Types++: Typesafe Metadata, and Other Thoughts Beyond int

Ryan Wilcox
Wilcox Development Solutions
rwilcox@wilcoxd.com

Abstract

C/C++ (and other strongly, statically typed languages) use type data as a compile-time check: you can't put that there, or the ever-popular "some data might be truncated" warning. Still there is possibility for error: can you know for certain, just by looking at a function prototype, that the Str255 parameter is a PString, or a shortcut by a lazy programmer? Or can you ever be certain that your divisor (int) is not zero, without checks all over? What character encoding does that std::string contain? In essence: metadata, not just type, is an important factor of many variables. Metadata focus can be achieved by using a concept called meta-types, which can supply compile-time checks, runtime checks, and serve as form of documentation for future maintainers. This paper will further discuss metadata, meta-type classes (implementation, usage, examples, and drawbacks), and other thoughts for modern "type" safety.

Introduction

This paper presents ideas on using C++'s template system as a compile-time check on the *contents*, or the value of a specific attribute, of a variable. This paper is focused around certain areas of C++, notably the use of simple template types, however some of these ideas could be applied to other languages, like Java generics, with possibly less useful results. Some of these concepts may apply to dynamically typed languages also, but it really depends on how and when/if your language performs type checking (and how strong the type identity is.)

About the Examples, Sample Code, and Mechanics

Sample code should be included with the (binary) distributions of the paper. Where this paper includes a source example, a file name is included (and underlined) at the beginning of the code block. The source file will include a use scenario, and possibly an expanded version of the code example presented in the body of this paper. These source files were tested on GCC (4.0.0 20041026 by Apple Computer, Inc), and should compile (unless the source file and error message were illustrating a particular topic.)

A Makefile is also included. Users unfamiliar with GCC, or want to ensure consistent flag use, are invited to use `make`. Targets for the makefile are simply the name of the file to compile, without extensions. So, to compile subclassVBaseClass.cpp, use the following make command:

```
$ make subclassVBaseClass
```

Make echoes every command that it executes back to the terminal, and readers are invited to use `make`'s `-n` parameter (which will echo the command, but will not execute it), to note the parameters passed to gcc.

```
$ make –n subclassVBaseClass
```

A brief introduction of the C++ type system

First a bit of background: The C++ type system could be divided into two parts, fundamental types and user defined objects. We'll discuss how C++'s type system affects both kinds of data in the following paragraphs.

Fundamental types

Fundamental types are the standard C defined types: `char`, `int`, short, `long`, float, `double`. Now, you could think of these types as describing the data inside, for example, `char` might contains values from -127 to 127. One of the axioms of C is type-safety: you can't put a `long` into a `short` without the compiler telling you about it (or at least it should.) Here the compiler is to protecting you from a silly error - trying to put a larger value (say a 4 byte value), into a smaller one (say a 2 byte value.) Such an operation could lead to an unexpected loss of data, or an overflow error.

This system, while it seems reasonable, only warns on these types of errors, and by default warnings don't stop the compilation flow. However, problems such as these could lead to unexpected side effects, and some compilers (like GCC) do not generate many warnings by default. C's struct mechanism prevents unintentional overflow errors, and provides more strict type checking, but can be easily subverted by accessing and assigning data inside the struct directly.

The ability to access members of a structure directly lets other programmers subvert any value checking we might have done in accessor methods. The techniques presented in this paper, while possible to implement in C, requires a guarantee that the data inside the structure will be what the programmer expects – you wouldn't want a zero creeping into a data structure that you thought would be AnyNumberButZero. If you can subvert this guarantee, the idea of compiler-level metadata, while a clever documentation idea, ultimately fails to be more than that.

Object types

C++ brought the ideas of OO programming to C. You could couple your member data with your implementation inside a class, and create functions that worked with that data. C++ also allows you to partition off parts of your class with members (methods and data) that users of your class can use, and members that users of your class can't access.

Life is good in C++ land: everybody has their own Employee classes, Shape classes, and whatever else they need, with public, protected, and private access controls. The world is also type-safe, for the compiler will not let you convert an Employee into a Shape unless you tell C++ how.

The Problem

There's a problem with this system in that it tells you all about the variable, but nothing about the data inside. For example, a function accepts a `const char*.` Is that `const char*` null terminated? Is it a specific length? Or is there a Pascal string hidden in there? There's really no way to tell – nothing in the language (or type) prevents any of these (semi) deviant behaviors.

You could "make sure" you get what you expect by documenting your functions, but this method comes with a catch: sooner or later someone won't read the documentation. Imagine a colleague passing a Pascal string into a function, which accepts a C string, crash, and spending 10 or 15 minutes debugging until they realize that the function takes a C string. This certainly doesn't sound like a lot, but it's a simple example. The real world is much harsher, more complex, and takes longer to debug.

Of course, the techniques presented in this paper are not a replacement for well-documented code. For documenting functions this author prefers HeaderDoc

(http://developer.apple.com/darwin/projects/headerdoc/), installed with the Developer Tools on OS X, but other systems of documentation, like Doxygen (http:// www.doxygen.org), exist for those who prefer differing documenting systems.

The Pascal string vs. C string issue is mildly interesting for the Carbon programmer who occasionally has to deal with such things, but let's imagine a more modern one: multilingual application. Even in a Unicode world you have different ways to represent characters in a `std::wstring` – UTF-8, UTF-16, and even a barrage of pre-Unicode character encodings. How do you keep all these formats straight in your program? Even being very careful, one can mix up function parameters and character encodings.

Number values would be another example: perhaps your function takes only positive numbers, numbers in a certain range, or percentages. A C base type seems to fit perfectly in these cases, but how do we ensure we get the kind of data our function expects, a way to ensure that What Your Function Asked for is What It Got (WYFAWIG - "way faw whig")?

In all of these examples the C++ type (`const char*`, `std::wstring`, and `long`) isn't enough to provide helpful information about the given parameter, the ever popular question of "how exactly do you want that data?" comes to mind. In the best-case scenario, users of the function simply read the documentation to find out… and in the worse case, the original author wrote the function, forgot to document it, and left the company. There's *got* to be a way to avoid this mess, and know with 100% confidence that you are getting the data you expect!

My Solution: Coupling Metadata with types (Meta-types)

The C++ type system enforces strong, static, typesafety at compile time. If you try to jam a Circle into a Square slot, C++ will complain that you can't do that (unless, of course, you provided copy constructors explicitly to do this.) Errors found at compile time force developers to go back and take another look at their code, instead of getting unexpected results at runtime. Or, to say it another way, "An error that you see now is better than one you see later."

Here we reach the central idea of our paper: creating small types that accentuate an attribute of the data inside the class. Perhaps this attribute is color, perhaps that the data matches a specific query, or perhaps simply the format of the data inside the class (our std::wstring example again.) Perhaps the class is a conversion class, taking input data, converting it to a specific format, and storing it inside the class itself, so that everybody knows what kind of data the object will contain.

Since they are classes, the C++ compiler enforces static typechecking on these small types, and throws a compiler error if the passed type does not match the expected type. This compile level error forces users of your class to use your meta-data accenting types (or: meta-types), ensuring that What Your Function Asked For is What It Got

Derived MetaTypes

How do we enforce WYFAWIG in our functions? One idea is to create a class derived from a common base class. The derived class would validate, or explicitly set, some member data from the base class. The following example shows a `RedCircle`, which is derived from `Circle`.

subclassVBaseClass.cpp

```
class Shape
{
      public:
            Shape() {;}
```

```
            Shape(long color) m_color(color) {;}
      private: long m_color;

};

class Circle : public Shape
{
      public : Circle(long color): Shape(color);
};

class RedCircle : public Circle
{
      public: RedCircle() : Circle(kRed);
};

class ScreenCanvas
{
      public:
            void DrawCircle (Circle& toDraw);
            void DrawRedCircle (RedCircle& toDraw);
};
```

It is valid to pass a `RedCircle` instance to the `ScreenCanvas::DrawCircle`(), but invalid to pass a `Circle` to `ScreenCanvas::DrawRedCircle`(). This way of creating compiler-aware meta-types works very well, as long you are willing (or can) subclass the base object, but fails at two critical junctures. First, it is not generic, in that `RedCircles`, `RedTriangles`, and `RedSquares`, require whole separate classes. It also requires a base object to derive from, which may or may not be the case (and, with fundamental types, *won't* be.) Sometimes sub-classing objects is not fun (especially STL classes), and it's best to find other ways to create metadata type-safety. However, a major advantage to this approach is that subclass instances can be used in place instances of the base class, and functions of the base class can be accessed easily through normal C++ function calling mechanisms.

Template Meta-types

C++'s template feature presents several unique opportunities for compiler aware metadata, and removes some of the issues with meta-types implemented via derived classes. For example, a `Red` template could be created, and hold `Circles`, `Triangles`, and `Squares`, without compromising on type-safety, as a `Red<Square>` can't be used in a function that accepts a `Red<Circle>`.

metadataTemplates.cpp

```
template <typename T>
class Red
{
      public:     Red() : m_elem(kRed) {;}
      private: T m_elem;
};

class Shape
{
      public:
            Shape();
            Shape(long color);
      private: long m_color;
};
```

```
class Circle : public Shape
{
      public : Circle(long color) : Shape(color) {;}
};

class ScreenCanvas
{
      public:
            void DrawCircle(Circle& toDraw);
            void DrawRedCircle(Red<Circle>& toDraw);
};
```

`ScreenCanvas::DrawRedCircle` only accepts `Red<Circles>`, not `Red<Square>`s, nor generic `Circles`. Both the template and base class examples show WYFAWIG in action, you know you will always get a `Red<Circle>` because the compiler forces your fellow programmers to always give you one.

Showdown: Derived vs. Template meta-types

Templates are generic structures that can be used with any class: `Red<T>` can apply to anything: shapes, cars, shirts, pens, and countless others, without any additional code. The one drawback is that template meta-types require additional programming to access arbitrary public methods of the instance of T.

Derived meta-types, on the other hand, have easy access to public methods of the base class, while at the disadvantage of having to create a new class for each derived object. Derived meta-types pose one significant advantage over template meta-types, in that one can use factory methods to create instances of the object. The ability to use factory methods to create instances of objects is a requirement if you are serializing an instance for transport over the network or even for storage on a disk.

Advantages

The Meta-type approach has numerous advantages, including WYFAWIG, the C++ type system working for you, users of your code are forced to think, and it allows a level of type-logic abstraction to be created.

(Knowing) What Your Function Asked for is What It Got

Functions that accept meta-type parameters can be certain that data in the variable is what the function expects; because the meta-type objects do the conversion and validation work and they guarantee that the data is correct.

One approach in creating Meta-types is to provide no copy-constructors for the type. For example, neither our `Red<T>` template nor `RedCircle` has a copy constructor for `T` and `Circle` (respectively) - a user must create a `Red` object based off no other data. Later in this paper we'll revise `Red<T>`, and allow it to accept (and validate) arbitrary `Color` objects passed to it in via a copy constructor.

Copy constructors work well if you only have one type of input data, but will invite ambiguity when applied to multiple types of data – especially if the different kinds of data fit in the same C++ type. `UTF8String`, also presented later in the paper, is an example of this exact scenario: the contents of a string can be one of any number of possible of character encodings. Sometimes separate functions have to be used to avoid this ambiguity, and the class sacrifices the instantiation-is-initialization paradigm for explicitness, clarity, and the idea that the programmer on the outside of this function knows what kind of data they started off with.

The C++ Type System Works For You

C++'s compile-time enforced type-safety is a great stopping mechanism – you have to fix these errors before going forward. Sometimes, however, it feels like the type system is fighting you – const vs. mutable, using exactly the right type, etc. The Meta-types concept is implemented using C++ static typing, which forces users to use the correct object in the correct place, while your meta-type enforces data sanity. In this way the compile-time type-safety is ensuring the safety of your data, doing something truly useful. You could certainly use the meta-type concept in other strongly typed languages, creating types based on the important data inside the object, with varying results.

Type-logic abstracted

What does this mean? Simply that any data attribute tests, or data attribute conversions, are done on creation of the object (and not multiple times.)

Reviewing the `Red<T>` template, we see that there's no way to feed it existing `T` instances. We need to create a copy constructor for `T`, which validates that the input shape really is red; if the shape is not we must throw an exception to prevent further use of the invalid not-red `Red` object. Validation only happens inside `Red<T>`, all other code can use `Red<T>` knowing that the object inside, whatever that is, must be red. Better to do a data validation once, on creation of a meta-type instance, than many times (as might happen in programs that don't use metadata proactively.) This revised `Red<T>` is presented in the following section.

More About Meta-types

We've seen two distinct categories of Meta-types. The first was `Red<Circle>/RedCircle`, both of which created a `Circle` object that was guaranteed to be red, all the operations are performed at initialization time, and no conversion was necessary, and there was no ambiguity about input data.

It would be helpful if `Red<T>` had a copy constructor, so we can feed it pre-existing variables. This illustrates a form of another group of Meta-types: those that provide validation and conversion operators on all incoming data. Such a meta-type should refuse data that doesn't validate (or convert) to the expected value – the author suggests throwing an appropriate exception on construction or during the conversion function. However, a word must be said about exception safety here: it is important to ensure the validity of the member data. Always look before you leap, and check the data before you assign it to the member variable (create a temporary variable if you have to.) See Appendix E in [Stroustrup2000], and a large section of [Sutter2000] for far more detail on exception safety than I could put in this paper.

metadataTemplatesCopyConstruct.cpp

```
template <typename T>
class Red
{
      public:
            Red() : m_elem(kRed) {;}
            Red(T& copied)          //throws std::invalid_argument
            {
                  if (copied.GetColor() != kRed )
                        throw std::invalid_argument("color must be red!");
                  m_elem =  copied ;      //data good, copy it

            }
      private: T m_elem;
};

class Shape
{
```

```
	public:
		Shape() {;}
		Shape(long color) : m_color(color) {;}
		Shape(Shape& copy)
		{
			SetColor( copy.GetColor() );
		}

		long GetColor() {return (m_color);}
		void SetColor(long color) {m_color = color;}

	private:
		long m_color;
};
```

Samples and Use Cases

Sample: AnyNumberButZero

`AnyNumberButZero` is a useful meta-type template when combined with functions that divide numbers. Your `AnyNumberButZero` instance would be used as the divisor, and your dividing code would not have to worry about dividing by zero, because `AnyNumberButZero` validates the contained value.

AnyNumberButZero.cpp

```
template <typename T>
class AnyNumberButZero
{
	public:
	AnyNumberButZero(T num) //throws std::out_of_range
	{
		if (num)
			m_elem = num;
		else
			throw std::out_of_range("must be non-zero!");
	}
	T Get() { return m_elem; }

	private:
	T m_elem;
};
```

Sample: UTF8String

UTF8String is a meta-type that provides conversion between different encodings and UTF-8. String encodings are an ideal use-case for meta-types, especially pre-Unicode encodings, because so many different encoding types fit in a string type, and "forgetting" what encoding a string is can bring up any number of visual bugs when your software is used internationally.

This implementation of UTF8String also illustrates an important point about meta-types: it is often easy to go overboard with meta-types, implementing a class that can do everything. This is not strictly necessarily –implementing a class in a minimalist fashion, but with an extensible framework, is often the most optimal design.

UTF8String.cpp

```
template <typename STRINGTYPE>
class UTF8String
{
	public:
		UTF8String() {;}

		void SetFromUTF8(UTF8String<STRINGTYPE>& utf8Data)
		{ m_elem = utf8Data.Get(); }

		void SetFromPStringMacRoman(Str255 pStr)  //throws OSStatus
		{
			OSStatus err;

			TextEncoding utf8Enc  =
				CreateTextEncoding(kTextEncodingUnicodeDefault,
					kUnicodeNoSubset, kUnicodeUTF8Format);

			TECObjectRef objRef;

			err = TECCreateConverter(&objRef,
				kTextEncodingMacRoman, utf8Enc);

			if (err) throw (err);

			try
			{
				ByteCount actualILen;
				char str255cStr[255];
				ByteCount outLen;

				err = TECConvertText(  objRef, pStr+1, pStr[0],
					&actualILen, TextPtr(str255cStr), 255,
					&outLen  );

				if (err) throw (err);

				m_elem.assign(str255cStr, outLen);
			}
			catch (OSStatus& err)
			{
				TECDisposeConverter(objRef);
				throw;
				//throw the same exception again
				//(after we dispose our TECConverter)
			}
			TECDisposeConverter(objRef);
		}

		STRINGTYPE GetUTF8EncodedStr() { return (m_elem); }

	private:
		STRINGTYPE m_elem;

};
```

A quick comment about the sample is necessary here: the UTF8String.cpp program reads input from standard-in, and sends output to standard-out. The standard-in file must be MacRoman encoded file, and

the resulting file will be UTF-8 without a BOM. Opening the output file in BBEdit may require you to Reopen Using Encoding, (UTF-8 no BOM) in order to display properly.

Specifying both a file for standard-in *and* a file for standard-out, in the same command, is non-obvious in most command-line shells. The following command, however, should work for you:

```
$ program < inputfile > outputfile
```

Concluding Thoughts On Type Safety

Generic Meta-type Parameters

The idea of implementing meta-types with templates leaves an unexplored pattern: functions that accept *any* `Red` object. Functions that don't differentiate between `Red<Circle>`, `Red<Square>` or `Red<Triangle>`, functions that accept any object, as long as it is `Red`. This can easily be accomplished with class member templates. An example, with `ScreenCanvas`, is as follows:

```
class ScreenCanvas
{
        public:
                void DrawCircle(Circle& toDraw) {;}
                void DrawRedCircle(Red<Circle>& toDraw) {;}

                template <typename REDTHING>
                void DrawRedObject(Red<REDTHING>& toDraw) {;}
};
```

Final Explorations and Thoughts

Meta-types present unique opportunities for program correctness, especially when at their simplest. The `Red` meta-type presented through this paper, and the `AnyNumberButZero` meta-type use case, are both extraordinarily simple classes, useful for longer-term variables or even for short-term variables that will fall out of scope quickly. Slightly more expensive classes, for example those that require more involved system calls (like `UTF8String`) may require careful thought, placement to avoid slowdowns – just like any other class in a program.

Meta-types enforce a level of consistency, not just ensuring the class of variable you expect is the class you get, but that attributes of the data (`Red`, `UTF8`, or any other quantifiable attribute) are what you expect as well: that What Your Function Asked For Is What It Got.

Bibliography

[Stroustrup2000] Stroughstrup, Bjarne, The C++ Programming Language: Special Edition. Addison-Wesley Publishing Company, Reading, MA, 2000.

[Sutter2000] Sutter, Herb, Exceptional C++: 47 Engineering Puzzels, Programming Problems, and Solutions, Addison-Wesley Publishing Company, Reading, MA, 2000.