Boost.Unicode

a Unicode string library for C++

Foreword

Boost.Unicode is a graduation project for the bachelor studies in computer science at Gjøvik University College.

The project has resulted in a Unicode string library for C++ that abstracts away the complexity of working with Unicode text.

The idea behind the project originated from the Boost community's developer mailing lists, and is developed with inclusion into the Boost library collection in mind.

The developers would like to thank the entire Boost community for giving us a chance to have a go at this project, and for all their help throughout the development process. Special thanks go out to Miro Jurisic´ and Rogier van Dalen who have been especially involved in the discussions around the design.

Gjøvik May 19, 2005

Erik Wien	Lars Olav Gigstad

Table of contents

1 Introduction

1.1 The project

1.1.1 Text representation on computers

If you want to use a computer to represent some kind of text, you have to teach the computer how it should do just that. Basically you need to "translate" the textual information into a form the computer can easily understand and process. This is done by giving each of the characters in the alphabet a number, and storing sequences of these numbers in the computer's memory. To be able to make this work, you do of course have to agree on one specific number for every character, and make that numbering-scheme universal. That way, every computer that understands this numbering-scheme, can interpret and display text from any other computer.

Traditionally this has been done through the ASCII encoding. ASCII (more specifically extended ASCII) uses 8 bits to represent one character, and some basic binary math tells us that ASCII thus is capable of representing 256 unique characters. This is more than enough for representing the Latin alphabet used in most parts of the western world, as well as numbers and some other handy characters. The problem is, there's more to the world that the western part of it. (Though many people seem to be intentionally ignorant of that fact.) In Asian countries for example, there are many different scripts, several of which have thousands of unique characters. Obviously this makes it impossible for ASCII to handle all the different characters that exist around the world, and therefore Asian computer users (amongst others) have been forced to make their own encoding-schemes to be able to represent their native languages on a computer. (Shift-JIS in one example of such an encoding) This is of course a logical solution and works quite well, but it raises one problem. It is

no longer possible to simply transfer the encoded text from one computer to the other. You need to know what encoding is used to be able to make sense of the data. This would be simple enough if there was only a few different encodings out there, but unfortunately there aren't. There are several hundred of them, and keeping track of which one is used in a given piece of encoded text can be a real nightmare. Enter Unicode.

1.1.2 The Unicode Standard

The Unicode Consortium has taken upon itself the monstrous task of once and for all defining one unique number for representing each and every character known to man. No more encoding nightmare. One encoding fits all. The Consortium's work has resulted in a standard (The Unicode Standard) that currently (version 4.1.0) sports a healthy 90,000+ characters, from all kinds of different scripts like Latin, Hebrew, Arabic, Katakana, Hangul and even the old Viking's Runic script. And the standard keeps on growing.

Great! So we have a universal way of representing text on computers that can be used throughout the world. That solves all problems associated with character encoding, doesn't it? Well.. not really. The thing is, all the different encodings that were created prior to Unicode are still in heavy use. This is basically due to one simple fact. Many of the programming tools used by programmers were created before the Unicode standard was, meaning they don't have built-in support for Unicode. This means the programmers have to manually implement support for Unicode in their applications. Writing a Unicode aware application is therefore a lot more complex than it has to be, very often so complex the programmers don't bother spending the time required to do it. (Yes, we're lazy!) One of the most popular programming languages in the world, C++, is unfortunately one of the aforementioned programming tools, a fact that has hindered the adaptation of Unicode quite a bit. Now, this is where we get to the point of all this talk about text and encodings. By developing an extension to the C++ language that abstracts away the complexity of handling Unicode strings, we can make it much easier for programmers to make their applications Unicode aware, and thus enabling them to release their application all over the world, without even having to recompile them. Wow, developing something like that sounds like a good idea for a final project doesn't it?

1.1.3 Motivation

“But”, I hear you say, “There are already a few Unicode libraries available for C++. Why do I need another one?”. Well, it's a good thing you asked. The thing is that the existing libraries all seem have some sort of problem attached to them.

The biggest problem, one that applies to most libraries, is that they're commercial. They cost you money, to put it bluntly. In addition to being lazy, programmers are also cheap, so that won't fly for the vast majority of us. There are a few libraries out there that are free of charge, but unfortunately most of them are plagued by a just as serious problem. They just don't follow many of the conventions with respect to design and coding style established in the C++ standard library, and instead rely on their own way of getting things done. The most notable way this becomes apparent is that very few libraries provide STL conforming iterators and containers, making it impossible to use C++' standard algorithms together with them.

Another big problem with the other libraries, probably the biggest one, is that they're not standard. This means that if you decide to use one of them, your application suddenly becomes much less portable. The library has to be available for any platform (or even compiler) you might want to port to and that is not always the case.

What this all boils down to is that there is a real need for a new, and standardized Unicode library. One that would work on any compiler on any platform, as well as playing nice with the existing components in the C++ standard. To do this however, we need some kind of standardizing organization. We need Boost.

1.1.4 Boost

Boost is a community working on developing different libraries to complement C++' existing standard library. It was started by members of "The C++ Standards Committee Library Working Group" as a means for establishing "existing practice" and developing/testing libraries that can later be included in the official C++ standard. Acceptance of a library into Boost can therefore be viewed as the next best thing to acceptance into the C++ standard itself.

Today Boost consists of in the region of 70 different libraries, assisting programmers with everything from multi threaded programming, to graphs and template meta-programming. Basically libraries helping programmers with every problem under the sun, except adding Unicode support to their applications. That is exactly what Boost.Unicode is meant to remedy.

1.2 The Developers

1.2.1 Background

The developers behind this project are Erik Wien and Lars Olav Gigstad. Both are at their final year of Gjøvik Univerity College's (GUC) computer science bachelor studies, and have chosen to specialize their degrees in programming.

Both developers have a fair amount of experience in C++ (including STL design and implementation techniques), and have previously worked together on a couple of projects in the course Compilers at GUC.

1.2.2 What must be learned

Erik does already have a little bit experience with Unicode, but both of the developers need to read up on a lot of Unicode documentation to get a better understanding of the internals of it.

The developers also need to read up on the STL documentation to ensure the workings of the STL are fully understood.

Furthermore the developers need to investigate how to build up a code-tree that can easily be merged with, and built alongside the existing Boost libraries. This means they have to learn how to configure and use the Boost.Build system, among other things.

1.3 Other participants

Some other people are also involved, in one way or another, in this project.

Firstly the Boost community will help in the specification and design faces on the project, as well as helping with design evaluation along the way. They are ultimately the users of the library, so their input is vital to ensure the library turns out well.

Second, Frode Haug of GUC will work as the project's supervisor. This means that he will look over the development process throughout the project period, and give guidance if the developers get stuck on something.

1.4 Development model

For this project we have chosen to go with an evolutionary development model. There are several reasons for this choice. Firstly, it allows us to rapidly develop a preliminary prototype of the library and get it out to the community for comments and getting tips on possible improvements. Evolutionary development also makes it easier to actually do something about the comments the community may have, as the model encourages changes to the requirements and design as the project goes forward. (Something that is very likely to happen in this case!)

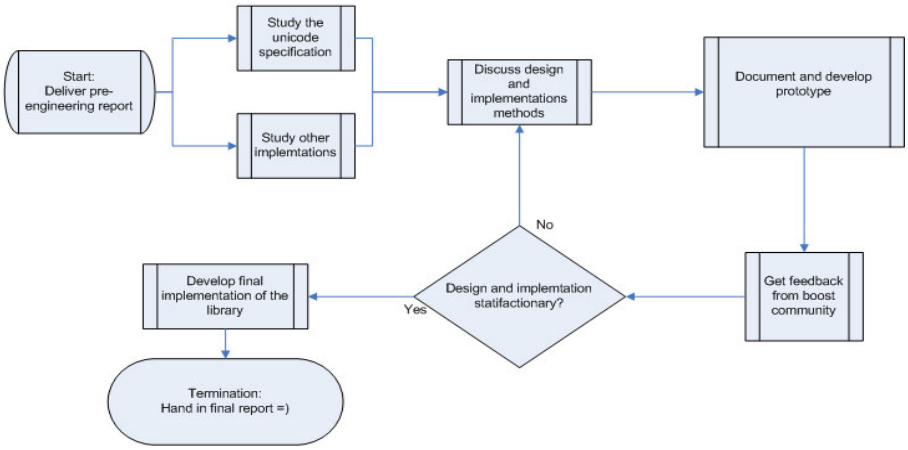

Figure 1 - Workflow diagram for the development process

The thing really that tipped the scale for us was the ability to develop using prototyping. This is a very good way of developing a library like this, as it allows us to actually try different design ideas to find the best one, instead of relying on completely designing the library beforehand and getting locked into that design. We know by experience that the design ideas you have at the beginning of such a project rarely are the best ones. We assume this to be true this time around as well.

In short

The development period will be split into 4 separate iterations. (With fixed start and end dates) Each of these iterations will have its own planning, specification and design period where we determine what should be done this iteration. Towards the end of each iteration, when all development is completed, we will have an evaluation where we discuss the results of the iteration and determine what (if anything) needs revision. All documentation (including specification and design

documents) will evolve along with the code itself, and it will therefore always represent the current state of the code.

1.5 This report

1.5.1 Target audience

Because of the rather technical nature of the implementation details of this library, and the fact that the end users are programmers themselves, this report has been written under the assumption that the reader have at least *some* knowledge of computer programming in general, and C++ and the STL in particular. A basic understanding of language constructs like templates, will be assumed, as having to go into detail on all such topics would leave the report looking like a C++ tutorial. Other readers can still follow large portions of the report, but they might have a hard time understanding some of the implementation details described.

1.5.2 Organization

This report is organized into 8 separate sections.

The first section, the one you are reading right now, concentrates on introducing the project and developers, as well as providing some other “introductory information”.

The second section is dedicated to a discussion around the topics this project relates to. This includes an in depth study of Unicode, as well as some information about the Standard Template Library. The section is rather important to read and understand to be able to make sense of the rest of the report.

The third section is the specification of Boost.Unicode. The section is more or less a snapshot of the document that has evolved through the entire development process, as more and more components were added to the library.

Fourth is the Design section. It contains the design document as it looked at the end of the project period. In it you will find information about the different components that make up this library.

The fifth section is the Implementation section. It provides some information about a couple of the implementation details of this

library. The aim of this section is to give a little insight into the depths of the source code, and how it all works.

Sixth is the testing section. It provides a some information about how the testing process was carried out.

Seventh is the section about the development process as a whole. It provides information about how we have worked, as well as our feelings about the project.

The eight and final section is the conclusion. This basically summarizes the entire project and our results in a couple of sentences.

For a detailed summary of the contents of the report, please take a look at the Table of Contents.

1.5.3 Layout

The layout of this report is based on the guidelines presented in the paper «Report templates. A guide to the writing and design of reports and R&D papers for students and staff at Gjøvik University College» - 2004 Nordström et.al. These guidelines are followed for anything from fonts and line-spacing, to placement of illustrations and other layout specific issues.

2 Analysis and theory

2.1 Introduction

Through the course of this project, we have spent a lot of time reading the Unicode and C++ standards to figure out how to best design an implement a Unicode library for C++. The knowledge we acquired during this research, has strongly influenced the final design and implementation of Boost.Unicode. In this section we will try to give a little overview of what we have learned through this process and we hope this will help you as the reader understand what on earth we are talking about in the Specification, Design and Implementation sections of this report.

2.2 Unicode in detail

2.2.1 Every character is a number

As mentioned in the introduction Unicode is a universal character encoding scheme that can represent just about every character known to man. This is accomplished by assigning one unique number, known as a code

Some Unicode code points:

Å U+00C5: LATIN CAPITAL LETTER A WITH RING ABOVE

딱 U+B564: HANGUL SYLLABLE SSANGTIKEUT AE KHIEUKH

point, to each character[1]. All of these code points are defined in the standard with a unique name, as well a lot of other data related to the code point in question. (More on this later in the report) It is important to note however, that the Unicode standard does not define how these characters look on your computer screen or on paper. This is outside the scope of Unicode and not defined in the standard.

2.2.2 Unicode encoding forms

As you have probably figured out "all characters known to man" add up to an awful lot of characters. Around one million code points (10FFFFh) are reserved in the current Unicode standard, even though only about 90.000 are in use at the moment. Still, compliant implementations must be able to represent every last one of these 10FFFFh code points. This means that if we are to represent a Unicode code point on a modern computer (Some x86 platform for instance), we will have to use a 32bit datatype(four bytes) to be able to represent a code point. (Strictly speaking, 24bit is sufficient, but because of alignment issues on 32bit processors, performance would be extremely poor if 24bit variables were used.) This is of course in stark contrast to ASCII that only requires one single byte (8bit) pr. "code point". Those extra 3 bytes quickly begin to add up when representing long sequences of characters. (Like this report for instance) To combat this size problem, the Unicode standard defines what is known as the Unicode encoding forms.

Unicode encoding forms:

`U+8A9E: CJK UNIFIED IDEOGRAPH`

Encoding	Bit-pattern
UTF-8	`E8 AA 9E`
UTF-16	`8A9E`
UTF-32	`00008A9E`

These encoding forms (UTF-8 UTF-16 and UTF32) each define a way of representing Unicode code points as a sequence of one or more 8, 16 or 32 bit *code units* respectively (Note the difference from code point here). These encoding forms are defined such that a single code point at most take up 32bits of memory, but most code points can be represented with far less. In UTF-8 for instance, the entire ASCII set

[1] http://www.unicode.org/charts/ has a complete list of the code points defined in the current Unicode standard.

of code points are represented using only one byte, resulting in huge memory (or disk space) savings for western users.

For more information about the Unicode encodings, and how exactly they work, you can have a look at the Unicode standard, section 2.5.

2.2.3 "Character" versus "code point"

Okay then. Unicode still doesn't sound too bad does it? Create a sequence of code points in an appropriate encoding form, and you can use it just like you would any other string in C++, right? Well, no. There's more to it than that. What people normally think of as a "character" is not the same thing as a Unicode code point, and therefore assuming that it is can lead to some interesting problems.

You see, code points can be much more than just some character. An interesting example of this, is something known as combining characters. These "characters" are code points just like any other, but they have a special meaning. When they are encountered inside a string they should be combined with the code point directly preceding it and together these code points have a unique meaning. One example of such a code point is `U+0301: COMBINING ACUTE ACCENT`. When combined with the character code point for 'e' for instance, this code point will result in the visual representation 'é'. Now, anyone who knows French knows that an 'e' is not the same thing as an 'é', but if you treated such a sequence purely as a series of code points and searched for the code point 'e' in that string, you *would* find a match, even though there isn't actually an 'e' in there.

Obviously this is no way to go. We need to have a way of separating the instances of 'e' and 'é' as two different characters to be able to have a reliable way of searching through Unicode text. Luckily, the creators of Unicode have thought of this already, and that leads us to the concept of text elements.

2.2.4 Text elements

A text element is a level of abstraction above "code points". A text element can consist of many code points, and together these code points have a certain unique meaning.

Grapheme clusters

One example of something that classifies as text-elements are grapheme clusters, the concept in the Unicode standard that is most similar to what people think of as a character. Put bluntly, a grapheme

cluster is a combination of some code point and any combining character that might follow it. This effectively solves the problem of the ‘e’ and ‘é’ we talked about earlier. They are now completely separate text elements, and you simply cannot confuse the two.

The Unicode standard defines what constitutes a grapheme cluster through a code point property known as “boundary property value”. This property defines what kind of code point the code point in question is. (A combining character will for instance have a boundary property value of `Extend`) The standard then defines a set of rules[2] that dictate which property values you are suppose to break between when reading grapheme clusters, and you can by employing these rules figure out which code points belong together in a cluster. (When moving through a code point sequence, you check the current and the next code point’s boundary property, check them against the rules, and if you are not supposed to break between them, the following code point is part of the same grapheme cluster as the current one.)

Words

Words are another type of text-elements defined in the standard. (Yes, words like in the normal sense of the term) These work in the exact same way as grapheme clusters (boundary properties and break rules), but instead of splitting the text into grapheme clusters they split it into words. This can be very useful when users are searching for some given word inside a string, and don’t want to find partial matches. (A search for “code” won’t return a match if you search in the string “Unicode”)

There is also a third type of text-elements, namely sentences, but we won’t waste your time describing them here. (It’s all in the standard if you’re interested.)

2.2.5 Normalization

Okay, but that’s it right? Surely there won’t be any problems if we represent our text as sequences of grapheme clusters? Well. Wrong again. There is one last problem that can still occur, and it has to do with (among other things) grapheme clusters with multiple combining characters. For instance, if you have the code point sequence `[LATIN SMALL LETTER C, COMBINING ACUTE ACCENT, COMBINING CEDILLE]` that would result in one grapheme cluster. The problem is, that cluster would not be binary equal to `[LATIN SMALL LETTER C, COMBINING`

---

[2] http://www.unicode.org/reports/tr29/tr29-9.html has a complete definition of these rules.

`CEDILLE, COMBINING ACUTE ACCENT]`, but it would have the exact same meaning. To put it short, the ordering of the combining characters can still screw things up for us.

Once again the Unicode Consortium is one step ahead of us, and to solve this problem they have introduced the four Unicode normalization forms[3]. We won't go into detail about how these work here, but to put it short, they make sure that equivalent text will have the same binary representation if they are normalized to the same normalization for, effectively eliminating the problem we outlined above.

2.2.6 Summarized

Just to summarize the last couple of pages, we can say that for a Unicode library to be of any use for most programmers, it has to be able to read any Unicode encoding form, represent text-elements like grapheme clusters and be able to normalize the text to any of the normalization forms. Anything less than that will lead to strange behavior that will not make sense to programmers unfamiliar with Unicode.

2.3 The Standard Template Library

The Standard Template Library, a.k.a. the STL, is a part of the C++ standard library. The STL provides a collection of different classes that enable programmers to store and manipulate sequences of objects (elements). In this section we will roughly describe how the STL of built up, and finally how that relates to Boost.Unicode.

2.3.1 Containers

Containers are the core components of the STL. A container is a class that can store a set of objects of any given type. Some of the containers available in the STL are `std::vector<T>` and `std::list<T>`. Containers are very similar to normal arrays in many ways, but they are much more powerful. We will explain how in a little bit.

---

[3] A complete definition of these can be found at:
http://www.unicode.org/reports/tr15/tr15-25.html

2.3.2 Algorithms

Algorithms are another very important part of the STL. Algorithms basically model “things you can do to a Container”. They are operations that you can carry out on any of the Containers in the STL. One example of an algorithm in the STL is `std::copy` that copies all the elements in one Container over to some other Container.

2.3.3 Iterators

Iterators are the final important concept in the STL. Each Container, has an iterator type associated with it, and by using some Collection’s iterator you can move back and fourth over the data inside the container. If you think of a Container as an array, Iterators would be the same thing as a pointer into that array. Put bluntly, Iterators are the “glue” that connects a Container and Algorithms together. The Algorithms in the STL simply use iterators to do they’re thing on a container.

Iterators are not limited to Containers however. The C++ iostream library also provides a set of iterators that enable you to iterate trough a file or memory buffer. This enables you to do things like plugging some source file directly into a STL algorithm and having the results pop out at the other end of the algorithm. It’s all quite ingenious really, and a true testament to the powers of generic programming, a programming paradigm the STL in many ways spawned.

2.3.4 Concepts

But, what if you want to create a type of Container of Iterator on your own? How do you make it work together with the rest of the STL? Well, it’s all defined in the so-called Concepts in the STL specification[4]. These Concepts define every aspect of a Container or Iterator’s external interface. Everything from associated typedefs and function signatures, to performance guarantees are defined in these specs. As long as your classes follow these concepts, they can, through the wonders of templates and generic programming, be plugged into any part of the STL that uses models of these Concepts. Nice.

---

[4] The spec can be found at: http://www.sgi.com/tech/stl/

2.3.5 Summarized

What does all this have to do with Boost.Unicode? Well, a string of text is nothing more than a collection of objects (text-elements or code points) right? Then perhaps it would be a good idea to model any string classes after some of the STL Concepts? That would enable us to plug parts of our library directly into the myriad of existing STL algorithms, without us having to do anything to make it work. It's free functionality, and that is always a good thing.

3 Specification

3.1 Introduction

This document is dedicated to specifying the requirements of Boost.Unicode. The current state of the document is a snapshot of the specification as it was at the end of the project period. It, as the code itself, has morphed throughout the process and has therefore changed multiple times.

The specification is developed in collaboration with the Boost community by discussing and throwing ideas back as forth on Boost's mailing lists.

3.2 Requirements

3.2.1 Requirement overview

The goal of this project is to develop a Unicode string library for C++. The library should:

- Abstract away the complexity of working with Unicode strings, thus making it easy for programmers to use.
- Support manipulation of Unicode strings in all encodings. (UTF-8, UTF-16 and UTF-32)
- Support normalization of strings to all normalization forms. (D, C, KD and KC[5])

[5] As defined in The Unicode Standard Annex #15 - Unicode Normalization Forms

- Support manipulation of Unicode text-elements.
- Any container classes developed must be STL compatible to enable usage together with the existing components of the STL.
- At the end of the project period be in such a state that it can be directly submitted to the Boost community for review, and hopefully inclusion into the Boost library.
- Conform to all requirements of libraries set by the Boost community[6].
- Conform to the most recent Unicode standard at the time of development. (as of 1st of April 2005 The Unicode Standard 4.1.0[7])
- Conform to the C++ standard to the extent allowed by the targeted compilers.
- Not have any dependencies to external libraries or use any platform specific APIs like WIN32. The exceptions to this rule are the Boost libraries and the C++ standard library, but also dependencies to these libraries should be kept to a minimum.

3.2.2 Limits

Boost.Unicode should at its core provide a stable platform for Unicode manipulation that can later be used to implement further features. The library should not provide a complete implementation of the entire Unicode standard. (Something that could easily require double digit man-years worth of work to complete.)

Furthermore, the main focus here is to develop a *design* that works well and can easily be optimized. Focus is not on developing a blazingly fast implementation, as that is something that can (and will) be improved at a later point in time.

Consequently the library should **not**:

- Directly support reading Unicode text through the C++ iostream library. File io and other types of text parsing like the iostream library provides will be added at a later time because of the complexity associated with this (time needed to implement it).

---

[6] http://www.boost.org/more/lib_guide.htm#Requirements

[7] http://www.unicode.org/versions/Unicode4.1.0/

- Support any kind of locale specific behavior. (Like for instance locale dependant collation.)
- Be implemented with performance in mind, except for the cases where performance guarantees must be conformed to.
- Provide support for collation as defined by the *Unicode Collation Algorithm*.[8] Basic (binary) sorting of text should still be available though.
- Support any other kind of functionality not mentioned specifically in this specification.

3.2.3 Target users

Boost.Unicode is intended for C++ programmers wanting to represent and manipulate any kind of text inside their programs. Library users should not have to be proficient in "advanced" C++ topics like templates beyond what is already required to be able to use the STL (not much at all). For programmers used to the standard basic_string template (a.k.a. std::string and std::wstring), transition to the developed library should be close to effortless.

3.3 Use-cases

To get a more detailed view of the actual functionality that needs to be implemented, it is important to think about the library from the user's point of view. To do this, Use-case diagrams are useful as they model user's wishes without focusing on implementation details.

In the following section you can find a set of use-cases developed through discussion with the Boost community. These describe all the functionality that should eventually be implemented in the library.

[8] Description available at: http://www.unicode.org/reports/tr10/

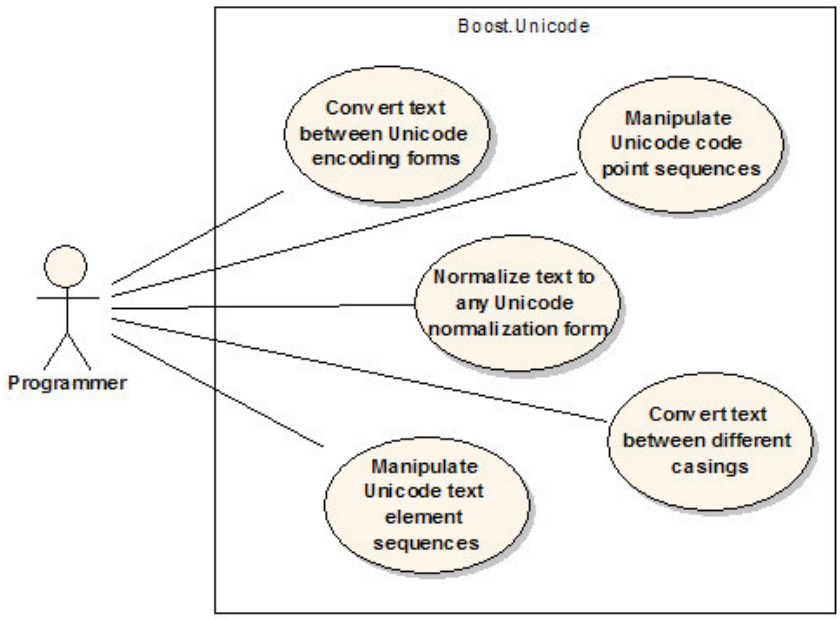

Figure 2 - Use-cases for the Programmer actor

3.3.1 Actors

The Use-cases associated with the Boost.Unicode library only have one actor; The Programmer. The Programmer represents any C++ programmer that is likely to use the library to represent text.

3.3.2 Use-case descriptions

Convert text between Unicode encoding forms	
Actor	Programmer
Description	The library should provide means for the programmer to convert a sequence of code units in any encoding form to the sequence of code points this sequence represents. It should also provide functionality for conversion the other way; from a code point sequence to an encoded code unit sequence.

Manipulate Unicode point sequences

Actor	Programmer
Description	The library should provide means for representing and manipulating any sequence of Unicode code points in a STL compliant container. This container should enable the user to choose the encoding form used internally to represent the code point sequence. The programmer should be able to change the sequence as needed and also perform searches inside the sequence.

Normalize text to any Unicode normalization form

Actor	Programmer
Description	The library should provide means for normalizing any sequence of Unicode code points to any of the 4 normalization forms defined in the Unicode standard. (NFC, NFD, NFKD and NFKC)

Convert between different casings

Actor	Programmer
Description	The library should provide means for converting any code point sequence to the simple case mappings of the same sequence.

Manipulate Unicode text-element sequences

Actor	Programmer
Description	The library should provide means for representing sequences of Unicode text-elements in a STL compliant container. This container should enable the user to choose the encoding and normalization form used internally to represent the text-element sequence. The programmer should be able to change the sequence as needed and also perform searches inside the it.

3.4 Documentation

Along with the library source code documentation describing the external interface of the library's components should be provided. This documentation should be developed using the boostbook[9] XML format, the default documentation format used in Boost. By using the Boost.Build[10] system, these XML files can be converted into a lot of different formats including HTML, PDF and *nix man pages. This format independency makes the format ideal for documenting source code libraries.

[9] For a complete reference of the boostbook documentation format, visit http://www.boost.org/doc/html/boostbook.html

[10] http://www.boost.org/tools/build/v1/build_system.htm

4 Design

4.1 Introduction

This section contains the complete design documents for Boost.Unicode. The document represents a snapshot of the state the design was in at the end of the development process.

4.1.1 Design format

Normally when designing an application, one would use UML to model the dataflow and classes that makes up the application and then implement the application from this.

For this project however, we have decided not to use UML as there are a few serious issues with regards to using UML to model C++ source code. Unfortunately UML lacks the ability to correctly model C++ concepts like typedefs and global functions. These are both extremely important concepts in the context of this library, and UML's inability to model these correctly forced us to do something else.

We have therefore chosen to simply create a set of tables that define the external interface of the different classes (typedefs and all). This does not look as good as a proper UML diagram, but it conveys more or less the same information and therefore does to job good enough.

The format is based around the following template:

General	
Template parameters	Describe the template parameters of the class.
Model of	Defines what generic concept the class is a model of. (This indicates that the class

	implements all functions required by this concept)
Associated types	Defines the public typedefs of the class
Functions	
Function name	Describes the functionality provided by this function.

4.2 Encoding traits

This component is meant to enable users to convert a sequence of code units (In either UTF-8, UTF-16 or UTF-32) to and from code points, as well as getting encoding specific traits like the type of a code unit, or whether or not the encoding is fixed width[11].

4.2.1 Encoding tags

Encoding tags are a set of classes that have no purpose but to represent the Unicode encoding forms generically (as template parameters). These should be named on the form `utfXY_tag`, where X is 8, 16 or 32, and Y represents the endian (big- or little-) of the encoding form (be or le if applicable). There should also be typedefs on the form `utfX_tag` and `utfXre_tag` that represent respectively the most efficient and reverse endian on the given platform.

4.2.2 Class encoding_traits

General	
Template parameters	Encoding: typename
Associated types	size_type: std::size_t code_unit_type: encoding dependant
Functions	
next_code_point	Increments the code unit iterator passed to it until it points to the previous code point in

[11] A code point is never more than 1 code unit wide in a fixed width encoding. Only true for UTF-32 in the context of Unicode.

	the sequence.
previous_code_point	Decrements the code unit iterator passed to it until it points to the previous code point in the sequence.
to_code_units	Converts its code point parameter to a sequence of code units and outputs it to the output iterator passed to it.
to_code_point	Converts the sequence of code units pointed to by the iterator passed to it to a code point.

Class description

The generic `encoding_traits` class should be specialized for all Unicode encoding forms, implementing all types and functions as required for the given encoding form. This enables the programmer to get encoding specific functionality (types, conversion functions, etc.) simply by changing the template parameter of `encoding_traits`.

4.2.3 Associated tests

Two test filed should be developed for this component. One that checks that all the Unicode encoding forms are correctly implemented, and one compile test that makes sure all associated types are present.

4.3 Code point string

General	
Template parameters	Encoding: typename - Determines the encoding form used internally in the string. Allocator: typename – Determines the allocator used internally by the string.
Model of	Boost.ReversibleCollection[12]
Additional types	encoding_type: Encoding allocator_type: Allocator

[12] http://www.boost.org/libs/utility/Collection.html

Additional functions	
insert	Inserts a code point or code point sequence at the position determined by its iterator parameter.
get_allocator	Returns the allocator object used internally in the string.
replace	Replaces a sequence of code points within the string with another sequence.
to_string	Returns a std::string with the UTF-8 equivalent of the code point string.
to_wstring	Returns a std::wstring with the UTF-16 equivalent of the code point string.
append/operator +=	Appends a code point or code point sequence to the string.
search/find	Searches for a code_point_sequence/code_point within the string.
search_end/find_end	Searches for a code_point_sequence/code_point within the string. (Starts search at the back)

Class description

This component is a STL compliant reversible container that can represent a string of Unicode code points in any encoding form.

4.4 Dynamic code point string

This component is a specialization of the `code_point_string` class for the encoding tag `dynamic_tag`. This specialization should have the exact same functionality as the `code_point_string` except it should be able to represent any encoding form internally, without having that affect the type of the class (Like it will in the normal `code_point_string` class).

The only change to the external interface is the addition of a set_encoding function templated on encoding that will change the encoding form used by the class internally to the form specified by the template parameter.

4.5 Normalization algorithms

The normalization algorithms component should be able to normalize a sequence of code points to any of the Unicode normalization forms.

4.5.1 Normalization tags

The normalization tags are a concept exactly like the encoding tags but representing normalization forms instead. They should be named on the form `nfX_tag`, where X represents one of the four normalization forms. (C, D, KC or KD).

4.5.2 The normalize algorithm

The `normalize` algorithm should be a generic function able to normalize a sequence of Unicode code points to any of the normalization forms defined in the standard. Which normalization form is used, is determined by the template parameter.

4.5.3 Associated tests

A file that tests the algorithms against the Unicode standard's NormalizationTest.txt[13] file. This file tests every possible facet of the normalization algorithms, and a successful test is a requirement for Unicode conformance.

4.6 Simple casing algorithms

This component should enable the programmer to convert a range of code points to any of the three simple casings (upper, lower or title).

[13] Available at http://www.unicode.org/Public/UNIDATA/NormalizationTest.txt

4.7 Boundary traits

This component should enable the user to get the boundaries of text-elements from within any code point sequence.

4.7.1 Boundary tags

Boundary tags are a set of classes that have no purpose but to represent the Unicode text-elements generically (as template parameters). The boundary tags available are `grapheme_cluster_boundary_tag` and `word_boundary_tag`.

4.7.2 Class boundary_traits

General	
Template parameters	BoundaryTag: typename
Functions	
next_break	Increments its code point iterator paramterer until it points to the beginning of the next text-element.
previous_break	Decrements its code point iterator paramterer until it points to the beginning of the previous text-element.

Class description

The generic `boundary_traits` class should be specialized for all boundary types, implementing all types and functions as required for the given boundary type. This enables the programmer to get encoding specific functionality (types, conversion functions, etc.) simply by changing the template parameter of `encoding_traits`.

4.7.3 Associated tests

The boundary_traits class should be tested by running it through the files `GraphemeBreakTest.txt` and `WordBreakTest.txt`[14] defined in the

[14] Both files are available at http://www.unicode.org/Public/UNIDATA/auxiliary/

Unicode standard. These tests check that all break rules are correctly implemented and a successful test is required for Unicode conformance.

4.8 Text element

This component should enable the programmer to represent any type of text-element, be it grapheme clusters or words.

4.8.1 Class text_element

General	
Template parameters	Encoding: typename Normalization: typename Boundary: typename Allocator: typename
Associated types	encoding_type: Encoding normalization_type: Normalization boundary_type: Boundary allocator_type: Allocator.
Functions	
to_string	Returns the UTF-8 equivalent of the text-element.
to_wstring	Returns the UTF-16 equivalent of the text-element.
get_allocator	Returns the allocator object used internally in the text-element.

4.9 Text element string

This component represents a sequence of text-elements.

General	
Template parameters	Encoding: typename Normalization: typename

	Boundary: typename Allocator: typename
Model of	Boost.ReversibleCollection[15]
Additional types	encoding_type: Encoding normalization_type: Normalization boundary_type: Boundary allocator_type: Allocator.
Additional functions	
insert	Inserts a text-element or text-element sequence at the position determined by its iterator parameter.
get_allocator	Returns the allocator object used internally in the string.
replace	Replaces a sequence of text-elements within the string with another sequence.
to_string	Returns a std::string with the UTF-8 equivalent of the text-element string.
to_wstring	Returns a std::wstring with the UTF-16 equivalent of the text-element string.
append/operator +=	Appends a text-element or text-element sequence to the string.
search/find	Searches for a text-element-sequence/text-element within the string.
search_end/find_end	Searches for a text-element-sequence/ text-element within the string. (Starts search at the back)

4.10 Dynamic text element string

This component is to text_element_string what the dynamic code point string component is to code point string.

The only change to the external interface of `text_element_string` is the addition of a `set_encoding` function templated on encoding that

[15] http://www.boost.org/libs/utility/Collection.html

will change the encoding form used by the class internally to the form specified by the template parameter.

5 Implementation

5.1 Introduction

In this section we have listed some general information about the actual implementation of the Boost.Unicode library, as well as pulling forward some implementation details we are particularly happy with.

5.2 Programming environment

5.2.1 IDE and compiler

As our main development IDE we have chosen to go with Microsoft Visual Studio 2003. This tool suite provides excellent editing facilities, a great debugger, and most importantly a fairly standard compliant C++ compiler. We also considered using the open-source Dev-Cpp IDE (which comes bundled with the GCC/MinGW compiler), but in the end VC++ turned out to be the better choice. Tests should still be run on GCC/MinGW to ensure portability though; we just won't de doing any development on this platform.

5.2.2 Coding standard

All code and documentation will be developed in English and must follow the coding guidelines defined by the Boost community[16].

[16] http://www.boost.org/more/lib_guide.htm#Guidelines

5.3 General

5.3.1 Source code dependencies

The source code should not have any kind of dependencies to external libraries and API's like for instance WIN32. This requirement is to ensure that the developed code is portable across as many compilers and platforms as possible.

The exceptions to this rule are the C++ standard library and the Boost libraries version 1.32. These libraries are available for all platforms, and therefore their usage doesn't limit the portability of Boost.Unicode.

5.3.2 Exception safety

The developed library should provide exception safety in all functions. Exception safety means that if any operation performed inside the library results in an exception being thrown, the program should remain in the same state it was before the error occurred. I.e. a throwing function should not modify any of its parameters or associated class' member variables in case of an exception.

5.3.3 Thread safety

The developed library should provide the same kind of thread safety that the STL provides. That is, two or more threads reading from the same container simultaneously is thread safe, but simultaneous reads/writes to the same container will result in undefined behavior. It's up to the user to secure such accesses using some sort of thread synchronization primitives.

5.4 Encoding traits

5.4.1 The classes

After having designed the `encoding_traits` class, implementing the specializations was trivial. Below is an outline of the implementation

of the UTF-32 encoding_traits class. The other specializations follow the same pattern.

```
template<>
class encoding_traits<utf32_tag>
    {
    public:
        // Public typedefs:
        typedef boost::uint32_t code_unit_type;
        typedef std::size_t size_type;

         // Constants:
        static const size_type min_code_point_width = 1;
        static const size_type max_code_point_width = 1;

        // Functions:
        template<typename ri_iter>
        static inline void next_code_point(ri_iter& start)
            {
            …
            }

        template<typename rb_iter>
        static inline void previous_code_point(
            rb_iter& start)
            {
            …
            }

        template<typename input_iter>
        static inline code_point_type
            to_code_point(input_iter start)
            {

            …
            }

        template<typename wi_iter>
        static void to_code_units(const code_point_type&
            code_point, wi_iter out)
            {
            …
            }

    };
```

5.4.2 Reversed endian encodings

The reversed endian encodings were implemented simply by creating an Iterator (boost/unicode/detail/util.hpp#reverse_endian_iterator) that can wrap any iterator type, and when dereferenced return the value pointed to by that iterator, but in the reversed endian. This enabled us to create a basic specialization like the one above, but inside each function wrap the iterator parameters in the reverse_endian_iterator class and forward the call to the correct endian specialization. By doing this we saved ourselves the trouble of having to implement all traits functions for both endians.

5.5 Dynamic code point string

The dynamic code point string is built up around a abstract class called `code_point_string_wrapper_base`. This class has a lot of virtual functions that enable manipulation of a code point sequence. Then we inherited a class, `code_point_string_wrapper`, templated on encoding form from this abstract class, and implemented all of the virtual functions from the base. This enabled us to create an instance of `code_point_string_wrapper` for any encoding tag for which there is an implementation of encoding_traits. The class hierarchy looks something like this:

```
class code_point_string_wrapper_base
    {
    public:
        virtual some_function() const = 0;
    };


template<typename Encoding >
class code_point_string_wrapper
    :
    public code_point_string_wrapper_base
    {
    public:
        virtual some_function() const
             {
             // Do encoding specific stuff here.
             }
```

Now we could implement the `code_point_string<dynamic_tag>` specialization to work through a `code_point_string_wrapper_base` pointer, and create a templated `set_encoding` function that would set the pointer to point to an `code_point_string_wrapper` instance for the encoding indicated by `set_encoding`'s template parameter, effectively changing the internal encoding used by the string.

```
code_point_string<dynamic_tag> some_string;

// Sets the internal encoding to UTF-8:
some_string.set_encoding<utf8_tag>();
```

5.6 Unicode character database

The Unicode Character Database is an enormous collection of data related to the different code points in the Unicode standard. This data is used to find things like upper and lower case mappings of code points, getting their decomposition mappings and other data that is important for things like normalization.

In Boost.Unicode this database is implemented as one gigantic static array. This has been done to have the data compile and become a part of the executable, instead of having to rely on reading them from an external file, something that can be highly unreliable, not to mention slow.

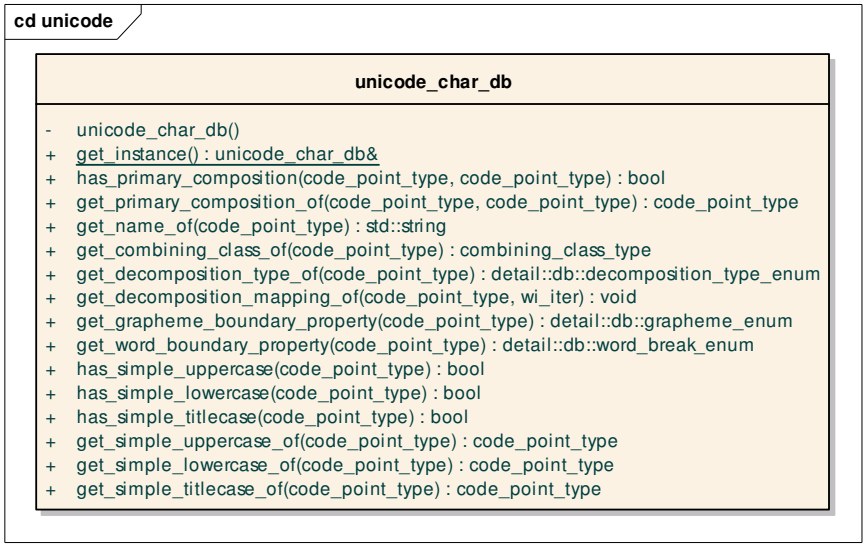

Figure 3 - The Unicode Character Database interface

To actually accomplish this we decided to make an external program that would parse through the UCD files and generate a compilable source code containing the data. The file generated would then be copied into the library and the rest of the code would then have access to the UCD data. To make the access to UCD data easy, a Singleton class was made function as the interface to the database. (Figure 3 - The Unicode Character Database interface)

To be able to save the data in a combilable file. We needed to design a data structure. We couldn't just have one large array of structs with the properties as there are 0x10FFFF code points in the standard, this would lead to a lot of redundant data stored in the database as a lot of code points have the same data.

We then decided to use a lookup table which contains a code point, range, and a reference to the struct with the data for current code point. The range is used when there are several code points following each other that have the same property. The range property tells how many code points after current code point there are in this range. Searching through the index table is done with a binary search algorithm which is tailored to take ranges into account. This is expected to save a lot of space while the speed loss is close to none. The binary search algorithm would only use 11 steps before finding the right code point, worst case, with around 20000 entries in the database.

```
struct db_entry
    {
    char *name;
    unsigned char canonical_combining_class;
    decomposition_type_enum decomposition_type;
    const code_point_type *decomposition_mapping;
    code_point_type simple_uppercase_mapping;
    code_point_type simple_lowercase_mapping;
    code_point_type simple_titlecase_mapping;
    };
```

The above struct shows the properties returned from a lookup on a code point. All fields should ideally have been defined as a const, but doing so increased the compile time, for some reason of another, way beyond acceptable. The reason the `decomposition_mapping` has to be a pointer is because this value can range from 0 to N code points, where N is unknown. That makes it impossible to use a static array as the size has to be know at compile time.

```
const db_index index_table[] =
{
// code point, range, reference
0x0000, 0, &unicode_database[0],
```

```
0x0001, 0, &unicode_database[1],
0x0002, 0, &unicode_database[2],
...
}

const db_entry unicode_database[] =
{
"<control>", 0, decomp_none, &decomp_map_zero[0], 0x0,
0x0, 0x0,
"<control>", 0, decomp_none, &decomp_map_zero[0], 0x0,
0x0, 0x0,
"<control>", 0, decomp_none, &decomp_map_zero[0], 0x0,
0x0, 0x0,
...
}
```

Above is an example of actual database entries, with with the forth column pointing to a new array with the decomposition mapping of the code point.

```
const code_point_type decomp_map_zero[] = {0};
const code_point_type decomp_map_00A0[] = {1,0x0020};
const code_point_type decomp_map_00A8[] = {2,0x0020,
                                           0x0308};
```

`decomp_map_zero` is a special case since most code points don't have a decomposition mapping, and listing one for each code point is a waste of space when you can just point them all to the same. The first value in decomposition map is always how many code points the code point is decomposed into. Rest of the values are the actual decomposition. Code point U+00A8 would be decomposed to 0x0020 and 0x0308 as given above.

6 Testing

6.1 Introduction

To test the different components of the Boost.Unicode library, we have employed a testing methodology used heavily in the Boost libraries called unit-testing. Unit-testing is done by creating separate cpp test files that test every function of a given class or other construct(using special macros defined in boost). These files are then compiled, linked against the Boost.Test[17] library and then finally run. This will result in the generation of a test report that lists any errors that may have occurred.

For the official Boost libraries, these tests are used to test the entire Boost library on a lot of different compilers, a process called regression tests. Every night the entire test code tree is run on a multitude of different compilers and platforms. The results of these tests are then posted online so library authors can see how their library did. This testing method makes it much easier to run tests as the developers don't have to have every platform and compiler themselves.

By following this testing methodology, Boost.Unicode is ready for easy integration with the existing Boost library if that should ever happen.

[17] Available at http://www.boost.org/libs/test/doc/index.html

6.2 Tests

In the section below we describe a couple of the more comprehensive test files we have developed, along with a couple of bugs we found because of them.

6.2.1 Normalization

For testing normalization the Unicode Consortium has made file holding test information on every code point and expected valuesi. This makes verifying the normalization algorithm easy to ensure it's working correctly. It contains information on expected results after given input to a normalization algorithm. This file is quite large (2mb) and tests every single code point against its expected normalized value in all the 4 different normalization forms. For this file we wrote a separate test program that parses through the file and uses the data it collects to normalize and test that it gets the correct result back form the normalization algorithms. Since normalization is heavily dependent of the Unicode database, this will get tested as well.

Results

When the first version of the normalization test program was done, we had amazingly few errors when we ran it. Only 1 normalization test failed. This led us to two bugs, one in our binary search algorithm and another in the Unicode Character Database implementation. When those were fixed the entire normalization test worked flawlessly, over 500000 test total.

6.2.2 Boundary traits

The boundary_traits classes also needed testing. As with other parts of Unicode that need an algorithm to function, they have provided a set of test files. Since both files for testing grapheme and word break where built up the same way, making a tester for word break was quick once grapheme was working. This lead to having a working test program for word break before the algorithm was started on. This eased the development of the word break algorithm, but lead to more trail and error development then actually thinking thoroughly through the code written.

Giving a full test of every possible boundary test is impossible due to the unlimited number of possible strings. So the test these files contain is just trying to cover all basic concepts. Since word break uses grapheme clusters, meaning that each boundary gets tested between 800-1000 times, which should cover most of the ground.

Results

The boundary traits tests worked flawlessly as soon as the classes were finished, ensuring us that our implementation worked correctly.

7 The development process

7.1 Evolutionary development

To give you a little feel for how the journey up to the current library implementation has been, this section has been dedicated to give a brief summary of how the work progressed, and what changes have been made to the library on the way. The reason for doing this is that the library, because of the evolutionary development model, has changed a lot since the first prototype, and thus the specification and design documents don't accurately reflect the process as a whole. This section complements and expands on much of the information you will find in the worklog, gannt diagrams, Iteration plans and evaluation documents. (See appendices)

Introduction

The development kicked off after the Pre-engineering project report was delivered at the 26th of January 2005. The first 4 days were spent reading through a lot of Unicode and STL documentation, until the 1st iteration started at the 31st of January.

7.1.1 Iteration 1

The first iteration was started with a short planning period to get a grasp of what we wanted to do (Resulting in Iteration plan 1), some more research on Unicode followed along with some discussion with the Boost community, before we started specifying and designing the first prototype. The `encoding_traits` class (more or less the same thing it is now, but without the UTF-8 specialization) and two classes called `encoded_string` and `character_set_traits` were the results of this process.

The `encoded_string` class later went on to be what is now known as the `code_point_sting<dynamic_tag>` specialization and it provided more or less the same functionality.

The `character_set_traits` class was a concept introduced to have the library support other character sets than Unicode. (Shift-JIS for instance) The idea was to separate all character set specific traits into this class (like `code_point_type`), and have all string classes use that class to define its external interface. This idea was eventually scrapped as it really made little sense to begin with. Different character sets are not really parallel concepts like different encoding forms are, and creating an interface for `character_set_traits` that would work for all encodings, would be next to impossible.

At the end of the iteration a short period of writing and running test files was carried out before releasing the first prototype.

7.1.2 Iteration 2

Iteration 2 was started at the 21st of February, and in the initial planning it was decided that we should concentrate on adding the normalization/casing algorithms, and as a consequence of that, the Unicode Character Database.

The normalization functions developed here were more or less the same as the ones in the current implementation, with one exception. They were not generic. (They didn't take the normalization form in as a template parameter, but were rather named according to normalization form)

The code generators created for building the database source code, are also more or less the same as they were then, but more functionality has been added to them along the way as more and more fields were added to the database.

Towards the end of the iteration (which was planned to be 2 weeks long), we found out that we wouldn't be able to finish what we were supposed to, and furthermore we got the feeling that 2 weeks was a little bit too short for one iteration. This resulted in the decision to extend all iterations to 3 weeks just like the first one, and thus cropping the number of iterations to 4. We also found out that we forgot to take easter into consideration in our original plan, so a one week vacation was added to the second week of iteration 3.

As in the first iteration, test files for the developed components were written and run before finishing off with another evaluation.

7.1.3 Iteration 3

Iteration 3 was broken up a bit because of the Easter vacation in the middle of it, so the amount of work getting done was not as great as it should have been.

The iteration started (as usual) with a planning period, and it was decided that the iteration should focus on completing the `encoding_traits` (adding UTF-8), create a `code_point_string` class locked to an encoding form (as opposed to the dynamic `encoded_string` class that was there at the time), add thread safety to `code_point_string`'s reference manager and upgrade the Unicode version from 4.0.1 to 4.1. The change to the new Unicode version went without incident. The only changes necessary were the generation of new database sources based off of the new version, a tiny change in one of the normalization algorithms to reflect a change of wording inside the standard, and finally adding a few tests to the already existing ones to test the changes made. All in all, we went a couple of days beyond the deadline for the prototype on this iteration, and that skewed Iteration 4 a little with respect to the timetable, something we were to pay for at the end of that Iteration.

Everything that was created in iteration 3 is still in the code tree mostly unchanged.

7.1.4 Iteration 4

The final iteration started, a little late of the development focused on the development of text_elements.

In the initial specification and design activities, the plans for the `boundary_traits`, `text_element` and `text_element_string` classes were finalized. The classes where then implemented and tests were written and run.

The implementation of these classes unfortunately took a bit longer than first anticipated, and therefore we were forced to drop implementing the dynamic_tag specialization of `text_element_string`. Furthermore the Boost documentation and test files for the `text_element` class was dropped as we didn't have time to write them up. (You can still take a look at the design section to get an idea of the interface.) Dropping these things was absolutely necessary for us to get the time to finish off the more important things like the project report and setting up the Boost.Build system for our library. All in all we probably underestimated the work required to complete the report in time, and that made this a very hectic iteration indeed.

The components implemented in this iteration, remain unchanged in the final code tree.

7.2 Objective critique

The project has resulted in a library that more or less does what we initially set out to accomplish. Programmers can, using the developed library, manipulate Unicode text inside their applications (both as sequences of code points and sequences of text-elements). The library is STL compliant, and

There are however a few things that are not all they could (and should) be.

The most obvious thing is the fact that the specialization of `text_element_string` providing support for dynamic encoding setting is missing. This component was mentioned in the specification and design documents, but we never got the time to actually implement it. This is of course regrettable, but the encoding tag `dynamic_tag` can still be used with the class (you just won't be able to actually change the encoding), so the loss is not really that big.

Another thing that is less than optimal, is the Boost documentation provided on the CD-ROM bundled with the report. This is currently a bit on the slim side, and could really use some design explanations and usage examples. The reason for this shortcoming is, as with the above one, lack of time in the final sprint towards project completion.

The final, and perhaps most serious shortcoming of the library is that it currently only works on Visual C++ 2003 (and therefore only the Windows platform), and not on GCC/MinGW 3.4 like we were aiming for. This basically boils down to a couple of bugs that suddenly appeared when building on MinGW late in the project period. Due to the time crunch, we were not able to track down and fix these bugs before delivery so the library will unfortunately not work on these compilers in the state provided on the CD-ROM.

7.3 Where do we go from here

Even though the developed library has more or less all the functionality we set out to implement, it is still a far cry from a complete Unicode implementation. As mentioned earlier in this

report, developing a complete Unicode library is an extremely time consuming task, and cannot be done within the confined time frame of a final project. Boost.Unicode does however provide a good platform for building such a library, and we are both keen on continuing development also after our time here at GUC.

The first and most important feature that should be added to the library is support for reading Unicode through C++' iostream library. This was something that was initially considered for this project, but the workload that would be involved in doing this turned of to be in the region of a final project on itself, so it was quickly apparent that we wouldn't get the time to do it.

Another thing that really should be implemented is support for true locale dependant collation of strings. This would enable users to sort strings in the order that is correct for their language (In Norwegian 'Å' would be sorted after 'Z', on English it would group together with 'A'). Again, the workload associated with something like this, is enormous, and we would never have had the time to implement it during this project.

There is also a lot of other Unicode related functionality that should be available in a complete Unicode implementation. Things like regular expressions and directionality (Support for scripts written from right to left, like Arabic) are obvious things, and these should also be added over time.

7.4 Subjective evaluation

In this section we will give a purely subjective evaluation of the fruits of our labor, as well as an evaluation of our own efforts throughout the development process.

7.4.1 Working on a project

The months we have spent working on this project have been truly interesting. None of us have ever worked on a project of this magnitude and complexity, and it has been a very rewarding experience. At the beginning of the process we had a little trouble getting used to the fact that we had to plan all aspects of the development along the way. Being hobby developers, we are used to working after the "code and fix" methodology, so the fact that we actually had to plan a little ahead this time was somewhat unfamiliar in the beginning. We quickly got into it though, and we feel we had

pretty good control over the project throughout the process. We did get a little short on time towards the end (as mentioned earlier), and because of that,a few parts of the library are a little rough, but that is to be expected when dealing with inexperienced developers like ourselves.

7.4.2 Organization

The work on this project has been carried out in a relatively orderly fashion. After a brief period of homelessness where we didn't have a proper room to work out of, we finally got a couple of seats in room A218 at GUC. From there on in we were there working more or less from 9 to 5 every weekday. We feel having a dedicated location like this to work out of was imperative to the success of our project as it enabled us to always be within arm's reach of each other. This made maintaining information flow a breeze, and we could focus less on written documents to keep each other updated on the goings on in the code. Actually having to get out of bed and go to another location to work also made it easier to actually work when you were supposed to, as working from home tends to lead to late breakfasts, early lunches and generally a lot of time wasting. (We're talking out of experience here.)

7.4.3 Workload balance

The division of the workload has worked out really well through the course of the project. Because of the number of different components in the library, it has been relatively easy to split the components down the middle, and having each of us working on "our own" separate components.

8 Conclusion

All in all we are very happy with the results of our project. We are very pleased with how the library has turned out, and how easy it is to use. Although we don't really feel the library is ready for inclusion into Boost just yet, Boost.Unicode as is stands today is very a good platform to base further development on, and hopefully one day leading to a complete Unicode implementation.

As students we have also learned a great deal about how working on a big programming project over an extended period of time really is. This experience will surely make the next programming project we get out hands on easier to get through.

9 Bibliography

Scott Meyers. *Effective C++ Second Edition*. 2004

Angelika Langer and Klause Kreft. *Standard C++ IOStreams and locals* 1999

Erich Gama et.al *Design Patterns Elements of Reusable Object-Oriented Software*. 1995

Andrei Alexandrescu *Modern C++ Design*. 2001

Robert Sedgewick *Algorithms C++*. 1992

The Unicode Consortium *The Unicode Standard 4.1.0*. 2005

Mark Davis, Martin Dürst *Unicode Standard Annex #15 Unicode Normalization Forms*. 2005

Mark Davis *Unicode Standard Annex #29 Text Boundaries*. 2005

10 Appendices

Appendix A – Definitions

Appendix B – Pre-engineering project report

Appendix C – Original development plan

Appendix D – Real development overview

Appendix E – Iteration plan 1

Appendix F – Iteration plan 2

Appendix G – Iteration plan 3

Appendix H – Iteration plan 4

Appendix I – Status report 1

Appendix J – Status report 2

Appendix K – Worklog

Appendix L – CD-ROM with contents:

- The Boost libraries version 1.32
- Boost.Unicode code-tree
- HTML documentation for Boost.unicode
- Installation notes
- This report