Lexical Casting in C++

stephan@s11n.net

August 14, 2005

Abstract

CVS Info: $Id: lexical_casting.lyx,v 1.5 2005/05/25 18:19:13 sgbeal Exp $

This paper covers the topic of "lexical casting": what it is, how it can be used in C++, and some potential cases for when to use it. It also develops a C++ class which encapsulates lexical cast behaviours. It is aimed at intermediate-level+ C++ programmers (that is, `++newbie`). That said, the software developed in this paper is suitable for use by new C++ users, but such users are recommended to learn a bit about type conversions, especially *implicit* type conversions, as they are defined by the ISO C++ standard. (*That* said, i personally know *embarrassingly little* about them, and learn about them more "the hard way" than anything else.)

The entire text of this document, and the accompanying source code, are released into the Public Domain.

Any sort of feedback, and suggestions for improving this paper, are welcomed: please email the author at the address shown above.

Change History:

August 2005: Minor touch-ups and corrections.

31 Dec 2004: Changed the impl of istream>> again.

28 Dec 2004: Re-implemented istream>> operator.

20 Aug 2004: Minor textual corrections (missing words and such).

19 Aug 2004: Initial release

Contents

1 Introduction

1.1 Disclaimer

i am *not* a C++ guru, nor have i been formally educated as a software developer (rather, it is a hobby that accidentally turned into a profession). Thus it is quite possible that this paper contains some misleading or downright incorrect statements about C++. i would ask any more-enlightened reader to please gently prod me the proper direction should (s)he find any information here which is misleading or blatantly incorrect.

1.2 Background

Programmers with experience in so-called *loosely typed* languages, like Perl or PHP, are familiar with language syntaxes allowing statements like the following:

```
foo = 17;
foo = ”17”;
foo = ”no, i said seventeen.”;
foo = 17.33;
```

Many languages, including C++, are *strongly typed*, and disallow this type of leniency, requiring that the programmer specify exactly what *type* he means when he creates a variable, as highlighted here:

```
int foo = 17;
foo = ”17”; // ERROR: not an integer, though it looks like one to you and me.
foo = ”no, i said seventeen”; // ERROR: not an integer.
foo = 17.33; // WARNING: will drop the post-decimal-point part.
```

While such strong typing is generally a Good Thing in applications, especially in large-scale applications, i personally sometimes miss the ability to treat a type essentially however i want to. Throughout the rest of this paper we will discuss how this can be achieved, at least to some level, and how we can generically make use of this technique in our own C++ code.

Before continuing, let's just take a quick peek at what's to come:

```
typedef std::map<lex_t,lex_t> MapT;
MapT map;
map[4] = "one";
map["one"] = 4;
map[123] = "eat this";
map["123"] = "this was re-set";
map['x'] = "marks the spot";
map["fred"] = 94.3 * static_cast<double>( map["one"] );
map["fred"] = 10 * static_cast<double>( map["fred"] );
int myint = map["one"];
```

Reminder: *this is C++, not Perl.*

And yes, the above code is perfectly legal and does exactly what it appears to do. Yes, even when compiled with a C++ compiler.

Some purists are likely to respond that the above code is not in line with common C++ thinking, and denounce it as ”unduly type-unsafe” or ”poor style.” Could we live without this type of flexibility in C++? *You bet!* However, this paper will attempt to convince the reader that for some use cases the ability to loosely type variables can greatly simplify implementation and usage of a particular piece of code.

1.3 What is Lexical Casting?

Let's take a moment to examine the phrase "lexical casting."

We know, from our experience with C++, what casting is. In short, it is the ability to convert an object of one type to an object of another type. C++ allows a variety of techniques for casting types, both via *explicit* (user-specified) and *implicit* (compiler-determined) type conversions. One could argue that *implicit* conversions are inherently *explicit*, as class designers can manipulate their classes in order to tell the compiler what types of conversions are and are not allowed, but here we mean *implicit* to mean cases where *users* of a class need not tell the compiler that they want a specific type conversion (i.e., cast) to take place.

The word *lexical* refers to *words* or, as we know them in C++, *character strings*.

So, *lexical casting* is:

> The ability to cast strings to other data types, and non-string types to strings. (Well, we *can* also cast strings to and from each other, but there is little point in doing so except to convert between different implementations of string-like types, e.g. a (`char *`) to or from `std::string`.)

While the term does not, in and of itself, imply any sort of limitation on what can and cannot be lexically cast, this paper is limited to the lexical casting of so-called "streamable" data types, as defined in the next section. As we will soon see, limiting ourselves to streamable types is not only a question of practicality, but it is also not as limited as one might initially think. Consider: all of C++'s built-in data types are streamable, as is `std::string`, and those types cover the majority of data which we would normally want to lexically cast. One might argue that *pointers* aren't generally streamable, but pointers are not *types* in and of themselves, but are *type qualifiers*. That is, a pointer *never* exists by itself, but is instead applied as a *qualifier* for a type.

1.4 Type requirements: what types can be lexically cast?

For our purposes, any type meeting the following requirements can be reliably lexically cast:

- It must have *complementary* `istream>>` and `ostream<<` operators. This means that `istream>>` must be able to populate an object based on data which its `ostream>>` counterpart has written. Whether the operators are member functions or not is irrelevant for our purposes.
- It must be *Default Constructable*. That is, the following expression must be valid:
  `T t;`
- It must be *Assignable to it's own type*. That is, the following expression must be valid:
  `T t = T();`
- While not a strict requirement, it should also have a copy constructor, such that the following expression is valid:
  `T t( T() );`

Any type which meets these requirement will, in theory, be lexically castable by the framework we will develop in this paper. In other words, the framework will cover *at least* C++'s predefined PODs, plus `std::string`. It will also cover any user-written types which meet these requirements.

There may be other factors which prohibit a type's use with our framework, one example being the requirement of having high-precision accuracy in floating-point numbers. (We will cover that point later on.)

1.5 Motivation

The original motivation for this paper, and the software developed through it, came about while working on a generic database layer for C++. Through that work the techniques shown here were applied in order to simplify the API related to fetching information from and updating to a database. As databases normally work with a fairly limited set of POD [Plain Old Data] types, lexical conversions are very well-suited for such a task. Rather than having to implement a handful (or more) of accessor and mutator functions, like `getInt()`, `getDouble()`, `getString()`, or overloaded variants, we end up with one `get()` and one `set()` function, both of which can act on behalf of all of the basic POD types (indeed, for all lexically castable types) by simply accepting or returning a `lex_t` object in place of a "strong" type.

Aside from usage in such a database framework, the `lex_t` type presented here can, in many cases, act as a generic replacement for type-specific "assignment proxies", as presented in detail in Scott Meyers' classic work, *More Effective C++*, Item 30. (If that book isn't on your bookshelf, next to your computer, in your book bag, or otherwise in your possession, *go buy a copy*! If, on the other hand, you don't have a copy because you have already memorized the book and gave it to a more needful colleague, then *give yourself a big pat on the back*!)

2 Developing an interface for lexically casting arbitrary types

In this section we develop a basic interface for performing lexical casts. Our requirements are:

- It must support all types which meet the type requirements listed above.
- It must be client-agnostic. That is, it must not be geared towards usage in any single environment, but should be generic enough to serve a number of different clients' needs.

Let's start...

2.1 What C++ provides us with already

Believe it or not, the STL provides us with everything we need to implement lexical casts in C++. If you don't believe that, consider the following snippet:

```
std::ostringstream os;
int myint = 7;
os << myint; // effectively "casts" 7 to a string
std::istringstream is (os.str() );
std::string mystr;
is >> mystr; // mystr == "7"
```

The end result is that `myint` equals 7 and `mystr` equals "7". That is, we have just lexically cast the number 7 to the string "7".

We're going to disregard, for the time being, the complications introduced by whitespace characters and the accuracy of floating-point vis-a-vis this approach. We will cover those in more detail later on.

The above probably shows you nothing you didn't already know. Assuming one is familiar with the stream-related parts of the STL, there is nothing outright mysterious about the `stringstream` classes. However, writing the above code over and over in an application would get *really* tedious *really* quickly. As one of the most prevalnt traits of programmers - common wisdom says - is *laziness*, we can assume that most programmers don't want to do tedious work, and that includes writing and re-writing code like that shown above. Let's now explore how we might shorten the above code to an amount more suitable to... well, to our utter laziness.

2.2 An interface and implementation

Our first step will be to create a pair of functions which act as a convenience interface to using `stringstream`s.

One immediate problem surfaces: how do we (generically) handle conversion failures? For example, if we try to cast the string "oh my" to a `double`, what should happen if it fails? Even more importantly, *how do we know* if such a conversion fails?

This second question is trickier than it sounds, and we're going to jump right into the answer without going into detail about why it is so deceptively tricky:

> ***We can't know with 100% certainty.***

That sounds a bit harsh, so i feel compelled to qualify that with:

At least, not from a generic interface.

Even with that qualification, that might not be 100% technically true, but i will continue to write as if it is true. :)

Let's consider this example:

```
std::string s = "oh my";
std::istringstream is( s );
double d;
is >> d;
```

Before you accuse me of lying when i assert we *cannot* know if the conversion failed, let's see what we *can* check:

```
if( !  is.good() ) { ...  error ...  }
```

To be honest, though, this tells us very little. It tells us that there was a stream-level error, which can theoretically be caused by *any number* of stream-related problems, like a device-level failure in the middle of a read. Remember that the streams we're using need not be `stringstream`s - they might be `ifstream`s. (True enough: we *are* explicitely dealing with `stringstream`s here, which are *highly* unlikely to fail during i/o, but let's think generically for the time being and assume that we are reading from and writing to arbitrary streams which *might* fail.)

So, for now i will continue to assert that, for all practical purposes, we cannot know if such a conversion fails.

"But wait" you say, "what about this:"

```
if( 0.0 == d ) { ...  error ...  }
```

Nice try, but no prize: 0.0 might be a valid value for any given conversion from a string to a `double`.

Before going any further, let's stress this point:

> There is *nothing* magical about *any specific value of a given object of type T* which makes it useful as a general-purpose "error" value for *any and all* conversions of type T to or from strings.

(Here we will not consider options such as stream operators throwing exceptions on errors. While realistic, this option opens up a *whole* other can of worms, the discussion of which is *well* beyond the scope of this document.)

Don't get hung up on the idea that we cannot check for casting errors, though, because there *is* actually a fool-proof way to *generically* determine when a lexical cast fails. We will see that approach in a moment, once we have laid out our basic convenience interfaces.

Without any further ado, here we will present one possible back-end for lexical casting. Let's start with the simpler of the two: conversion of non-strings to strings:

```
template <typename value_type>
std::string to_string( const value_type & obj )
{
    std::ostringstream os;
    os << std::fixed; // Very arguable!  Discussed later on.
    os << obj;
    return os.str();
}
```

That should be pretty straightforward: the function lexically casts `obj` to its string representation. For most purposes we can assume that this function essentially "can't fail". That is, an `ostringstream` is essentially a container, and container-level insertions don't, for practical purposes, fail.

The converse, casting from a string to a non-string, is a bit trickier because of error handling. Here we will see one possible implementation, which has actually worked very well for me over the past couple of years:

```
template <typename value_type>
value_type from_string( const std::string & str, const value_type & errorVal )
{
    std::istringstream is( str );
    if ( !is ) return errorVal;
    value_type foo = value_type();
    if ( is >> foo ) return foo;
    return errorVal;
}
```

Again, this should be pretty straightforward, but let's take a closer look at the second parameter that `from_string()` accepts. Let's consider the following call:

```
double d = from_string( "17.3", 0.1 );
```

If the above function fails then the passed-in "error value", 0.1, is returned, otherwise the properly-cast value of 17.3 will be returned. This has several implications:

1. The *client* specifies what he considers to be an error value. What if 0.1 is a valid value? More specifically (and more likely), what if there is no known "invalid" value for the type? We'll answer that in a moment, as the answer is a bit awkward (but fail-safe).
2. The "error value" may also be interpretted as a "default value". (This is almost always how my applications interpret it.) For example, passing "a string" to `from_string()` is not going to return a `double`, and applications can use the second parameter to mean, "if `str` is not really a `double` then let's use the default value of 0.1."
3. Explicitely passing a second argument ensures that clients do not have to type `from_string<double>(...)`, as standard argument-type deduction can do this job (most of the time, anyway - there *are* ambiguous cases where the client must specify the type).

Before continuing, let's answer the first point's question:

To solve this problem we simply need to make the following observation about lexical casts:

> A lexical cast failure *will always fail in the same way*. Or, more correctly, it *cannot* succeed twice in *different ways*.

Why is this so? Let's take a close look:

If we try to convert a given string to a another type *twice*, and it converts *differently* both times, then we know that one or both of the conversions failed. Thus, we can deduce that the conversion as a whole failed. i can almost year you say, "*what the foo is he going on about?!?!*" Let's show an example, which should make it clear:

```
std::string dstr = "not a number";
double d = from_string( dstr, 0.0 ); // d == 0.0
if( 0.0 == d ) {
    d = from_string( dstr, 1.0 ) // d == 1.0
    if( 1.0 == d ) {
        // A real error, as deduction tells us that dstr
        // could not possibly have been successfully cast
        // to both 0.0 and 1.0.
    }
}
```

We know that, according to the rules of reading a `double` from an `istream`, lexically casting `dstr` to `d` will fail. This means that we will get back the second `from_string()` parameter as our return result for *both* calls to `from_string()`. We also know that if `dstr` fails to cast to a `double`, it will also fail to cast to *any other* `double`. Thus we can check against *two* cases, and if the cast returns our different "error values" twice then we know that the failure is "real", and that `dstr` was not a `double` (in string format) to begin with. If, on the other hand, the first call succeeds and the second call fails, then we know (through deduction) that 0.0 was the actual value stored in `dstr`.

Got that?

Note that while the above example uses `double`s, deduction implies that this property holds true for all lexically-castable types.

Is that a tedious way to check for an error? *You betchya!* However:

- My own use of this interface has been such that i generally interpret "error value" as "default value". For example, "if option FOO is not set in the application's configuration data, or is set to an invalid value, let's use some default, known-good value instead." This has served my purposes well for 99% of use cases.
- i believe this approach to be client-agnostic enough to be useful for a wide variety of cases.
- Imposing exception-handling conventions for a failure is not only disconcerting, but would be, in my opinion, *downright WRONG*, as the `to`/`from_string()` functions are too generic to *really* know what an error is, and therefore cannot claim to know when they should or should not throw an exception.
- i believe double-checking to be only *100% reliable* way to generically check for a string-to-T conversion failure. (Remember that T-to-string conversions essentially cannot fail.)

That concludes the introduction to our implementation, with one minor exception...

2.3 The obligatory kludge

Before we wrap up our interface and call it "frozen", we will throw in a couple of quasi-bogus (i.e., arguable) additions. The functions shown above have a problem with lexically casting *strings* to or from *strings*. Why would we want to do that? Ideally, we wouldn't, but our interface is designed such that it doesn't really care *what* type we pass to it, as long as a stream can be used to convert the value. So, to ease client-side use *and* to keep us from unduly (and arbitrarily) having to modify stream flags (like `ios_base::skipws`), we will provide the following overloads:

```
std::string from_string( const std::string & str, const std::string & ) { return str;
}
std::string from_string( const char *str, const char * ) { return str; }
std::string to_string( const char *str ) { return str; }
std::string to_string( const std::string & str ) { return str; }
```

The reasons for and implications of these additions should be apparent, so we won't dwell on them here.

It might be interesting to note that we removed the argument *names* from the second argument to `from_string()`. The reasoning is three-fold: first off, some compilers will complain about unused named arguments. Secondly, we do this to stress that those arguments are not used by those functions: they exist only for compatibility with the basic interface, to keep clients from having to know when they're actually casting a string to a string ("*What!?!?*" Yes, this can and does happen in generic algorithms). Lastly, these additions to the API have proven to be useful in generic algorithms which use these functions for, e.g., converting a `std::list<X>` to or from a `std::list<std::string>` (e.g., for serialization purposes).

2.4 Sample usage

Now that we have an implementation for handling lexical casts, let's see what it looks like in client code. Let's first look at `from_string()`:

```
std::string s = "17";
double d = from_string( s, 0.0 ); // d == 17.0
std::string s2 = from_string( s, "doh!" ); // s2 == "17"'
int i = from_string<int>( s, 0 ); // i == 17.  See below.
bool b = from_string<bool>( s, false ); // b == true.  See below
std::string longstr = "this is a long string";
double d2 = from_string( longstr, 0.0 ); // d2 == 0.0
std::string s3 = from_string( longstr, std::string() ); // == "this is a long string"
```

In the `int` and `bool` conversions we explicitely provide the templatized type to avoid an ambiguity, because 0 is a valid value for a variety of built-in types, such as `int`, `bool`, and `char`. In other words, it is not practical to expect the compiler to inherently know what *type* the number 0 should represent, so we help it out here by being explicit.

Now let's take a look at using `to_string()`:

```
int i = 7;
std::string si = to_string<int>( i ); // si == "7"
// ^^^^ <int> may or may not be required here, depending on the compiler
double d = 7.7;
std::string sd = to_string( d ); // sd == "7.7"
// ^^^^ again, <double> may or may not be required.
```

So far, so good.

We can see here that checking for a failure in `to_string()` is easy, compared to `from_string()`: such operations essentially never fail. But keep in mind: while that property holds for C++'s built-in types, it *may or may not* apply to arbitrary user-defined types!

2.5 Eeek! A bug!

There is actually one glaring problem in this implementation: casting floating-point values, such as `doubles`, may not do what is expected. Floating-point numbers are not 100% accurately handled by the above code once the precision gets beyond six digits (or so - the accuracy is almost certainly compiler- or STL-implementation dependent).

Generically solving this problem is left as a dreaded *exercise for the reader* (i've always wanted to write that).

2.6 Conclusion

We have written a generic interface for lexically casting and have shown that it does what we set out to do. This interface is, in and of itself, useful in a variety of cases, and i have personally used it in several projects for a couple of years. Thus, we won't harp on the point that it works, nor will we dwell on the cases where it is (or is not) useful. Despite its generic utility, this interface can get slightly tedious to use at times. The rest of this paper will be spent trying to hide this interface from client code, wrapping it in a more generic, easier-to-use interface.

Now go re-fill your coffee (or your tea, if that's your thing. Or your beer.) and then let's continue our search for a more appealing client-side interface to lexical casting

3 `lex_t`: a "lexical type"

In the previous sections we developed our basic interface for lexical casting. In this section we will develop a class which simplifies the process of adding lexical casting to our client-side code.

Every class needs a name before it can be coded, so we'll go ahead and make that decision up front. Because the class is intended to lexically cast types to and from strings, we will call it a "lexical type", or `lex_t` for short. Another good name might be `variant_t`, but we'll go ahead and stick with `lex_t`, primarily because i already have so much code with that name in it that i have a hard time changing the name ;).

The requirements for our type are essentially the same as for the basic `to/from_string()` interface, with these additions:

- We must be able to assign values of lexically castable types to a `lex_t`. e.g., the following must work as expected:
  `lex_t lex = 27;`
- We must be able to assign `lex_t` objects to variables of supported types. e.g., the following must work as expected:
  `int foo = lex;`
- It must provide a way for the user to explicitely state the type to cast *to*, to get around ambiguous cases (as shown earlier). Note that for casting *from* other types we do not need an "ambiguity buster", as we already have C++'s various cast operators to do this for us.

3.1 Backing up: class-defined, implicit type conversions

Let's briefly cover one of C++'s features: the ability to define arbitrary type conversion operators for a class. An an example, assume we have the following member functions in a class:

```
operator bool() const { ...  }
operator double() const { ...  }
```

There a number of very valid reasons to *not* include such conversions in a class, *especially* conversion for for `bool`, but we won't go into detail about them here. What we will say is: for what we are about to do, this type of conversion is *exactly what we want*. Thus we will shamefully abuse this feature. What we *won't* do is write such an operator for every type we want to convert to. What we *will* do is take advantage of C++'s template facilities and use the following single function (you may want to sit down first...):

```
template <typename T>
operator T() const { ...  }
```

Doh!

The implications of that function are pretty far-reaching, and it is *not* recommended for general use.

To be honest, until a few days before writing this paper, i didn't know C++ would let us get away with this. It does, or at least my compiler allows it.

Now there are people out there (i'm one of them) who will shudder in discomfort when seeing the above code. Consider: it tells the compiler that our type can be implicitely converted to *any other type*. Dangerous? Definately so. We will console ourselves with the knowledge that the interface is documented well enough, and is easy enough to use, that such conversions "shouldn't" Cause Grief, at least not in "common sense" cases. Let's not dwell on the downright underhandedness of the above code, and try to continue without letting that weigh on our minds too much.

Aside from conversion *from* a given type, we need to be able to convert *to* a given type, an operation we can summarize with one function:

```
template <typename T>
lex_t & operator=( const T & ) { ...  }
```

Again, this is potentially a spawning ground for Grief, but we will accept this possibility for the reasons given above.

Given the above two functions, we now have the majority of what we need to wrap `to/from_string()` in a class interface. We'll need to clean it up a bit, so let's get going. Before we do, though, let us stress the following point:

> The above type conversion operators are ***not at all*** well-suited for general purpose use in arbitrary classes! *Please* be aware that by relying on implicit conversions of *any* type (no pun intended), you are leaving yourself open to seeing some, shall we say, "surprising" behaviour in your software!

3.2 Class interface

We'll jump the gun a bit here and take a look at a class which should achieve our design requirements. Afterwards we will discuss some of the implications of the interface.

Because the implementations is so small and straightforward, we will go ahead and show the whole class in one sitting, as opposed to breaking it down into example-sized chunks. Don't dwell too long on the implementation details here: focus only on the interface. We could probably debate the merits and non-merits of the implementation all day long and never reach an all-around satisfactory agreement. (That said, readers who have definate ideas about improvements are encouraged to get in touch with me!)

Our class is pasted in below, reformatted a bit for presentation here. The calls to our previously-written `to/from_string()` functions are highlighted in blue.

```
class lex_t {
private:
    std::string m_data; // stores our raw data
public:
    lex_t(){}
    ~lex_t() {}
    // Standard copy ctor:
    lex_t( const lex_t & rhs ) {
        this->m_data = rhs.m_data;
    }
    // An efficiency overload:
    lex_t( const std::string & v ) :  m_data(v) {}
    // Standard assignment operator:
    inline lex_t & operator=( const lex_t & rhs ) {
        if( &rhs != this ) this->m_data = rhs.m_data;
        return *this;
    }
    // Generic implicit conversion ctor:
    template <typename FromT>
    lex_t( const FromT & f ) :  m_data(to_string( f )) {}
    // Casts this object's value to a ToType, returning dflt
    // if the conversion fails:
    template <typename ToType>
    ToType cast_to( const ToType & dflt = ToType() ) const {
        return from_string( this->m_data, dflt );
    }
    // Provide implicit conversions for lex_t objects
    // in rvalue contexts (i.e., on the right-hand side
    // of an expression):
    template <typename ToType>
    inline operator ToType() const {
```

```
        return this->cast_to( ToType() );
    }
    // Provide implicit conversion for lex_t objects
    // used in lvalue contexts (i.e., being assigned to).
    template <typename ToType>
    inline lex_t & operator=( const ToType & f ) {
        this->m_data = to_string( f );
        return *this;
    }
    // return the raw data as a string:
    inline std::string & str() { return this->m_data; }
    inline const std::string & str() const { return this->m_data; }
    // Implement operator< so we can use this type in
    // std::map<> and the like:
    inline bool operator<( const lex_t & rhs ) const {
        return this->str() < rhs.str();
    }
    inline bool operator>( const lex_t & rhs ) const {
        return this->str() > rhs.str();
    }
    inline bool operator==( const lex_t & rhs ) const {
        return this->str() == rhs.str();
    }
    // An efficiency overload:
    inline operator std::string () const { return this->str(); }
    // Another efficiency overload:
    inline operator const char * () const { return this->str().c_str(); }
}; // end lex_t class
```

Notice how simple the overall class implementation is: none of the functions are longer than two lines of code (well, three if we count "`if (...)`" as a line by itself).

Because `lex_t` is essentially a `std::string` proxy, implementing the ostream operator is trivial:

```
// Sends lt.str() to the given ostream:
inline std::ostream & operator<<( std::ostream & os, const lex_t & lt ) {
    return os << lt.str();
}
```

Interestingly, it is not so straightforward for istream:

```
// Populates lt from the given istream.
// Note that this implementation seems to be extremely dubious,
// but actually does exactly what we need, as discussed below.
inline std::istream &
operator>>( std::istream & is, lex_t & lt ) {
    /****
    Attempt #1:
        is >> lt.str(); // depends on skipws.
```

```
    ****/
    /****
    Attempt #2:
        while( std::getline( is, lt.str() ).good() );
    Eeek!  strips newlines!
    ****/
    /****
    Attempt #3:
        char c;
        while( is.get(c).good() ) { lt.str() += c; }
    WTF???  On my box this does nothing!
    ****/
    /****
    Attempt #4:
    std::getline( is, lt.str(), '\v' ); // UGLY, EVIL hack!
    The \v char ("vertical tab") is an ugly hack:  it is simply a char from the ascii chart
    which never shows up in text.  At least, i hope it doesn't.  AFAIK, \v was historically
    used on old line printers and some ancient terminals, but i've never seen it actually
    used.  Unicode maps 0-255 to the ascii set, so this shouldn't be a problem for Unicode
    either.
    This hack is likely to work for most data, but is definately worth -2 Style Points (or
    more).
    *****/
    /****
    Finally, a hack which essentially does what i want:
    (Many thanks to Marc Duerner for this idea.)
    *****/
    return std::getline( is, lt.str(), \
        static_cast<std::istream::char_type>(std::istream::traits_type::eof()) );
}
```

(Side note: the source code available from the URL in section 3.5 might be more up-to-date than that shown here.)

Justifying the istream>> operator:

> In practice, `lex_t` is normally used in containers, and not with file streams. Its istream operator is designed to read in all data, as conventions imply that it is really reading from a stringstream. One partially satisfying solution to the istream-related problem might be to replace the i/ostream operators with operators accepting only stringstreams.

The istream design issues notwithstanding, let's now show how we can use this type, discussing its implications as we go...

3.3 Using `lex_t`

Using `lex_t` in client code is anything but difficult. We use it just like any other type, and allow the compiler to arrange the type conversions for us (or most of them, anyway). Note that we have the `cast_to()` member function to give us explicit control over conversions when we need it.

Here's how it works:

```
lex_t lex = 17;
int i = lex; // i == 17
std::string s = lex; // s == "17"
lex = "a string";
std::cout << "lex="<<lex<<std::endl;
```

This can't be C++ code, can it? It looks too much like Perl!

It is indeed C++. If you think the above looks odd, though, let's take another look at the sneak-preview code we showed at the beginning of this paper:

```
typedef std::map<lex_t,lex_t> MapT;
MapT map;
map[4] = "one";
map["one"] = 4;
map[123] = "eat this";
map["123"] = "this was re-set";
map['x'] = "marks the spot";
map["fred"] = 94.3 * static_cast<double>( map["one"] );
map["fred"] = 10 * static_cast<double>( map["fred"] );
int myint = map["one"];
lex_t envvar = "USER";
std::cout << envvar <<"="<< ::getenv(envvar) << std::endl;
```

Doh! Now it *really* starts to look like Perl code!

There are cases where implicit type conversions probably won't do what we might expect them to, but clients always have the option of using `lex_t::cast_to()` to force a conversion to a specific type, or can use the standard `static_cast<T>()` when passing an object to, e.g., `lex_t`'s template-based constructor or assignment operator. Those options give us all the flexibility we need for `lex_t`-to/from-`T` conversions, and also provide us with an escape route if an implicit conversion backfires on us somehow (which, given the all-purpose nature of the template-based ctor, assignment operator, and type-conversion operator, is *quite likely* to happen sooner or later!).

Note that we have avoided using any `double`s as map *keys* in the above code. The reason for that is something we mentioned before: the precision of `double`s cannot, as far as i know, be 100% reliably, generically controlled via the interface proposed here (*please correct me if i am mistaken*!). Let's take a look at an example:

```
map["1.0"] = "foo";
map[1.0] = 17;
```

When we iterate over `map` we will probably find that there are two entries instead of one. The stringified key for 1.0 (as a `double`) is probably something like "1.000000", whereas "1.0" (as a string) will be stored literally as "1.0". Thus, using `double`s as keys in such a map is discouraged. Using `double` values is also not recommended if precision is an issue (unless, of course, the reader has accepted the above-mentioned *dreaded exercise* and fixed that problem!).

3.3.1 Shameless plug: s11n

If we use libs11n (`http://s11n.net`), saving and loading the above map is *trivial*:

```
s11nlite::save( map, std::cout ); // pass it your favourite stream or a filename
```

To load it is also trivial:

```
MapT * map = s11nlite::load_serializable<MapT>( instream ); // or pass a filename
```

It *can't* get much simpler than that!

The point is this: if you're wasting your precious coding hours by trying to save your data, *STOP IT*! Serializing data in C++ *absolutely doesn't get any simpler* than it does when using s11n! For examples' sake, the map demonstrated above might end up looking like the following:

```
<!DOCTYPE s11n::simplexml>
<data_node s11n_class="map">
    <pair s11n_class="pair">
        <first s11n_class="lex_t" v="123" />
        <second s11n_class="lex_t" v="this was re-set" />
    </pair>
    <pair s11n_class="pair">
        <first s11n_class="lex_t" v="4" />
        <second s11n_class="lex_t" v="one" />
    </pair>
    <pair s11n_class="pair">
        <first s11n_class="lex_t" v="fred" />
        <second s11n_class="lex_t" v="3772.000000" />
    </pair>
    <pair s11n_class="pair">
        <first s11n_class="lex_t" v="one" />
        <second s11n_class="lex_t" v="4" />
    </pair>
    <pair s11n_class="pair">
        <first s11n_class="lex_t" v="x" />
        <second s11n_class="lex_t" v="marks the spot" />
    </pair>
</data_node>
```

The exact data format is unimportant: s11n is data-format agnostic and currently (as of this writing, version 1.0.0) supports 7(!!!) different data flavours.

Utilizing libs11n, saving and loading such types is *child's play*: adding such support to an application is a matter of a couple minutes of work, as opposed to several hours (or even days) of work.

3.4 Potential uses

Now that we have `lex_t`, what are we going to do with it? Here is a partial list of potential uses:

- Abstract symbol tables. The demonstrated map, e.g., would be suitable for a simple symbol table.
- Fetching data from or updating data in database record objects.
- Converting numeric data entered via, e.g., a string-based UI widget, or a command-line or console-style interface. One example which immediately comes to mind is for easily converting arguments passed in to an application's `main()` function:
  `int foo = lex_t(argv[i]);`
- Conversely, converting numeric information to strings for insertion into a widget which requires strings.
- Converting lexer-parsed strings to various types. (That is, "lex" as in the classic text-parsing tool, not as in "lexical type.")
- Use a `std::map`, as shown above, to store application configuration information.

3.5 Source code for `lex_t`

The latest "official" source code for the `lex_t` type can be obtained via the s11n web site:

`http://s11n.net/download/#lex_t`

4 Conclusion

If this paper has greatly offended your sense of strongly-typed code by presenting this paper, be consoled by the fact that `EOF` is very near...

We've covered quite a bit of ground here. (i set out to write 4 or 5 pages, and suddenly it's well over 10 :/.) We have learned what lexical casting is. We have shown that lexical casting works sufficiently well for a number of use cases. We have shown two possible approaches to adding it to your C++ toolbox. And, finally, we touched on ideas for some potential uses.

Now, if your gut is not still wrenching from the Perl-ness of the above example code, *go and give it a try!* Creative coders will certainly find uses for lexical casting in their C++ projects!

Thanks for taking the time to read this paper. i sincerely hope that you have learned something from it, or have at least enjoyed reading it.

Any bug fixes or enhancements to the `lex_t` source code or API documentation, or for this document, are of course welcomed, as is feedback of *any constructive sort* (regardless of whether it is *positive* or not). Blatant flame-mails will be lexically cast to a `double`, which, as we have seen here, will result in useless data. :)

Happy hacking,

stephan beal (stephan@s11n.net)

19 August, 2004