Onestring Documentation

Release 2.0

MousePaw Media

Jun 17, 2022

CONTENTS

Arctic Tern is a string class for C++ with full Unicode (UTF-8) support and seamless compatibility with std::string and C-style strings.

See Arctic Tern's `README.md`, `CHANGELOG.md`, `BUILDING.md`, and `LICENSE.md` for more information.

CHAPTER

ONE

CONTENTS

1.1 About Trilean

A trilean is an atomic data type with three distinct states: “true”, “maybe”, and “false”. It is designed to behave like a boolean, but with the additional feature of “uncertainty.”

In order to understand trilean, one must remember that “true” and “false” are *certain* states, entirely distinct from and un-equivalent to “maybe”. All of trilean’s behavior is based around this rule.

1.1.1 Why Use Trilean?

The logic of trilean has historically been achieved in C++ using either an enumeration or a pair of booleans. However, trilean offers a couple of distinct advantages:

- A trilean variable is exactly one byte in size.
- All three states can be represented in a single variable.
- Trilean is fully compatible with boolean and its constants.
- Conditional logic with trilean is simpler and cleaner.

1.2 Using Trilean

1.2.1 Including Trilean

To include Trilean, use the following:

```
#include "arctic-tern/tril.hpp"
```

1.2.2 “Maybe” Constant

Along with the normal `true` and `false` constants provided by C++, trilean offers a `maybe` constant. This new constant is designed to be distinct from `true` and `false`, yet usable in the same manner for trileans.

1.2.3 Defining and Assigning

Defining a new trilean variable is simple.

```
tril foo = true;
tril bar = maybe
tril baz = false;
```

In addition to the three constants, you can assign other booleans and trileans.

```
bool foo = true;
tril bar = foo;
tril baz = bar;

// foo, bar, and baz are all 'true'
```

1.2.4 Conditionals

Obviously, one can compare a trilean against of its state constants directly.

```
tril foo = maybe;

if(foo == true)
    // code if TRUE...
else if(foo == maybe)
    // code if MAYBE...
else if(foo == false)
    // code if FALSE...
```

However, one can also test a trilean in the same manner as a boolean.

```
tril foo = maybe;

if(foo)
    // code if TRUE...
else if(~foo)
    // code if MAYBE...
else if(!foo)
    // code if FALSE...
```

You will notice that, in addition to the familiar tests for "true" (`if(foo)`) and "false" (`if(!foo)`), trilean has a third unary operator, ~, for "maybe" (`if(~foo)`).

Important: Remember, neither the "true" or "false" conditions will ever match "maybe".

This basic behavior is what makes trilean so useful. For example, you may want to repeat a block of code until you encounter specific scenarios to cause it to either pass or fail, using something like `while(~foo)`.

1.2.5 Comparisons

Trilean can be compared to booleans and other trileans using the == and != operators.

```
tril foo = true;
tril bar = maybe;
bool baz = true;

if(foo == bar)
    // This fails.

if(foo != bar)
    // This passes.

if(foo == baz)
    // This passes.

if(baz == foo)
    // This passes.

if(baz == bar)
    // This fails.
```

1.2.6 Switch

The idea of allowing a trilean to cast to an integer was discussed and debated in great deal. Finally, the decision was made to prevent casting a trilean to anything but a boolean (discussed later).

This means that trileans **are not compatible with switch statements**. While this may be initially disappointing to anyone used to using an enumeration for three-state logic, one will notice that an if-statement covering all three states of a trilean has at least 4 less lines of boilerplate.

```
tril foo;

/* This code demonstrates an if statement covering all three states
 * of a trilean. */

if(foo)
{
    // Some code.
}
else if(~foo)
{
    // Some code.
}
else if(!foo)
{
    // Some code.
}
```

1.2.7 Gotchas

Casting to Bool

In order to preserve the core logic that "maybe != true" in statements like `if(foo)`, casting a trilean to a boolean causes "maybe" to be converted to "false".

```
tril foo = maybe;
bool bar = foo;

// bar is now 'false'
```

In most cases, it is recommended to use the `certain()` function.

Note: In case you were wondering, we ensured that "maybe != false" in comparisons and conditionals by separately overloading the `!`, `!=`, and == operators.

1.3 Certainty

Because trilean stores its data in two bits, it is possible for a variable to track its last certain state. In other words, if a trilean is "true" or "false," and then is set to "maybe", that true/false value is still being stored behind the scenes.

To make this useful, trilean offers a `certain()` function, which returns the last certain state of the variable without actually modifying itself.

```
tril foo = true;
foo = maybe;

bool bar = foo.certain();

// bar is now 'true', while foo is still 'maybe'
```

This behavior can also be used to revert a trilean to its last certain state.

```
tril foo = true;
foo = maybe;
foo = foo.certain();

// foo is now 'true'
```

1.3.1 Implications

The concept of "certainty" technically allows one to recognize and use four trilean states:

- Certain true (`if( foo )`)
- Uncertain true (`if( ~foo && foo.certain() )`)
- Uncertain false (`if (~foo && !foo.certain() )`)
- Certain false (`if (!foo)`)

1.3.2 Uncertainty Variables

The "magical" behavior of assigning the constant "maybe" not affecting the previous certain state is achieved through "uncertainty" variables. Any time an uncertainty is assigned to a trilean, only the uncertainty of the trilean is affected.

The constant "maybe" is usually the only uncertainty object you will interact with. However, it is possible to create your own `certainty`. Be aware that this data type does not provide any mechanism for modifying it after creation.

```
uncertainty my_maybe(true);
uncertainty my_certain(false);
```

As is expected, an uncertainty can never match "true" or "false", or be directly cast to a boolean. However, the ~ operator works as with trileans.

```
if(~my_maybe)
    // This passes.

if(~my_certain)
    // This fails.

if(~my_certain == false)
    // This passes.
```

The usefulness of an uncertainty variable is, quite probably, limited to allowing manipulation of a trilean's certainty.

CHAPTER

TWO

INDICES AND TABLES

Note: The index is still a work in progress. If you'd like to help with this, please see our Contribution Guide.

- genindex
- search