords.

```

```

// There will be 4 notes.

```

```

// Each staff will hold 8
// chord positions, or 4
// bars.

```

```

// Print the current note
// into the string.

```

```

// If this is an
// odd-numbered position:

```

```

// Insert a barline.

```

```

// Otherwise:

```

```

// Insert spaces.

```

```

// Move to the next line for
// the next series of
// notes.

```

```

// Return the string.

```

```

// Make sure the index is
// valid to add 4 to it.

```

```

{
    SetBoth(
        p_iIndex,
        p_cProg->GetRoot( 0 ),
        p_cProg->GetType( 0 ) );
    SetBoth(
        p_iIndex + 1,
        p_cProg->GetRoot( 1 ),
        p_cProg->GetType( 1 ) );
    SetBoth(
        p_iIndex + 2,
        p_cProg->GetRoot( 2 ),
        p_cProg->GetType( 2 ) );
    SetBoth(
        p_iIndex + 3,
        p_cProg->GetRoot( 3 ),
        p_cProg->GetType( 3 ) );
}
}

```

```

// Set my chords starting at
// the index to the 4
// chords of the short
// progression.

```

```

PITCHES CCHShortProgression::SetRoot(
    int p_iIndex,
    PITCHES p_eRoot )

```

```

{
    if( p_iIndex >= 0 && p_iIndex < 4 )
    {
        m_cChords[ p_iIndex ].SetRoot(
            p_eRoot );
    }
    return p_eRoot;
}

```

```

// Do nothing if the index
// is not valid.

// Set the root of the
// appropriate chord.

// Return the root.

```

```

CHORD_TYPES CCHShortProgression::SetType(
    int p_iIndex,
    CHORD_TYPES p_eType )

```

```

{
    if( p_iIndex >= 0 && p_iIndex < 4 )
    {
        m_cChords[ p_iIndex ].SetType(
            p_eType );
    }
    return p_eType;
}

```

```

// Do nothing if the index
// is not valid.

// Set the type of the
// appropriate chord.

// Return the root.

```

```

void CCHShortProgression::SetBoth(
    int p_iIndex,
    PITCHES p_eRoot,
    CHORD_TYPES p_eType )

```

```

{
    if( p_iIndex >= 0 && p_iIndex < 4 )
    {
        m_cChords[ p_iIndex ].SetRoot(
            p_eRoot );
        m_cChords[ p_iIndex ].SetType(
            p_eType );
    }
}

```

```

// Do nothing if the index
// is not valid.

// Set the root of the
// appropriate chord.
// Set the type of the
// appropriate chord.

```

```

PITCHES CCHShortProgression::GetRoot(
    int p_iIndex ) const

```

```

{
    if( p_iIndex >= 0 && p_iIndex < 4 )
    {
        return m_cChords[ p_iIndex ].GetRoot();
    }
}

```

```

// Fail if index is invalid.

// Return the root of the
// appropriate chord.

```

```

return PITCH_NONE;                                     // If index was invalid,
                                                         // return an uninitialized
                                                         // pitch.
}

CHORD_TYPES CCHShortProgression::GetType(
    int p_iIndex ) const
{
    if( p_iIndex >= 0 && p_iIndex < 4 )                // Fail if index is invalid.
    {
        return m_cChords[ p_iIndex ].GetType();        // Return the type of the
                                                         // appropriate chord.
    }
    return CHORD_TYPE_NONE;                             // If index was invalid,
                                                         // return an uninitialized
                                                         // type.
}

const CCHChord CCHShortProgression::GetChord(
    int p_iIndex ) const
{
    if( p_iIndex >= 0 && p_iIndex < 4 )                // Fail if index is invalid.
    {
        return m_cChords[ p_iIndex ];                  // Return a copy of the
                                                         // appropriate chord.
    }
    return CCHChord();                                 // If index was invalid,
                                                         // return an uninitialized
                                                         // chord.
}

```

```

/*! \file CHCS491\CHChordsSearch\ChordSearchTree.h
 *  \brief Defines a searching algorithm for finding chord progressions.
 *
 *  Documented in the format of Doxygen.
 */

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// This is a header file that will be #included by other header //
// files. To prevent multiple declarations of the same things, //
// we must make sure the actual code is only ever added once //
// for each compilation. This check is performed by the use of //
// this macro, which will be defined when the contents of this //
// file are added during a compilation. When the macro is //
// defined, the contents of this file do not include any code. //
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

#ifdef __CHCS491_CHCHORDSSEARCH_CHORDSEARCHTREE_H
#define __CHCS491_CHCHORDSSEARCH_CHORDSEARCHTREE_H

#include "Progression.h" // The base classes for
                        // chords and progressions
#include "Afxttempl.h" // of chords are here.
                        // The MFC collection
#include <iostream> // classes are here.
                        // The stream i/o classes
                        // are here.
using namespace std;

/*! \brief A single node in the depth first search.
 */
class CCHChordSearchNode : public CCHShortProgression
{
public:

    /*! \brief Return the first uninitialized position or -1 for none.
     */
    int GetCurrIndex() const;

    /*! \brief Copy the contents of an existing node to myself.
     *
     * \param p_cOther A pointer to the node to be copied.
     */
    void CopyFromOther( CCHChordSearchNode* p_cOther );
};

/*! \brief Add evaluation functionality to the 32-bar chord progression.
 *
 * \nosubgrouping
 */
class CCHChordEvalNode : public CCHProgression
{
friend class CCHChordSearchTree; // Allow the tree I am in to
                                // access me.
public:

    /*! \brief Constructor for the node.
     */
    CCHChordEvalNode();

    /*! \brief Copy the information in p_nOther into myself.
     *
     * \param p_nOther Another node to copy from.
     * \param p_iStart The first position to copy.
     * \param p_iEnd The last position to copy.
     * \param p_iOffset The difference between positions in this
     *                  and l_nOther.
     *
     * The real purpose for this function is to move things around.
     * The new locations will be p_iStart -- p_iEnd.
     * The old locations will be p_iStart + p_iOffset -- p_iEnd + p_iOffset.
     */

```

```

void CopyFromOther( CCHChordEvalNode p_nOther, int p_iStart = 0,
    int p_iEnd = 64, int p_iOffset = 0 );

/*! \brief Assign a numerical value to how good this progression is.
 *
 * \param p_iStart      The 0-based index of the first chord to
 *                      be evaluated.
 * \param p_iEnd        The 0-based index of the first chord to
 *                      not be evaluated.
 *
 * Uses a variety of metrics about chord progressions to evaluate it.
 * Running this function on all 64 chords will take quite a long time.
 */
int Evaluate( short p_iStart, short p_iEnd );
};

/*! \brief A class to search for an optimal chord progression.
 *
 * \nosubgrouping
 */
class CCHChordSearchTree
{
public:
    /*! \brief Construct a new search tree.
     *
     * This doesn't do anything.
     */
    CCHChordSearchTree();

    /*! \brief Destruct a search tree.
     *
     * This doesn't do anything.
     */
    ~CCHChordSearchTree();

    /*! \brief Print out the best chord progression that was found.
     *
     * \param p_iNum      The number of chords to print out.
     */
    void PrintBest( int p_iNum = 64 );

    /*! \brief Perform a depth-first search of all combinations of 4 chords.
     *
     * The resulting progressions will be in m_lQueue.
     */
    void DepthFirstSearch();

    /*! \brief Expand the 4 chord progressions to something longer.
     *
     * This is a somewhat nasty hack, but we can't afford the
     * resources to search further.
     */
    void ExpandNodes();

    /*! \brief Return a pointer to the best progression.
     */
    CCHProgression* GetBest();

private:
    /*! \brief A queue of nodes that are not filled.
     *
     * Used by CCHChordEvalNode::DepthFirstSearch().
     */
    CList<CCHChordSearchNode *, CCHChordSearchNode *> m_lQueue;

    /*! \brief A queue of nodes that are filled ( for now ).
     *
     * Used by CCHChordEvalNode::DepthFirstSearch().
     */
    CList<CCHChordSearchNode *, CCHChordSearchNode *> m_lFinished;

    /*! \brief A default node to start from.
     */

```



```

CCHChordEvalNode m_nDefaults;

/*! \brief The best node found thus far.
 *
 * Used by CCHChordEvalNode::DepthFirstSearch().
 */
CCHChordEvalNode m_nBest;

/*! \brief Empty the queue so the search can run again.
 */
void EmptyQueue();

/*! \brief Add another four chords to the best progression so far.
 *
 * /param p_iIndex      The 1-based index of the two-bar phrase.
 */
void AddFour( int p_iIndex );
};

/////////////////////////////////////////////////////////////////
// Everything between this and the corresponding #ifndef    //
// CHCS491_CHOCHORDSSEARCH_CHORDSEARCHTREE_H will only be  //
// include once.                                           //
/////////////////////////////////////////////////////////////////

#endif

```

```

/*! \file CHCS491\CHRandomChords\CHRandomChords.cpp
* \brief A simple test of producing a chord progression randomly.
*
* Documented in the format of Doxygen.
*/

#include "stdafx.h" // This include is necessary
// for all source files in
// an MFC project.
#include "ChordData.h" // Include the engine for
// looking up chord
// information.
#include "Progression.h" // Include the chord
// progression class.
using namespace std; // Defines how to qualify
// things in the included
// files.

////////////////////////////////////
// The block of preprocessor directives below assists the
// compiler in producing useful debugging information.
// Specifically, it allows you to track where exactly any
// dynamic memory allocations occur and which ones do not get
// freed properly.
// See this section in CHCS491\CHChordBase\ChordData.cpp for a
// more detailed explanation of how it works.
////////////////////////////////////

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

CWinApp theApp; // A global object that
// represents the
// application itself.

int _tmain( // This is the "main-like"
int argc, // function for MFC
TCHAR* argv[], // console applications.
TCHAR* envp[] )
{
int nRetCode = 0; // A success or failure
// value to return.

if( !AfxWinInit( // Initialize the process.
::GetModuleHandle( NULL ),
NULL,
::GetCommandLine(),
0 ) )
{
cerr << _T( "Fatal Error: MFC " ) // Alert the user if MFC
<< _T( "initialization failed." ) // could not be
<< _T( "\nYou will need to get " ) // initialized.
<< _T( "a new version of MFC to " )
<< _T( "run this program." )
<< endl;
nRetCode = 1; // Return an error code.
}
else
{
srand( time( NULL ) ); // Initialize the random
// number generator.
cout << "Random 32-bar chord " // Explain the output.
<< "progression:\n\n";

CCHChordData d1; // This is the engine for
// looking up musical
// information.

```

```

CCHProgression l_cProg;                                // The chord progression I
                                                         // am building.

for( int k = 0; k < 64; k++ )                          // Move sequentially through
                                                         // each of the 64
                                                         // positions. k will be
                                                         // an index into arrays
                                                         // l_iChordPitches and
                                                         // l_iChordTypes.

{
    l_cProg.SetRoot(                                     // Choose randomly from the
    k,                                                     // 12 possible roots.
    (PITCHES)( ( ( rand() %
    ( PITCH_LAST / PITCH_FIRST ) )
    + 1 ) * PITCH_FIRST ) );

    l_cProg.SetType(                                     // Choose randomly from the
    k,                                                     // 3 possible chord types.
    (CHORD_TYPES)( ( ( rand() %
    ( CHORD_TYPE_LAST / CHORD_TYPE_FIRST
    ) ) + 1 ) * CHORD_TYPE_FIRST ) );
}

cout << (LPCSTR)l_cProg.Print();                        // Print the progression.

cout << "\n\nThe components of the chords " // Inform the user what is
<< "for the first 4 bars:\n\n";                     // next in the output.

cout << (LPCSTR)l_cProg.PrintNotes();                // Print out the components
                                                         // of the first 8 chords.

int qwer;                                                // An integer to input.
cout << "\n\nEnter any integer to quit: ";          // Prompt the user.
cin >> qwer;                                             // Wait for this input
                                                         // before exiting.
}

return nRetCode;                                         // Return the success or
                                                         // failure value.
}

```

```

/*! \file CHCS491\CHChordsSearch\MelodyMarkovChain.h
 * \brief Defines an algorithm for generating melodies from Markov chains.
 *
 * Documented in the format of Doxygen.
 */

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// This is a header file that will be #included by other header
// files. To prevent multiple declarations of the same things,
// we must make sure the actual code is only ever added once
// for each compilation. This check is performed by the use of
// this macro, which will be defined when the contents of this
// file are added during a compilation. When the macro is
// defined, the contents of this file do not include any code.
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

#ifndef __CHCS491_CHCHORDSSEARCH_MELODYMARCOVCHAIN_H
#define __CHCS491_CHCHORDSSEARCH_MELODYMARCOVCHAIN_H

#include "Progression.h"

/*! \brief A class to represent a melodic line.
 *
 * This progression will consist of exactly 256 notes, or 32 bars of
 * eighth notes.
 */
class CCHMelody
{
public:
    /*! \brief Sets up the probability matrix.
     */
    CCHMelody();

    /*! \brief Set a pitch value in the sequence.
     *
     * \param p_iPosition The 0-based index of the position to alter.
     * \param p_eValue The new pitch value to set.
     */
    void SetNote( int p_iPosition, PITCHES p_eValue );

    /*! \brief Get a pitch value in the sequence.
     *
     * \param p_iPosition The 0-based index of the position to retrieve.
     */
    PITCHES GetNote( int p_iPosition );

    /*! \brief Fill the sequence as a Markov chain from set probabilities.
     */
    void GenerateChain();

    /*! \brief Print the sequence to a string object.
     */
    CString Print();

    /*! \brief The chord progression on which this melody will fit.
     *
     * This value MUST be set outside the class.
     */
    CCHProgression* m_pChords;
private:
    /*! \brief The actual sequence of pitches.
     */
    PITCHES m_iNotes[ 256 ];

    /*! \brief A matrix of probabilities for generating Markov chains.
     *
     * [ chord type ] [ measure position ] [ previous note ] [ this note ]
     */
    short m_iProbs[ 3 ][ 2 ][ 13 ][ 13 ];

    /*! \brief Choose the next pitch in a Markov chain.

```

```

*
* \param p_eChord      The underlying chord for this note.
* \param p_iPos        The position of the note to choose.
* \param p_ePrev       The pitch of the previous note.
*/
PITCHES Choose( CHORD_TYPES p_eChord, short p_iPos, PITCHES p_ePrev );

/*! \brief A data object to lookup information in.
*/
CCHChordData m_cLookup;
};

/////////////////////////////////////////////////////////////////
// Everything between this and the corresponding #ifndef    //
// CHCS491_CHOCHORDSSEARCH_CHORDSEARCHTREE_H will only be  //
// include once.                                           //
/////////////////////////////////////////////////////////////////

#endif

```



```

/*! \file CHCS491\CHChordsSearch\ChordSearchTree.cpp
 *  \brief Defines a class for finding an optimal chord progression.
 *
 *  Documented in the format of Doxygen.
 *  See also CCHChordEvalNode and CCHChordSearchTree.
 */

#include "StdAfx.h" // This include is necessary

// for all cpps in an MFC
// project.
#include "ChordSearchTree.h" // Declarations of the
// classes that are
// defined in this file.

CCHChordData g_dLookup; // A global object for
// looking up musical
// information.

PITCHES g_iAKey; // The pitch corresponding
// to the major key of the
// A section.
PITCHES g_iBKey; // The pitch corresponding
// to the major key of the
// B section.

/*! \brief Return a value between 0.6 and 1.5 on a somewhat normal curve.
 */
double RandDist()
{
    int l_iInt = rand() % 100; // Randomly generate a
    // number between 0 and
    // 99.

    if( l_iInt < 5 ) // Return a certain number
        return 0.6; // based on the
    else if( l_iInt < 13 ) // distribution.
        return 0.7;
    else if( l_iInt < 23 )
        return 0.8;
    else if( l_iInt < 34 )
        return 0.9;
    else if( l_iInt < 45 )
        return 1;
    else if( l_iInt < 65 )
        return 1.1;
    else if( l_iInt < 76 )
        return 1.2;
    else if( l_iInt < 87 )
        return 1.3;
    else if( l_iInt < 95 )
        return 1.4;
    else
        return 1.5;
}

int CCHChordSearchNode::GetCurrIndex() const
{
    if( GetRoot( 0 ) == PITCH_NONE ) // If there is no chord in
        return 0; // the first position, the
    // first position is where
    // we need to be working.

    if( GetRoot( 1 ) == PITCH_NONE ) // Etc.
        return 1;
    if( GetRoot( 2 ) == PITCH_NONE )
        return 2;
    if( GetRoot( 3 ) == PITCH_NONE )
        return 3;
    return -1;
}

void CCHChordSearchNode::CopyFromOther(

```

```

CCHChordSearchNode *p_cOther )
{
    SetBoth(                                     // Copy the first chord from
        0,                                     // the other node to this.
        p_cOther->GetRoot( 0 ),
        p_cOther->GetType( 0 ) );
    SetBoth( 1,                                 // Etc.
        p_cOther->GetRoot( 1 ),
        p_cOther->GetType( 1 ) );
    SetBoth( 2,
        p_cOther->GetRoot( 2 ),
        p_cOther->GetType( 2 ) );
    SetBoth( 3,
        p_cOther->GetRoot( 3 ),
        p_cOther->GetType( 3 ) );
}

CCHChordEvalNode::CCHChordEvalNode()
{
}

void CCHChordEvalNode::CopyFromOther(
    CCHChordEvalNode p_nOther,
    int p_iStart,
    int p_iEnd,
    int p_iOffset )
{
    for( int i = p_iStart; i < p_iEnd; i++ )    // Iterate through the
                                                // requested positions.
    {
        SetRoot(                               // Set the root in this from
            i,                                  // the offset position of
            p_nOther.GetRoot( i + p_iOffset ) ); // the one to copy.
        SetType(                               // Set the type in this from
            i,                                  // the offset position of
            p_nOther.GetType( i + p_iOffset ) ); // the one to copy.
    }
}

int CCHChordEvalNode::Evaluate(
    short p_iStart,
    short p_iEnd )
{
    int l_iValue = 0;                          // The current value of this
                                                // chord progression.
    int i;                                      // Counter for iterating
                                                // through loops.

    ////////////////////////////////////////////
    // This loop searches for dominant7 chords that resolve //
    // down a perfect fifth or a minor second.             //
    ////////////////////////////////////////////

    for( i = p_iStart; i < p_iEnd - 1; i++ )    // Check all but the last
                                                // chord that we are
                                                // considering.
    {
        if( GetType( i ) == CHORD_TYPE_DOMINANT7 ) // This will have no effect
                                                    // if the studied chord is
                                                    // not a dominant seventh.
        {
            if( GetRoot( i ) == g_dLookup.        // Resolution down a perfect
                AddIntervalToPitch(               // fifth is very good.
                    GetRoot( i + 1 ),
                    INTERVAL_PERFECT5 ) )
                l_iValue += 4;
            else if( GetRoot( i ) == g_dLookup.    // Resolution down a minor
                AddIntervalToPitch(               // second is also good.
                    GetRoot( i + 1 ),
                    INTERVAL_MINOR2 ) )
                l_iValue += 3;
            else
                l_iValue -= 10;                    // A dominant chord that
                                                    // does not resolve sounds
        }
    }
}

```

```

// awful.

if( i % 2 )
    l_iValue += 1;
else
    l_iValue -= 1;
}
}

/////////////////////////////////////////////////////////////////
// This loop searches for II-7 V7 I progressions. This is //
// the primary cadence in jazz.                          //
/////////////////////////////////////////////////////////////////

for( i = p_iStart; i < p_iEnd - 2; i++ )    // Check all but the last
// two chords that we are
// considering.

{
    if( GetType( i ) == CHORD_TYPE_MINOR7 ) // We are concerned only
// with minor7 chords.

    {
        PITCHES l_iKey =
            ( ( i < 32 || i > 47 )
              ? g_iKey : g_iBKey );

        if( GetType( i + 1 ) ==
            CHORD_TYPE_DOMINANT7 &&
            GetRoot( i ) == g_dLookup.
            AddIntervalToPitch(
                GetRoot( i + 2 ),
                INTERVAL_MAJOR2 ) )
        {
            l_iValue += 4;

            // A minor7 chord that
            // functions in a
            // II-7 V7 I progression
            // adds to the cadential
            // feeling.

            if( GetRoot( i ) == g_dLookup.
                AddIntervalToPitch( l_iKey,
                                    INTERVAL_MAJOR3 ) )
                l_iValue += 4;
            else if( GetRoot( i ) == g_dLookup.
                    AddIntervalToPitch( l_iKey,
                                        INTERVAL_MAJOR6 ) )
                l_iValue += 4;
            else if( GetRoot( i ) == g_dLookup.
                    AddIntervalToPitch( l_iKey,
                                        INTERVAL_MAJOR2 ) )
                l_iValue += 3;
        }
        else
            // Otherwise, this chord is
            // not part of a cadence.

            {
                if( GetRoot( i ) == g_dLookup.
                    AddIntervalToPitch( l_iKey,
                                        INTERVAL_MAJOR3 ) )
                    l_iValue += 3;
                else if( GetRoot( i ) == g_dLookup.
                        AddIntervalToPitch( l_iKey,
                                            INTERVAL_MAJOR6 ) )
                    l_iValue += 3;
                else if( GetRoot( i ) == g_dLookup.
                        AddIntervalToPitch( l_iKey,
                                            INTERVAL_MAJOR2 ) )
                    l_iValue += 2;
                else
                    l_iValue -= 9;

                // A III-7 that is not part
                // of a cadence is in the
                // appropriate key but
                // serves no purpose.
                // A VI-7 that is not part
                // of a cadence is in the
                // appropriate key but
                // serves no purpose.
                // A II-7 that is not part
                // of a cadence is in the
                // appropriate key but
                // serves no purpose.
                // Otherwise, this minor7
                // chord is not in the key
                // and it is not in a
                // cadence, which is quite
            }
        }
    }
}

```

```

// awkward.
}
}

////////////////////////////////////
// This loop checks all major7 chords to see if they fit in //
// the key. //
////////////////////////////////////

for( i = p_iStart; i < p_iEnd; i++ ) // Check every chord in
// consideration.
{
    if( GetType( i ) == CHORD_TYPE_MAJOR7 ) // We are looking just at
// major7 chords.
    {
        PITCHES l_iKey = // If you are in the A
        ( ( i < 32 || i > 47 ) // section, you should be
        ? g_iAKey : g_iBKey ); // in the key of the A
// section. Otherwise,
// you should be in the
// key of the B section.
        if( GetRoot( i ) == l_iKey ) // A IMA7 gets 5 points for
            l_iValue += 5; // being in the key.
        else if( GetRoot( i ) == g_dLookup. // A IVMa7 gets 4 points for
            AddIntervalToPitch( l_iKey, // being in the key.
            INTERVAL PERFECT4 ) )
            l_iValue += 4;
        else if( GetRoot( i ) == g_dLookup. // A bIIIMA7 gets 3 points
            AddIntervalToPitch( l_iKey, // for being in the key.
            INTERVAL MINOR3 ) )
            l_iValue += 3;
        else if( GetRoot( i ) == g_dLookup. // A bVIIIMA7 gets 3 points
            AddIntervalToPitch( l_iKey, // for being in the key.
            INTERVAL MINOR7 ) )
            l_iValue += 3;
        else // A major7 chord that is
            l_iValue -= 5; // not in the key gets -5
// points.
    }
}

////////////////////////////////////
// Discourage chords repeating across a barline. //
////////////////////////////////////

for( i = ( p_iStart % 2 ? p_iStart
: p_iStart + 1 ); i < p_iEnd - 1;
i = i + 2 ) // Check each odd-numbered
// chord in consideration.
{
    if( GetRoot( i ) == GetRoot( i + 1 ) && // If the chord is the same
        GetType( i ) == GetType( i + 1 ) ) // as the next chord, it
    { // gets -5 points.
        l_iValue -= 5;
    }
}

////////////////////////////////////
// The song should start in the key. //
////////////////////////////////////
if( p_iEnd == 4 ) // Only do this check when
// choosing the initial
// four chords. Later it
// cannot be changed and
// will just lessen the
// value of other things.
{
    if( GetRoot( 0 ) == g_iAKey && // If the first chord is a
        GetType( 0 ) == CHORD_TYPE_MAJOR7 ) // major chord on the
        l_iValue += 10; // tonic of the key give
// int 10 points.
}

```



```

else if( GetRoot( 2 ) == g_iAKey && // If the third chord is a
    GetType( 2 ) == CHORD_TYPE_MAJOR7 && // major chord on the
    GetRoot( 1 ) == g_dLookup. // tonic of the key and
    AddIntervalToPitch( g_iAKey, // the second chord is a
    INTERVAL PERFECT5 ) && // dominant chord that
    GetType( 1 ) == CHORD_TYPE_DOMINANT7 ) // resolves to it give it
    l_iValue += 9; // 9 points.
else if( GetRoot( 2 ) == g_iAKey && // If the third chord is a
    GetType( 2 ) == CHORD_TYPE_MAJOR7 && // major chord on the
    GetRoot( 1 ) == g_dLookup. // tonic of the key and
    AddIntervalToPitch( g_iAKey, // the second chord is a
    INTERVAL MINOR2 ) && // dominant chord that
    GetType( 1 ) == CHORD_TYPE_DOMINANT7 ) // sub-resolves to it give
    l_iValue += 9; // it 9 points.
}
if( p_iEnd == 36 ) // Same as above, but for
// the bridge.

{
    if( GetRoot( 32 ) == g_iBKey &&
        GetType( 32 ) == CHORD_TYPE_MAJOR7 )
        l_iValue += 10;
    else if( GetRoot( 34 ) == g_iBKey &&
        GetType( 34 ) == CHORD_TYPE_DOMINANT7 &&
        GetRoot( 33 ) == g_dLookup.
        AddIntervalToPitch( g_iBKey,
        INTERVAL PERFECT5 ) &&
        GetType( 33 ) == CHORD_TYPE_DOMINANT7 )
        l_iValue += 9;
    else if( GetRoot( 34 ) == g_iBKey &&
        GetType( 34 ) == CHORD_TYPE_DOMINANT7 &&
        GetRoot( 33 ) == g_dLookup.
        AddIntervalToPitch( g_iBKey,
        INTERVAL MINOR2 ) &&
        GetType( 33 ) == CHORD_TYPE_DOMINANT7 )
        l_iValue += 9;
}

////////////////////////////////////
// Encourage sections to end with a cadence. //
////////////////////////////////////
if( p_iStart == 8 && p_iEnd == 20 // At the end of a phrase,
    && GetType( 15 ) == CHORD_TYPE_DOMINANT7 ) // there should be a
    l_iValue += 8; // cadence to the next
if( p_iStart == 24 && p_iEnd == 36 // phrase, which has not
    && GetType( 31 ) == CHORD_TYPE_DOMINANT7 ) // yet been generated.
    l_iValue += 8;
if( p_iStart == 40 && p_iEnd == 52
    && GetType( 47 ) == CHORD_TYPE_DOMINANT7 )
    l_iValue += 8;

////////////////////////////////////
// Encourage the end of the A section to morph properly. //
////////////////////////////////////
if( p_iEnd == 64 )
{
    if( GetType( 63 ) !=
        CHORD_TYPE_MAJOR7 ||
        GetRoot( 63 ) != g_iAKey )
        l_iValue -= 100;
    if( GetType( 62 ) !=
        CHORD_TYPE_MAJOR7 ||
        GetRoot( 62 ) != g_iAKey )
        l_iValue -= 50;
    if( GetType( 61 ) != CHORD_TYPE_DOMINANT7
        || ( GetRoot( 61 ) != g_dLookup.
            AddIntervalToPitch( g_iAKey,
            INTERVAL PERFECT5 ) && GetRoot( 61 ) !=
            g_dLookup.AddIntervalToPitch( g_iAKey,
            INTERVAL MAJOR2 ) ) )
        l_iValue -= 25;
}

return l_iValue * RandDist(); // Randomize the result

```



```

    }

CCHChordSearchTree::CCHChordSearchTree()
{
}

CCHChordSearchTree::~CCHChordSearchTree()
{
    EmptyQueue(); // Clean up memory used.
}

void CCHChordSearchTree::PrintBest(
    int p_iNum )
{
    cout << (LPCSTR)m_nBest.Print( ); // Print the results in the
                                        // best node.
}

void CCHChordSearchTree::EmptyQueue()
{
    while( m_lQueue.GetCount() ) // Remove everything in the
        delete m_lQueue.RemoveHead(); // queue.
    while( m_lFinished.GetCount() ) // Remove everything in the
        delete m_lFinished.RemoveHead(); // finished queue.
}

void CCHChordSearchTree::DepthFirstSearch()
{
    EmptyQueue(); // Make sure everything is
                  // setup empty for me.

    // Choose a key.
    // Generate a random number
    // between 0 and 99.
    // Determine the key from
    // percentages.

    int x = rand() % 100;

    if( x < 15 ) g_iAKey = PITCH_Bb;
    else if( x < 30 ) g_iAKey = PITCH_F;
    else if( x < 45 ) g_iAKey = PITCH_Eb;
    else if( x < 60 ) g_iAKey = PITCH_C;
    else if( x < 70 ) g_iAKey = PITCH_G;
    else if( x < 75 ) g_iAKey = PITCH_D;
    else if( x < 80 ) g_iAKey = PITCH_Ab;
    else if( x < 85 ) g_iAKey = PITCH_A;
    else if( x < 89 ) g_iAKey = PITCH_Db;
    else if( x < 93 ) g_iAKey = PITCH_E;
    else if( x < 97 ) g_iAKey = PITCH_Gb;
    else g_iAKey = PITCH_B;

    // Determine the starting
    // bridge key in relation.
    // Generate a random number
    // between 0 and 99.

    x = rand() % 100;
    if( x < 5 ) g_iBKey = g_iAKey;
    else if( x < 20 ) g_iBKey = g_dLookup.
        AddIntervalToPitch(
            g_iAKey, INTERVAL PERFECT5 );
    else if( x < 35 ) g_iBKey = g_dLookup.
        AddIntervalToPitch(
            g_iAKey, INTERVAL PERFECT4 );
    else if( x < 50 ) g_iBKey = g_dLookup.
        AddIntervalToPitch(
            g_iAKey, INTERVAL TRITONE );
    else if( x < 60 ) g_iBKey = g_dLookup.
        AddIntervalToPitch(
            g_iAKey, INTERVAL MAJOR2 );
    else if( x < 70 ) g_iBKey = g_dLookup.
        AddIntervalToPitch(
            g_iAKey, INTERVAL_MINOR7 );
}

```

```

else if( x < 77 ) g_iBKey = g_dLookup.
    AddIntervalToPitch(
        g_iAKey, INTERVAL MAJOR6 );
else if( x < 84 ) g_iBKey = g_dLookup.
    AddIntervalToPitch(
        g_iAKey, INTERVAL MINOR3 );
else if( x < 89 ) g_iBKey = g_dLookup.
    AddIntervalToPitch(
        g_iAKey, INTERVAL MAJOR3 );
else if( x < 94 ) g_iBKey = g_dLookup.
    AddIntervalToPitch(
        g_iAKey, INTERVAL MINOR6 );
else if( x < 97 ) g_iBKey = g_dLookup.
    AddIntervalToPitch(
        g_iAKey, INTERVAL MAJOR7 );
else g_iBKey = g_dLookup.
    AddIntervalToPitch(
        g_iAKey, INTERVAL MINOR2 );

cout << "\nKey of A section is: " << // Print out the key.
    (LPCTSTR)g_dLookup.GetPitchName( g_iAKey );
cout << "\nKey of B section is: " <<
    (LPCTSTR)g_dLookup.GetPitchName( g_iBKey )
    << "\n\n";

CCHChordSearchNode *l_nAdd = // Create a new node to add
    new CCHChordSearchNode; // to the queue.
m_lQueue.AddHead( l_nAdd ); // Add it to the queue.

CCHChordSearchNode *l_nCurrent = // Take the head off of the
    m_lQueue.RemoveHead(); // queue and store it in a
// current.
int iteration = 0; // A count of the times
// through the loop.

while( m_lQueue.GetCount() // This loop runs at least
    || iteration == 0 ) // once, and until its
// queue is empty.
{
    iteration++; // Increment the counter.

    int l_iCurrentDepth = // Get the node depth.
        l_nCurrent->GetCurrIndex();
    if( l_iCurrentDepth == -1 ) // If the node is full, put
// it on the finished
// queue.
    {
        m_lFinished.AddHead( l_nCurrent ); // Move this onto the queue
// of finished nodes.
        l_nCurrent = m_lQueue.RemoveHead(); // Get the next node to work
// with.
        if( !m_lQueue.GetCount() ) // If I removed the last
// element, it needs to be
// added to the queue,
// because this is the
// last loop iteration.
        {
            continue; // Skip the rest of this
// function.
        }
    }

    for( CHORD_TYPES i = CHORD_TYPE_FIRST; i <= // For every possible
        CHORD_TYPE_LAST; i = // combination of chord
        (CHORD_TYPES)( i + CHORD_TYPE_FIRST ) ) // types and roots we will
// be making a copy and
// inserting that chord at
// the next position.
    {
        for( PITCHES j = PITCH_FIRST;
            j <= PITCH_LAST;
            j = (PITCHES)( j + PITCH_FIRST ) )
        {
            l_nAdd = new CCHChordSearchNode; // Create a new node.
            l_nAdd->CopyFromOther( l_nCurrent ); // Copy the current into the

```

```

        l_nAdd->SetBoth( l_iCurrentDepth,
            j, i );
        m_lQueue.AddHead( l_nAdd );
        l_nAdd = NULL;

    }
    delete l_nCurrent;

    l_nCurrent = m_lQueue.RemoveHead();
}

while( m_lFinished.GetCount() )

    m_lQueue.AddHead(
        m_lFinished.RemoveHead() );
}

////////////////////////////////////
// The basic process is, for each successive 2 measures, to put
// the existing progression in l_nCurrent then concatenate each
// possible 4 chord progression to it.
////////////////////////////////////
void CCHChordSearchTree::ExpandNodes()
{
    CCHChordEvalNode *l_nAdd =
        new CCHChordEvalNode;
    CCHChordEvalNode *l_nCurrent =
        new CCHChordEvalNode;
    CCHChordSearchNode *l_nTemp;

    int l_iBest = -999999;

    int l_iTemp;

    int l_iBestCount = 0;

    //////////////////////////////////////
    // Choose one of the possible 2-bar chord progressions.
    //////////////////////////////////////

    while( m_lQueue.GetCount() )

        {
            l_nTemp = m_lQueue.RemoveHead();

            m_lFinished.AddHead( l_nTemp );

            l_nAdd->AppendShort( l_nTemp, 0 );

            l_iTemp = l_nAdd->Evaluate( 0, 4 );

            if( l_iTemp > l_iBest )

```

```

// the best for now.
{
    l_iBest = l_iTemp;
    m_nBest.CopyFromOther( *l_nAdd );
    l_iBestCount = 0;
}
else if( l_iTemp == l_iBest )
    l_iBestCount++;
}

////////////////////////////////////////
// Choose one of the possible 4-bar chord progressions that //
// begins with the chosen 2-bar chord progression.           //
////////////////////////////////////////
AddFour( 2 );

////////////////////////////////////////
// Choose one of the possible 6-bar chord progressions that //
// begins with the chosen 4-bar chord progression.           //
////////////////////////////////////////
AddFour( 3 );

////////////////////////////////////////
// Choose one of the possible 8-bar chord progressions that //
// begins with the chosen 6-bar chord progression.           //
////////////////////////////////////////
AddFour( 4 );

////////////////////////////////////////
// Copy the first 16 chords to the second 16 chords, AABA. //
////////////////////////////////////////

m_nBest.CopyFromOther( m_nBest, 16, 32, -16 );

////////////////////////////////////////
// Choose the chords in the bridge.                           //
////////////////////////////////////////

AddFour( 9 );
AddFour( 10 );
AddFour( 11 );
////////////////////////////////////////
// Copy the first 16 chords to the last 16 chords, AABA. //
////////////////////////////////////////

m_nBest.CopyFromOther( m_nBest, 48, 64, -48 );
////////////////////////////////////////
// Choose the last four chords in the bridge while looking //
// at what is in the last A section.                         //
////////////////////////////////////////

l_iBestCount = 0;
// The number of different
// progressions that
// shared the highest
// value.

l_nCurrent->CopyFromOther( m_nBest );
// The l_nCurrent node is
// the existing
// progression.
l_iBest = -999999;
// Reset the best value to a
// very small number.
while( m_lFinished.GetCount() )
// Take each 2-bar
// progression from the
// queue to use.
{
    l_nTemp = m_lFinished.RemoveHead();
    // Take the 2-bar
    // progression off of the

```



```

m_lQueue.AddHead( l_nTemp );
l_nAdd->CopyFromOther( *l_nCurrent );
l_nAdd->AppendShort( l_nTemp, 44 );

l_iTemp = l_nAdd->Evaluate( 40, 52 );
if( l_iTemp > l_iBest )
{
    l_iBest = l_iTemp;
    m_nBest.CopyFromOther( *l_nAdd );
}
}

//////////////////////////////////////
// Redo the last 4 bars to make it end nicely. //
//////////////////////////////////////

l_iBestCount = 0; // Etc.

l_nCurrent->CopyFromOther( m_nBest );
l_iBest = -999999;
while( m_lQueue.GetCount() )
{
    l_nTemp = m_lQueue.RemoveHead();
    m_lFinished.AddHead( l_nTemp );
    l_nAdd->CopyFromOther( *l_nCurrent );
    l_nAdd->AppendShort( l_nTemp, 60 );
    l_iTemp = l_nAdd->Evaluate( 56, 64 );
    if( l_iTemp > l_iBest )
    {
        l_iBest = l_iTemp;
        m_nBest.CopyFromOther( *l_nAdd );
    }
}

//////////////////////////////////////
// Redo the last 4 bars of the first A section to make it //
// lead smoothly into another. //
//////////////////////////////////////

l_iBestCount = 0; // Etc.

l_nCurrent->CopyFromOther( m_nBest );
l_iBest = -999999;
while( m_lFinished.GetCount() )
{
    l_nTemp = m_lFinished.RemoveHead();
    m_lQueue.AddHead( l_nTemp );
    l_nAdd->CopyFromOther( *l_nCurrent );
    l_nAdd->AppendShort( l_nTemp, 12 );
    l_iTemp = l_nAdd->Evaluate( 8, 20 );
    if( l_iTemp > l_iBest )
    {
        l_iBest = l_iTemp;
        m_nBest.CopyFromOther( *l_nAdd );
    }
}

//////////////////////////////////////
// Redo the last 4 bars of the second A section to make it //
// lead smoothly into the bridge. //
//////////////////////////////////////

l_iBestCount = 0; // Etc.

l_nCurrent->CopyFromOther( m_nBest );

```



```

l_iBest = -999999;
while( m_lQueue.GetCount() )
{
    l_nTemp = m_lQueue.RemoveHead();
    m_lFinished.AddHead( l_nTemp );
    l_nAdd->CopyFromOther( *l_nCurrent );
    l_nAdd->AppendShort( l_nTemp, 28 );
    l_iTemp = l_nAdd->Evaluate( 24, 36 );
    if( l_iTemp > l_iBest )
    {
        l_iBest = l_iTemp;
        m_nBest.CopyFromOther( *l_nAdd );
    }
}

delete l_nAdd;
delete l_nCurrent;

}

void CCHChordSearchTree::AddFour(
int p_iIndex )
{
    CCHChordEvalNode *l_nAdd =
        new CCHChordEvalNode;
    CCHChordEvalNode *l_nCurrent =
        new CCHChordEvalNode;
    CCHChordSearchNode *l_nTemp;

    int l_iBest = -999999;

    int l_iTemp;

    int l_iBestCount = 0;

    l_nCurrent->CopyFromOther( m_nBest );

    l_iBest = -999999;

    while( ( p_iIndex % 2 ? m_lQueue.GetCount() :
        m_lFinished.GetCount() ) )
    {
        l_nTemp = ( p_iIndex % 2 ?
            m_lQueue.RemoveHead() :
            m_lFinished.RemoveHead() );
        ( p_iIndex % 2 ?
            m_lFinished.AddHead( l_nTemp ) :
            m_lQueue.AddHead( l_nTemp ) );
        l_nAdd->AppendShort( l_nTemp,
            4 * ( p_iIndex - 1 ) );
        l_nAdd->CopyFromOther( *l_nCurrent, 0,
            4 * ( p_iIndex - 1 ), 0 );
        l_iTemp = l_nAdd->Evaluate(
            p_iIndex * 4 - 8,
            p_iIndex * 4 );

        if( l_iTemp > l_iBest )

```

```

// Delete the node we were
// adding to.
// Delete the base node.

```

```

// This is the node we are
// currently expanding.
// This is the root of the
// node we are currently
// expanding.
// This is a 4 chord
// progression to work
// with.

```

```

// The value of the best
// progression so far.
// It is initialized to a
// very small number so
// any real progression
// should be better.
// The value of the current
// progression in l_nAdd.
// The number of different
// progressions that
// shared the highest
// value.

```

```

// The l_nCurrent node is
// the existing
// progression.
// Reset the best value to a
// very small number.
// Take each 2-bar
// progression from the
// queue to use.

```

```

// Take the 2-bar
// progression off of the
// one queue
// Add the 2-bar progression
// onto the other queue.
// Copy the 2 bars from the
// queue into the next
// bars.
// Copy the previous bars.
// Calculate the value of
// this progression.

```

```

// If it is better than the
// best so far, set it as

```

```

    {
        l_iBest = l_iTemp;
        m_nBest.CopyFromOther( *l_nAdd );
    }
}

delete l_nAdd;
delete l_nCurrent;

}

CCHProgression* CCHChordSearchTree::GetBest()
{
    return &m_nBest;
}

```

```

// the new best.

// The best value so far is
// now the value of this.
// Copy this into the best
// node storage.

// Delete the node that is
// being tested.
// Delete the node that
// holds the shorter
// progression.

```

```

/*! \file CHCS491\CHChordsSearch\MelodyMarkovChain.cpp
* \brief Defines a class for generating a melody.
*
* Documented in the format of Doxygen.
* See also CCHMelody.
*/

```

```

#include "stdafx.h"
#include "melodymarkovchain.h"

```

```

CCHMelody::CCHMelody()
{

```

```

#ifdef NORMAL_MELODY

```

```

// This include alters the
// probabilities to create
// a more classic melody.

```

```

    int l_iChord = CHORD_TYPE MAJOR7 / CHORD_TYPE FIRST - 1;

```

```

    m_iProbs[ l_iChord ][ 0 ][ PITCH_C ] [ PITCH_C ] = 21;
    m_iProbs[ l_iChord ][ 0 ][ PITCH_C ] [ PITCH_Db ] = 0;
    m_iProbs[ l_iChord ][ 0 ][ PITCH_C ] [ PITCH_D ] = 21;
    m_iProbs[ l_iChord ][ 0 ][ PITCH_C ] [ PITCH_Eb ] = 0;
    m_iProbs[ l_iChord ][ 0 ][ PITCH_C ] [ PITCH_E ] = 8;
    m_iProbs[ l_iChord ][ 0 ][ PITCH_C ] [ PITCH_F ] = 3;
    m_iProbs[ l_iChord ][ 0 ][ PITCH_C ] [ PITCH_Gb ] = 0;
    m_iProbs[ l_iChord ][ 0 ][ PITCH_C ] [ PITCH_G ] = 10;
    m_iProbs[ l_iChord ][ 0 ][ PITCH_C ] [ PITCH_Ab ] = 0;
    m_iProbs[ l_iChord ][ 0 ][ PITCH_C ] [ PITCH_A ] = 2;
    m_iProbs[ l_iChord ][ 0 ][ PITCH_C ] [ PITCH_Bb ] = 0;
    m_iProbs[ l_iChord ][ 0 ][ PITCH_C ] [ PITCH_B ] = 20;
    m_iProbs[ l_iChord ][ 0 ][ PITCH_C ] [ PITCH_NONE ] = 15;

```

```

    m_iProbs[ l_iChord ][ 1 ][ PITCH_C ] [ PITCH_C ] = 20;
    m_iProbs[ l_iChord ][ 1 ][ PITCH_C ] [ PITCH_Db ] = 6;
    m_iProbs[ l_iChord ][ 1 ][ PITCH_C ] [ PITCH_D ] = 15;
    m_iProbs[ l_iChord ][ 1 ][ PITCH_C ] [ PITCH_Eb ] = 1;
    m_iProbs[ l_iChord ][ 1 ][ PITCH_C ] [ PITCH_E ] = 8;
    m_iProbs[ l_iChord ][ 1 ][ PITCH_C ] [ PITCH_F ] = 3;
    m_iProbs[ l_iChord ][ 1 ][ PITCH_C ] [ PITCH_Gb ] = 1;
    m_iProbs[ l_iChord ][ 1 ][ PITCH_C ] [ PITCH_G ] = 8;
    m_iProbs[ l_iChord ][ 1 ][ PITCH_C ] [ PITCH_Ab ] = 1;
    m_iProbs[ l_iChord ][ 1 ][ PITCH_C ] [ PITCH_A ] = 2;
    m_iProbs[ l_iChord ][ 1 ][ PITCH_C ] [ PITCH_Bb ] = 1;
    m_iProbs[ l_iChord ][ 1 ][ PITCH_C ] [ PITCH_B ] = 19;
    m_iProbs[ l_iChord ][ 1 ][ PITCH_C ] [ PITCH_NONE ] = 15;

```

```

    m_iProbs[ l_iChord ][ 0 ][ PITCH_Db ] [ PITCH_C ] = 30;
    m_iProbs[ l_iChord ][ 0 ][ PITCH_Db ] [ PITCH_Db ] = 1;
    m_iProbs[ l_iChord ][ 0 ][ PITCH_Db ] [ PITCH_D ] = 25;
    m_iProbs[ l_iChord ][ 0 ][ PITCH_Db ] [ PITCH_Eb ] = 1;
    m_iProbs[ l_iChord ][ 0 ][ PITCH_Db ] [ PITCH_E ] = 5;
    m_iProbs[ l_iChord ][ 0 ][ PITCH_Db ] [ PITCH_F ] = 1;
    m_iProbs[ l_iChord ][ 0 ][ PITCH_Db ] [ PITCH_Gb ] = 1;
    m_iProbs[ l_iChord ][ 0 ][ PITCH_Db ] [ PITCH_G ] = 5;
    m_iProbs[ l_iChord ][ 0 ][ PITCH_Db ] [ PITCH_Ab ] = 1;
    m_iProbs[ l_iChord ][ 0 ][ PITCH_Db ] [ PITCH_A ] = 1;
    m_iProbs[ l_iChord ][ 0 ][ PITCH_Db ] [ PITCH_Bb ] = 1;
    m_iProbs[ l_iChord ][ 0 ][ PITCH_Db ] [ PITCH_B ] = 13;
    m_iProbs[ l_iChord ][ 0 ][ PITCH_Db ] [ PITCH_NONE ] = 15;

```

```

    m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ] [ PITCH_C ] = 29;
    m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ] [ PITCH_Db ] = 5;
    m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ] [ PITCH_D ] = 25;
    m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ] [ PITCH_Eb ] = 1;
    m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ] [ PITCH_E ] = 5;
    m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ] [ PITCH_F ] = 1;
    m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ] [ PITCH_Gb ] = 1;
    m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ] [ PITCH_G ] = 5;
    m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ] [ PITCH_Ab ] = 1;
    m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ] [ PITCH_A ] = 1;
    m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ] [ PITCH_Bb ] = 1;
    m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ] [ PITCH_B ] = 10;
    m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ] [ PITCH_NONE ] = 15;

```



```

m_iProbs[ l_iChord ][ 0 ][ PITCH_NONE ][ PITCH_B ] = 9;
m_iProbs[ l_iChord ][ 0 ][ PITCH_NONE ][ PITCH_NONE ] = 50;

m_iProbs[ l_iChord ][ 1 ][ PITCH_NONE ][ PITCH_C ] = 9;
m_iProbs[ l_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Db ] = 5;
m_iProbs[ l_iChord ][ 1 ][ PITCH_NONE ][ PITCH_D ] = 8;
m_iProbs[ l_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Eb ] = 4;
m_iProbs[ l_iChord ][ 1 ][ PITCH_NONE ][ PITCH_E ] = 9;
m_iProbs[ l_iChord ][ 1 ][ PITCH_NONE ][ PITCH_F ] = 4;
m_iProbs[ l_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Gb ] = 5;
m_iProbs[ l_iChord ][ 1 ][ PITCH_NONE ][ PITCH_G ] = 9;
m_iProbs[ l_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Ab ] = 5;
m_iProbs[ l_iChord ][ 1 ][ PITCH_NONE ][ PITCH_A ] = 4;
m_iProbs[ l_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Bb ] = 5;
m_iProbs[ l_iChord ][ 1 ][ PITCH_NONE ][ PITCH_B ] = 8;
m_iProbs[ l_iChord ][ 1 ][ PITCH_NONE ][ PITCH_NONE ] = 25;

```

```
l_iChord = CHORD_TYPE_MINOR7 / CHORD_TYPE_FIRST - 1;
```

```

m_iProbs[ l_iChord ][ 0 ][ PITCH_C ][ PITCH_C ] = 21;
m_iProbs[ l_iChord ][ 0 ][ PITCH_C ][ PITCH_Db ] = 1;
m_iProbs[ l_iChord ][ 0 ][ PITCH_C ][ PITCH_D ] = 14;
m_iProbs[ l_iChord ][ 0 ][ PITCH_C ][ PITCH_Eb ] = 19;
m_iProbs[ l_iChord ][ 0 ][ PITCH_C ][ PITCH_E ] = 1;
m_iProbs[ l_iChord ][ 0 ][ PITCH_C ][ PITCH_F ] = 1;
m_iProbs[ l_iChord ][ 0 ][ PITCH_C ][ PITCH_Gb ] = 1;
m_iProbs[ l_iChord ][ 0 ][ PITCH_C ][ PITCH_G ] = 5;
m_iProbs[ l_iChord ][ 0 ][ PITCH_C ][ PITCH_Ab ] = 1;
m_iProbs[ l_iChord ][ 0 ][ PITCH_C ][ PITCH_A ] = 1;
m_iProbs[ l_iChord ][ 0 ][ PITCH_C ][ PITCH_Bb ] = 19;
m_iProbs[ l_iChord ][ 0 ][ PITCH_C ][ PITCH_B ] = 1;
m_iProbs[ l_iChord ][ 0 ][ PITCH_C ][ PITCH_NONE ] = 15;

```

```

m_iProbs[ l_iChord ][ 1 ][ PITCH_C ][ PITCH_C ] = 20;
m_iProbs[ l_iChord ][ 1 ][ PITCH_C ][ PITCH_Db ] = 5;
m_iProbs[ l_iChord ][ 1 ][ PITCH_C ][ PITCH_D ] = 10;
m_iProbs[ l_iChord ][ 1 ][ PITCH_C ][ PITCH_Eb ] = 20;
m_iProbs[ l_iChord ][ 1 ][ PITCH_C ][ PITCH_E ] = 1;
m_iProbs[ l_iChord ][ 1 ][ PITCH_C ][ PITCH_F ] = 1;
m_iProbs[ l_iChord ][ 1 ][ PITCH_C ][ PITCH_Gb ] = 1;
m_iProbs[ l_iChord ][ 1 ][ PITCH_C ][ PITCH_G ] = 5;
m_iProbs[ l_iChord ][ 1 ][ PITCH_C ][ PITCH_Ab ] = 1;
m_iProbs[ l_iChord ][ 1 ][ PITCH_C ][ PITCH_A ] = 1;
m_iProbs[ l_iChord ][ 1 ][ PITCH_C ][ PITCH_Bb ] = 15;
m_iProbs[ l_iChord ][ 1 ][ PITCH_C ][ PITCH_B ] = 5;
m_iProbs[ l_iChord ][ 1 ][ PITCH_C ][ PITCH_NONE ] = 15;

```

```

m_iProbs[ l_iChord ][ 0 ][ PITCH_Db ][ PITCH_C ] = 20;
m_iProbs[ l_iChord ][ 0 ][ PITCH_Db ][ PITCH_Db ] = 1;
m_iProbs[ l_iChord ][ 0 ][ PITCH_Db ][ PITCH_D ] = 15;
m_iProbs[ l_iChord ][ 0 ][ PITCH_Db ][ PITCH_Eb ] = 19;
m_iProbs[ l_iChord ][ 0 ][ PITCH_Db ][ PITCH_E ] = 1;
m_iProbs[ l_iChord ][ 0 ][ PITCH_Db ][ PITCH_F ] = 5;
m_iProbs[ l_iChord ][ 0 ][ PITCH_Db ][ PITCH_Gb ] = 1;
m_iProbs[ l_iChord ][ 0 ][ PITCH_Db ][ PITCH_G ] = 5;
m_iProbs[ l_iChord ][ 0 ][ PITCH_Db ][ PITCH_Ab ] = 1;
m_iProbs[ l_iChord ][ 0 ][ PITCH_Db ][ PITCH_A ] = 1;
m_iProbs[ l_iChord ][ 0 ][ PITCH_Db ][ PITCH_Bb ] = 15;
m_iProbs[ l_iChord ][ 0 ][ PITCH_Db ][ PITCH_B ] = 1;
m_iProbs[ l_iChord ][ 0 ][ PITCH_Db ][ PITCH_NONE ] = 15;

```

```

m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ][ PITCH_C ] = 19;
m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ][ PITCH_Db ] = 5;
m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ][ PITCH_D ] = 14;
m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ][ PITCH_Eb ] = 19;
m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ][ PITCH_E ] = 1;
m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ][ PITCH_F ] = 4;
m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ][ PITCH_Gb ] = 1;
m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ][ PITCH_G ] = 5;
m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ][ PITCH_Ab ] = 1;
m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ][ PITCH_A ] = 1;
m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ][ PITCH_Bb ] = 10;
m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ][ PITCH_B ] = 5;

```



```
m iProbs[ 1 iChord ][ 1 ][ PITCH Db ][ PITCH NONE ] = 15;
```

```
m_iProbs[ 1_iChord ][ 0 ][ PITCH_G ][ PITCH_C ] = 10;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_G ][ PITCH_Db ] = 1;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_G ][ PITCH_D ] = 5;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_G ][ PITCH_Eb ] = 10;
```



```

m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_Bb ] = 9;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_B ] = 1;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_NONE ] = 50;

m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_C ] = 10;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Db ] = 3;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_D ] = 6;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Eb ] = 10;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_E ] = 4;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_F ] = 6;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Gb ] = 4;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_G ] = 10;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Ab ] = 4;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_A ] = 5;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Bb ] = 10;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_B ] = 3;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_NONE ] = 25;

```

1_iChord = CHORD_TYPE_DOMINANT7 / CHORD_TYPE_FIRST - 1;

```

m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_C ] = 25;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_Db ] = 1;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_D ] = 15;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_Eb ] = 1;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_E ] = 15;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_F ] = 1;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_Gb ] = 1;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_G ] = 10;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_Ab ] = 1;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_A ] = 1;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_Bb ] = 18;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_B ] = 1;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_NONE ] = 10;

```

```

m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_C ] = 23;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_Db ] = 5;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_D ] = 15;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_Eb ] = 3;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_E ] = 10;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_F ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_Gb ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_G ] = 10;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_Ab ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_A ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_Bb ] = 15;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_B ] = 5;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_NONE ] = 10;

```

```

m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_C ] = 23;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_Db ] = 3;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_D ] = 23;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_Eb ] = 1;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_E ] = 10;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_F ] = 1;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_Gb ] = 1;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_G ] = 10;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_Ab ] = 1;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_A ] = 1;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_Bb ] = 15;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_B ] = 1;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_NONE ] = 10;

```

```

m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_C ] = 21;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_Db ] = 6;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_D ] = 20;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_Eb ] = 5;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_E ] = 10;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_F ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_Gb ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_G ] = 10;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_Ab ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_A ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_Bb ] = 10;

```



```

m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_A ] = 1;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_Bb ] = 9;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_B ] = 1;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_NONE ] = 50;

m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_C ] = 9;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Db ] = 4;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_D ] = 10;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Eb ] = 4;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_E ] = 9;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_F ] = 4;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Gb ] = 4;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_G ] = 9;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Ab ] = 4;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_A ] = 4;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Bb ] = 10;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_B ] = 4;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_NONE ] = 25;

```

```

#else

```

```

int 1_iChord = CHORD_TYPE MAJOR7 / CHORD_TYPE FIRST - 1;

m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_C ] = 31;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_Db ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_D ] = 21;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_Eb ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_E ] = 8;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_F ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_Gb ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_G ] = 5;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_Ab ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_A ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_Bb ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_B ] = 20;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_NONE ] = 15;

m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_C ] = 25;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_Db ] = 6;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_D ] = 15;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_Eb ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_E ] = 10;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_F ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_Gb ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_G ] = 8;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_Ab ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_A ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_Bb ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_B ] = 19;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_NONE ] = 15;

m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_C ] = 33;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_Db ] = 1;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_D ] = 28;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_Eb ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_E ] = 5;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_F ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_Gb ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_G ] = 5;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_Ab ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_A ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_Bb ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_B ] = 13;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_NONE ] = 15;

m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_C ] = 30;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_Db ] = 5;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_D ] = 28;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_Eb ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_E ] = 5;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_F ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_Gb ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_G ] = 5;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_Ab ] = 0;

```



```
m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ][ PITCH_A ] = 0;
m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ][ PITCH_Bb ] = 0;
m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ][ PITCH_B ] = 12;
m_iProbs[ l_iChord ][ 1 ][ PITCH_Db ][ PITCH_NONE ] = 15;
```



```
m_iProbs[ l_iChord ][ 1 ][ PITCH_A ][ PITCH_C ] = 8;
m_iProbs[ l_iChord ][ 1 ][ PITCH_A ][ PITCH_Db ] = 0;
m_iProbs[ l_iChord ][ 1 ][ PITCH_A ][ PITCH_D ] = 5;
m_iProbs[ l_iChord ][ 1 ][ PITCH_A ][ PITCH_Eb ] = 0;
```



```

m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_G ] = 5;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_Ab ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_A ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_Bb ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_B ] = 5;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_NONE ] = 75;

```

```

m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_C ] = 9;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Db ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_D ] = 8;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Eb ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_E ] = 9;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_F ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Gb ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_G ] = 9;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Ab ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_A ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Bb ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_B ] = 8;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_NONE ] = 50;

```

```

1_iChord = CHORD_TYPE_MINOR7 / CHORD_TYPE_FIRST - 1;

```

```

m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_C ] = 40;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_Db ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_D ] = 10;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_Eb ] = 15;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_E ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_F ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_Gb ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_G ] = 5;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_Ab ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_A ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_Bb ] = 15;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_B ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_NONE ] = 15;

```

```

m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_C ] = 25;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_Db ] = 10;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_D ] = 10;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_Eb ] = 15;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_E ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_F ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_Gb ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_G ] = 5;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_Ab ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_A ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_Bb ] = 15;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_B ] = 5;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_NONE ] = 15;

```

```

m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_C ] = 26;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_Db ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_D ] = 15;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_Eb ] = 19;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_E ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_F ] = 5;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_Gb ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_G ] = 5;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_Ab ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_A ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_Bb ] = 15;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_B ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_NONE ] = 15;

```

```

m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_C ] = 23;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_Db ] = 5;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_D ] = 23;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_Eb ] = 10;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_E ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_F ] = 4;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_Gb ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_G ] = 5;

```



```

m_iProbs[ 1_iChord ][ 1 ][ PITCH_A ][ PITCH_Eb ] = 6;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_A ][ PITCH_E ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_A ][ PITCH_F ] = 3;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_A ][ PITCH_Gb ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_A ][ PITCH_G ] = 22;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_A ][ PITCH_Ab ] = 8;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_A ][ PITCH_A ] = 14;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_A ][ PITCH_Bb ] = 24;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_A ][ PITCH_B ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_A ][ PITCH_NONE ] = 15;

```

```

m_iProbs[ 1_iChord ][ 0 ][ PITCH_Bb ][ PITCH_C ] = 18;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Bb ][ PITCH_Db ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Bb ][ PITCH_D ] = 1;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Bb ][ PITCH_Eb ] = 7;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Bb ][ PITCH_E ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Bb ][ PITCH_F ] = 4;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Bb ][ PITCH_Gb ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Bb ][ PITCH_G ] = 10;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Bb ][ PITCH_Ab ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Bb ][ PITCH_A ] = 5;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Bb ][ PITCH_Bb ] = 40;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Bb ][ PITCH_B ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Bb ][ PITCH_NONE ] = 15;

```

```

m_iProbs[ 1_iChord ][ 1 ][ PITCH_Bb ][ PITCH_C ] = 20;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Bb ][ PITCH_Db ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Bb ][ PITCH_D ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Bb ][ PITCH_Eb ] = 6;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Bb ][ PITCH_E ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Bb ][ PITCH_F ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Bb ][ PITCH_Gb ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Bb ][ PITCH_G ] = 10;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Bb ][ PITCH_Ab ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Bb ][ PITCH_A ] = 15;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Bb ][ PITCH_Bb ] = 25;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Bb ][ PITCH_B ] = 9;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Bb ][ PITCH_NONE ] = 15;

```

```

m_iProbs[ 1_iChord ][ 0 ][ PITCH_B ][ PITCH_C ] = 29;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_B ][ PITCH_Db ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_B ][ PITCH_D ] = 1;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_B ][ PITCH_Eb ] = 8;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_B ][ PITCH_E ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_B ][ PITCH_F ] = 5;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_B ][ PITCH_Gb ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_B ][ PITCH_G ] = 10;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_B ][ PITCH_Ab ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_B ][ PITCH_A ] = 1;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_B ][ PITCH_Bb ] = 30;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_B ][ PITCH_B ] = 1;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_B ][ PITCH_NONE ] = 15;

```

```

m_iProbs[ 1_iChord ][ 1 ][ PITCH_B ][ PITCH_C ] = 30;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_B ][ PITCH_Db ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_B ][ PITCH_D ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_B ][ PITCH_Eb ] = 7;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_B ][ PITCH_E ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_B ][ PITCH_F ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_B ][ PITCH_Gb ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_B ][ PITCH_G ] = 10;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_B ][ PITCH_Ab ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_B ][ PITCH_A ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_B ][ PITCH_Bb ] = 30;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_B ][ PITCH_B ] = 5;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_B ][ PITCH_NONE ] = 15;

```

```

m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_C ] = 6;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_Db ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_D ] = 1;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_Eb ] = 5;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_E ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_F ] = 3;

```



```

m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_Gb ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_G ] = 5;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_Ab ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_A ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_Bb ] = 5;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_B ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_NONE ] = 75;

```

```

m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_C ] = 8;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Db ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_D ] = 4;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Eb ] = 8;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_E ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_F ] = 4;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Gb ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_G ] = 8;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Ab ] = 2;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_A ] = 3;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Bb ] = 8;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_B ] = 2;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_NONE ] = 50;

```

```

1_iChord = CHORD_TYPE_DOMINANT7 / CHORD_TYPE_FIRST - 1;

```

```

m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_C ] = 40;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_Db ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_D ] = 15;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_Eb ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_E ] = 8;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_F ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_Gb ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_G ] = 8;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_Ab ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_A ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_Bb ] = 14;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_B ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_C ][ PITCH_NONE ] = 15;

```

```

m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_C ] = 25;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_Db ] = 5;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_D ] = 15;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_Eb ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_E ] = 9;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_F ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_Gb ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_G ] = 9;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_Ab ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_A ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_Bb ] = 15;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_B ] = 5;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_C ][ PITCH_NONE ] = 15;

```

```

m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_C ] = 31;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_Db ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_D ] = 31;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_Eb ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_E ] = 7;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_F ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_Gb ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_G ] = 7;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_Ab ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_A ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_Bb ] = 9;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_B ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_Db ][ PITCH_NONE ] = 15;

```

```

m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_C ] = 29;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_Db ] = 6;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_D ] = 29;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_Eb ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_E ] = 6;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_F ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_Db ][ PITCH_Gb ] = 0;

```

```
m_iProbs[ 1_iChord ][ 1 ][ PITCH Db ] [ PITCH G ] = 6;
m_iProbs[ 1_iChord ][ 1 ][ PITCH Db ] [ PITCH Ab ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH Db ] [ PITCH A ] = 0;
m_iProbs[ 1_iChord ][ 1 ][ PITCH Db ] [ PITCH Bb ] = 6;
m_iProbs[ 1_iChord ][ 1 ][ PITCH Db ] [ PITCH B ] = 2;
m_iProbs[ 1_iChord ][ 1 ][ PITCH Db ] [ PITCH NONE ] = 15;
```

```
m_iProbs[1 iChord][0][ PITCH D ][ PITCH C ] = 16;
m_iProbs[1 iChord][0][ PITCH D ][ PITCH Db ] = 0;
m_iProbs[1 iChord][0][ PITCH D ][ PITCH D ] = 40;
m_iProbs[1 iChord][0][ PITCH D ][ PITCH Eb ] = 0;
m_iProbs[1 iChord][0][ PITCH D ][ PITCH E ] = 17;
m_iProbs[1 iChord][0][ PITCH D ][ PITCH F ] = 0;
m_iProbs[1 iChord][0][ PITCH D ][ PITCH Gb ] = 0;
m_iProbs[1 iChord][0][ PITCH D ][ PITCH G ] = 6;
m_iProbs[1 iChord][0][ PITCH D ][ PITCH Ab ] = 0;
m_iProbs[1 iChord][0][ PITCH D ][ PITCH A ] = 0;
m_iProbs[1 iChord][0][ PITCH D ][ PITCH Bb ] = 6;
m_iProbs[1 iChord][0][ PITCH D ][ PITCH B ] = 0;
m_iProbs[1 iChord][0][ PITCH D ][ PITCH NONE ] = 15;
```

```
m_iProbs[1 iChord][1][ PITCH D ][ PITCH C ] = 18;
m_iProbs[1 iChord][1][ PITCH D ][ PITCH Db ] = 6;
m_iProbs[1 iChord][1][ PITCH D ][ PITCH D ] = 27;
m_iProbs[1 iChord][1][ PITCH D ][ PITCH Eb ] = 6;
m_iProbs[1 iChord][1][ PITCH D ][ PITCH E ] = 18;
m_iProbs[1 iChord][1][ PITCH D ][ PITCH F ] = 0;
m_iProbs[1 iChord][1][ PITCH D ][ PITCH Gb ] = 0;
m_iProbs[1 iChord][1][ PITCH D ][ PITCH G ] = 5;
m_iProbs[1 iChord][1][ PITCH D ][ PITCH Ab ] = 0;
m_iProbs[1 iChord][1][ PITCH D ][ PITCH A ] = 0;
m_iProbs[1 iChord][1][ PITCH D ][ PITCH Bb ] = 5;
m_iProbs[1 iChord][1][ PITCH D ][ PITCH B ] = 0;
m_iProbs[1 iChord][1][ PITCH D ][ PITCH NONE ] = 15;
```

```
m iProbs[1 iChord][0][PITCH Eb][PITCH C] = 12;
m iProbs[1 iChord][0][PITCH Eb][PITCH Db] = 0;
m iProbs[1 iChord][0][PITCH Eb][PITCH D] = 29;
m iProbs[1 iChord][0][PITCH Eb][PITCH Eb] = 0;
m iProbs[1 iChord][0][PITCH Eb][PITCH E] = 30;
m iProbs[1 iChord][0][PITCH Eb][PITCH F] = 0;
m iProbs[1 iChord][0][PITCH Eb][PITCH Gb] = 0;
m iProbs[1 iChord][0][PITCH Eb][PITCH G] = 7;
m iProbs[1 iChord][0][PITCH Eb][PITCH Ab] = 0;
m iProbs[1 iChord][0][PITCH Eb][PITCH A] = 0;
m iProbs[1 iChord][0][PITCH Eb][PITCH Bb] = 7;
m iProbs[1 iChord][0][PITCH Eb][PITCH B] = 0;
m iProbs[1 iChord][0][PITCH Eb][PITCH NONE] = 15;
```

```
m_iProbs[1_iChord][1][PITCH Eb][PITCH C] = 7;
m_iProbs[1_iChord][1][PITCH Eb][PITCH Db] = 0;
m_iProbs[1_iChord][1][PITCH Eb][PITCH D] = 31;
m_iProbs[1_iChord][1][PITCH Eb][PITCH Eb] = 3;
m_iProbs[1_iChord][1][PITCH Eb][PITCH E] = 31;
m_iProbs[1_iChord][1][PITCH Eb][PITCH F] = 3;
m_iProbs[1_iChord][1][PITCH Eb][PITCH Gb] = 0;
m_iProbs[1_iChord][1][PITCH Eb][PITCH G] = 5;
m_iProbs[1_iChord][1][PITCH Eb][PITCH Ab] = 0;
m_iProbs[1_iChord][1][PITCH Eb][PITCH A] = 0;
m_iProbs[1_iChord][1][PITCH Eb][PITCH Bb] = 5;
m_iProbs[1_iChord][1][PITCH Eb][PITCH B] = 0;
m_iProbs[1_iChord][1][PITCH Eb][PITCH NONE] = 15;
```

```
m_iProbs[1_iChord][0][PITCH_E][PITCH_C] = 10;
m_iProbs[1_iChord][0][PITCH_E][PITCH_Db] = 0;
m_iProbs[1_iChord][0][PITCH_E][PITCH_D] = 14;
m_iProbs[1_iChord][0][PITCH_E][PITCH_Eb] = 40;
m_iProbs[1_iChord][0][PITCH_E][PITCH_E] = 0;
m_iProbs[1_iChord][0][PITCH_E][PITCH_F] = 1;
m_iProbs[1_iChord][0][PITCH_E][PITCH_Gb] = 0;
m_iProbs[1_iChord][0][PITCH_E][PITCH_G] = 15;
m_iProbs[1_iChord][0][PITCH_E][PITCH_Ab] = 0;
m_iProbs[1_iChord][0][PITCH_E][PITCH_A] = 0;
```



```
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_C ] = 5;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_Db ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_D ] = 5;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_Eb ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_E ] = 5;
```

```

m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_F ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_Gb ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_G ] = 5;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_Ab ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_A ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_Bb ] = 5;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_B ] = 0;
m_iProbs[ 1_iChord ][ 0 ][ PITCH_NONE ][ PITCH_NONE ] = 75;

m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_C ] = 9;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Db ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_D ] = 8;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Eb ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_E ] = 9;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_F ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Gb ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_G ] = 9;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Ab ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_A ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_Bb ] = 8;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_B ] = 1;
m_iProbs[ 1_iChord ][ 1 ][ PITCH_NONE ][ PITCH_NONE ] = 50;

#endif
}

void CCHMelody::SetNote(
    int p_iPosition,
    PITCHES p_eValue )
{
    if( p_iPosition >= 0 && p_iPosition < 256 ) // Set the appropriate value
        m_iNotes[ p_iPosition ] = p_eValue; // if position is in
                                              // the valid range.
}

PITCHES CCHMelody::GetNote(
    int p_iPosition )
{
    if( p_iPosition >= 0 && p_iPosition < 256 ) // Get the appropriate value
        return m_iNotes[ p_iPosition ]; // if position is in
    return PITCH_NONE; // the valid range.
}

void CCHMelody::GenerateChain()
{
    CHORD_TYPES l_eChord; // The current chord type.
    PITCHES l_eRealRoot; // The actual chord root.
    PITCHES l_eAdjustedRoot; // The root adjusted for a
    // C-based chord. Namely,
    // the note C.
    PITCHES l_eRealPrev; // The actual previous note.
    PITCHES l_eAdjustedPrev; // The previous note
    // adjusted for the chord.
    // For example, if the
    // note was G and the
    // chord was an Em7, the
    // adjusted note would be
    // to a Cm7 as G is to
    // Em7. Thus it is the
    // minor third of the
    // chord, or Eb.
    PITCHES l_eRealThis; // The actual new note.
    PITCHES l_eAdjustedThis; // The new note adjusted for
    // a chord with root C.

    l_eChord = m_pChords->GetType( 0 ); // Get the first chord type.
    l_eRealRoot = m_pChords->GetRoot( 0 ); // Get the first chord root.
    l_eAdjustedRoot = PITCH_C; // The adjusted root is C.
    l_eRealPrev = PITCH_NONE; // There is no previous.
    l_eAdjustedPrev = PITCH_NONE;
    l_eAdjustedThis = Choose( // Choose the first note.
        l_eChord,

```

```

0,
1_eAdjustedPrev );
1_eRealThis = m_cLookup.AddIntervalToPitch( // Convert to the note in
1_eAdjustedThis, // the actual chord.
m_cLookup.GetInterval(
1_eAdjustedRoot,
1_eRealRoot ) );
m_iNotes[ 0 ] = 1_eRealThis; // Set the note.
m_iNotes[ 64 ] = 1_eRealThis; // Copy for second and third
m_iNotes[ 192 ] = 1_eRealThis; // A sections.

int i;
for( i = 1; i < 48; i++ ) // Fill the first 6 bars.
{
1_eChord = m_pChords->GetType( i / 4 ); // This is essentially the
1_eRealRoot = m_pChords->GetRoot( i / 4 ); // same process as above.
1_eRealPrev = m_iNotes[ i - 1 ];
1_eAdjustedPrev = m_cLookup.
AddIntervalToPitch(
1_eRealPrev,
m_cLookup.GetInterval(
1_eRealRoot,
1_eAdjustedRoot ) );
1_eAdjustedThis = Choose(
1_eChord,
i % 2,
1_eAdjustedPrev );
if( 1_eAdjustedThis != PITCH_NONE )
1_eRealThis = m_cLookup.
AddIntervalToPitch(
1_eAdjustedThis,
m_cLookup.GetInterval(
1_eAdjustedRoot,
1_eRealRoot ) );
else
1_eRealThis = 1_eAdjustedThis;
m_iNotes[ i ] = 1_eRealThis;
m_iNotes[ i + 64 ] = 1_eRealThis;
m_iNotes[ i + 192 ] = 1_eRealThis;
}

for( i = 48; i < 64; i++ ) // Bars 6-7 are unique.
{
1_eChord = m_pChords->GetType( i / 4 ); // This is essentially the
1_eRealRoot = m_pChords->GetRoot( i / 4 ); // same process as above.
1_eRealPrev = m_iNotes[ i - 1 ];
1_eAdjustedPrev = m_cLookup.
AddIntervalToPitch(
1_eRealPrev,
m_cLookup.GetInterval(
1_eRealRoot,
1_eAdjustedRoot ) );
1_eAdjustedThis = Choose(
1_eChord,
i % 2,
1_eAdjustedPrev );
if( 1_eAdjustedThis != PITCH_NONE )
1_eRealThis = m_cLookup.
AddIntervalToPitch(
1_eAdjustedThis,
m_cLookup.GetInterval(
1_eAdjustedRoot,
1_eRealRoot ) );
else
1_eRealThis = 1_eAdjustedThis;
m_iNotes[ i ] = 1_eRealThis;
}

for( i = 112; i < 192; i++ ) // Bars 14-23 are unique.
{
1_eChord = m_pChords->GetType( i / 4 ); // This is essentially the
1_eRealRoot = m_pChords->GetRoot( i / 4 ); // same process as above.
1_eRealPrev = m_iNotes[ i - 1 ];

```

```

1_eAdjustedPrev = m_cLookup.
AddIntervalToPitch(
1_eRealPrev,
m_cLookup.GetInterval(
1_eRealRoot,
1_eAdjustedRoot ) );
1_eAdjustedThis = Choose(
1_eChord,
i % 2,
1_eAdjustedPrev );
if( 1_eAdjustedThis != PITCH_NONE )
1_eRealThis = m_cLookup.
AddIntervalToPitch(
1_eAdjustedThis,
m_cLookup.GetInterval(
1_eAdjustedRoot,
1_eRealRoot ) );
else
1_eRealThis = 1_eAdjustedThis;
m_iNotes[ i ] = 1_eRealThis;
}

for( i = 240; i < 256; i++ ) // Bars 31-32 are unique.
{
1_eChord = m_pChords->GetType( i / 4 ); // This is essentially the
1_eRealRoot = m_pChords->GetRoot( i / 4 ); // same process as above.
1_eRealPrev = m_iNotes[ i - 1 ];
1_eAdjustedPrev = m_cLookup.
AddIntervalToPitch(
1_eRealPrev,
m_cLookup.GetInterval(
1_eRealRoot,
1_eAdjustedRoot ) );
1_eAdjustedThis = Choose(
1_eChord,
i % 2,
1_eAdjustedPrev );
if( 1_eAdjustedThis != PITCH_NONE )
1_eRealThis = m_cLookup.
AddIntervalToPitch(
1_eAdjustedThis,
m_cLookup.GetInterval(
1_eAdjustedRoot,
1_eRealRoot ) );
else
1_eRealThis = 1_eAdjustedThis;
m_iNotes[ i ] = 1_eRealThis;
}

}

PITCHES CCHMelody::Choose(
CHORD_TYPES p_eChord,
short p_iPos,
PITCHES p_ePrev )
{
int l_iProb = rand() % 100; // Generate a random number
// in the range 0-99.
int l_iTotal = 0; // The probability of an
// event occurring.
for( PITCHES l_eNew = PITCH_NONE; // Check each potential pitc
h.
1_eNew <= PITCH_LAST;
1_eNew = (PITCHES)( 1_eNew + 1 ) )
{
1_iTotal += m_iProbs // Add the probability of
[ p_eChord / CHORD_TYPE_FIRST - 1 ] // this pitch to the
[ p_iPos ] // total.
[ p_ePrev ]
[ l_eNew ];
if( 1_iProb < 1_iTotal )
return l_eNew; // Get the pitch where the
// total crosses this

```



```

    }
    ASSERT( FALSE );

    return PITCH_NONE;
}

CString CCHMelody::Print()
{
    CString l_sRet;
    for( int i = 0; i < 256; i++ )
    {
        l_sRet += m_cLookup.GetPitchName(
            m_iNotes[ i ] );
        if( i % 8 == 7 )
            l_sRet += " | ";
        else
            l_sRet += " ";
        if( i % 16 == 15 )
            l_sRet += "\r\n";
    }
    return l_sRet;
}

// threshold.
// There should be some
// note, or the matrix
// of probabilities is
// corrupt.

// The string to print to.
// Use each note in the
// sequence.

// Write the pitch name.

// Write a barline every
// eight notes.

// Write two bars on a line.

// Return the string.

```

```

/! \file CHCS491\CHChordsSearch\CHChordsSearch.cpp
* \brief A simple driver program for the classes.
*
* Documented in the format of Doxygen.
*/

#include "stdafx.h"
#include "CHChordsSearch.h"
#include "ChordSearchTree.h"
#include "Progression.h"
#include "MelodyMarkovChain.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

//////////////////////////////////////
// The one and only application object

CWinApp theApp;

using namespace std;

int _tmain(int argc, TCHAR* argv[], TCHAR* envp[])
{
    int nRetCode = 0;

    // initialize MFC and print and error on failure
    if (!AfxWinInit(::GetModuleHandle(NULL), NULL, ::GetCommandLine(), 0))
    {
        // TODO: change error code to suit your needs
        cerr << _T("Fatal Error: MFC initialization failed") << endl;
        nRetCode = 1;
    }
    else
    {
        // TODO: code your application's behavior here.
        CString strHello;
        strHello.LoadString(IDS_HELLO);
        DWORD t1 = GetTickCount(); // Get the initial time.
        srand(time(0));
        {
            CCHChordSearchTree l_tTree;
            l_tTree.DepthFirstSearch();
            l_tTree.ExpandNodes();
            l_tTree.PrintBest();

            CCHMelody l_cMelody;
            l_cMelody.m_pChords = l_tTree.GetBest();
            l_cMelody.GenerateChain();
            cout << "\r\n\r\n" << (LPCSTR)l_cMelody.Print();
        }
        t1 = GetTickCount() - t1;
        cout << "\n\nTime to finish: "
             << t1 / 1000
             << " seconds or: "
             << t1 / 60000
             << " minutes.";
        /* cout << "\n\nWaiting for input to exit:";
           int asdf;
           cin >> asdf;*/
    }

    return nRetCode;
}

```