Assignment 1: `Variant`

Due May 28, 11:59 PM

Introduction

Programming languages can be broadly categorized into two groups – *statically-typed languages* and *dynamically-typed languages*. Statically-typed languages are languages like C++ that associates a specific type with each variable. Once a variable has been assigned a type, it maintains that type for the rest of its lifetime; that is, an integer variable will always hold an integer, and a string variable will always hold a string. Dynamically-typed languages, on the other hand, allow the programmer to change the type of variables at will. For example, the following Javascript code creates a variable, then assigns it values of several different types:

```
var myVar = 137;                          // myVar is now an integer.
myVar = "This is a string!";              // myVar is now a string.
myVar = function(x) { return x + 24; }; // myVar is now a function (!).
```

Many of C++'s strengths stem from the fact that it is statically-typed. Because every variable has a unique type, the C++ compiler can determine at compile-time the exact meaning of expressions like `a + b`, which in a dynamically-typed language could correspond to one of many operations depending on the runtime type of `a` and `b` (integer addition, floating-point addition, string concatenation, etc.). Similarly, as you'll see next week, static typing in C++ allows for efficient implementations of virtual functions and inheritance.

However, because C++ is statically-typed, it can be difficult to use C++ to implement software systems that themselves use dynamic typing. For example, suppose we want to write a Javascript interpreter in C++. We would need to create data structures that can store the values in a Javascript program, but those values can change types freely. How can we model such a system using C++'s static type system? Similarly, suppose that we want to author a web browser in C++ and want to parse CSS values to determine how to render web pages. Different CSS properties have different units – for example, the `width` and `height` properties might expect dimensions in pixels, while the `color` and `align` properties might require strings or specially-named constants. How can we build a unified framework for decoding the CSS values if the objects might have multiple different types?

One solution to this problem is to create a special data type known as a *variant type,* an object that can have one of many possible types. For example, a variable with a variant type of "`int` or `string`" can hold either an integer value or a string value. However, the variable could not hold a `double` or a `vector<int>`, since the variant type was not expressly declared to hold either of these types. Note that if the variant holds an integer it does not also hold a string representation of that integer nor vice-versa – the variant always holds exactly one object of one type and does not necessarily permit implicit conversions among them.

Variant types are an elegant solution to the above problems. To represent a Javascript value, we can define a variant type that can take on any possible Javascript type, then use this type to represent Javascript values. Similarly, to implement as CSS parser, we could store the parsed values in a variant type, then decode them appropriately based on the context.*

* The Mozilla Firefox web browser uses an approach similar to this one.

In this assignment, you will learn how to implement a simple variant type in C++ called `Variant`, which can have either an integral or a string value. The approach outlined in this assignment is scalable, so you can easily extend `Variant` to store other types as well. In the process, you'll also learn about another C++ construct called a `union` and will get an excellent chance to practice your newfound C++ language skills.

C++ `unions`

In addition to the traditional `struct` and `class`, C++ has a third type of object called a `union`, a primitive version of a variant type. At first glance, `union`s look like `struct`s or `class`es – they can have data members, constructors, destructors, and member functions. However, `union`s differ from `struct`s and `class`es in one major aspect: *at any given time, of all the data members of a `union`, only one actually exists*. For example, consider the following `union` declaration:

```
union MyUnion
{
    int   myInteger;
    float myFloat;
    char* myCString; // Can't use a C++ string; we'll explain why in a bit.
};
```

This declares a `union` type called `MyUnion` with three data members – an `int`, a `float`, and a `char*` C string. Despite the syntactic similarity to a `struct` or `class`, this declaration does *not* create an object that has three distinct fields. Instead, it creates a type called `MyUnion` that can hold either an `int` *or* a `float` *or* a `char*` C string. To store a value in a union, you can use the familiar member-selection dot operator. For example:

```
MyUnion u;
u.myInteger = 137; // Store 137 in the union
```

Note that to store the value 137 in the `union` we simply assigned the value 137 to the `myInteger` member. We can read back the value we stored in the `union` using similar syntax:

```
cout << u.myInteger << endl; // Prints 137
```

Now, suppose that we now want to store a real number in the `union` instead of an `int`. Then we can simply write

```
u.myFloat = 2.71828;
```

We can read this value back as we did the integer:

```
cout << u.myFloat << endl;
```

The critical difference between a `union` and a traditional `struct` or `class` is the fact that only one of the `union` members actually exists at any one time – the last data member that was written to. If we store a value in a `union` and then try reading the value of a different data member of the union, the result is undefined.* For example, the following code will almost certainly cause a runtime crash because we're

* Usually, `union`s are implemented by having all of the `union`'s data members share the same memory location. If you read back a value other than the last one you wrote in, the value you get back will be based on the binary representation of the last value written.

reading from a field that wasn't the last one written to:

```
MyUnion u;

/* Store a C string in the union. */
u.myCString = StringDuplicate("This is a string!");

/* Store an integer in the union.  This means that the myCString field is
 * no longer valid!
 */
u.myInteger = 137;

/* This line crashes, since myCString has a garbage value. */
cout << u.myCString << endl;
```

Provided that we are careful never to read a field out of a `union` that wasn't the last field written, however, `union`s are perfectly safe.

`union`s are subject to several restrictions that do not apply to traditional `struct`s or `class`es. In particular, `union`s cannot contain classes with nontrivial constructors, destructors, assignment operators, or data members that have nontrivial versions of these functions. Effectively, this means that it is illegal to have C++ classes as data members of a `union`.† For example, the following `union` is illegal:

```
union ErrorUnion
{
    string      myString; // Error: Cannot store a C++ string in a union.
    vector<int> myVector; // Error: Same as above, but for vector<int>.
};
```

While it is illegal to store most classes in a `union`, it is perfectly fine to store *pointers* to classes in a `union`. Thus we could emulate the functionality of the above `union` by writing

```
union LegalUnion
{
    string*      myString; // Legal: Okay to store pointers in a union.
    vector<int>* myVector; // Legal: Same as above.
};
```

and then using dynamic memory allocation to store the elements of the union, as seen here:

```
LegalUnion lu;
lu.myString = new string("Help!  I can't pay my union dues!");
```

Storing dynamically-allocated objects in a `union` can be tricky, since you must be careful to explicitly deallocate the memory before overwriting the pointer. Consider the following code:

```
LegalUnion lu;
lu.myString = new string("Help!  I can't pay my union dues!");
lu.myVector = new vector<int>; // Whoops!  Just leaked memory
```

† Technically speaking some classes can legally be stored in `union`s, but this case is so rare that it's not worth mentioning.

Here, by assigning the `myVector` field of the `union` to point to a `new vector<int>`, we lost the only pointer we had to the `string` object allocated in the first line. Worse, if we tried to `delete lu.myString` at this point, because `lu.myString` was not the last field written to in the `union`, we would end up with undefined behavior that will almost certainly be a program crash.*

Tagged Unions

Because reading back a value from a `union` that wasn't the last value written can cause problems, C++ programmers commonly use a technique known as a *tagged union*, a pair of a `union` variable and a flag tracking which variable is currently active. For example, if we wanted to use the `LegalUnion` class declared above, we might create an enumerated type like this one:

```
enum UnionTag {StringType, VectorType};
```

Now, whenever we create a variable of type `LegalUnion`, we would also create a `UnionTag` variable to track the last-accessed field. Provided that we update the `UnionTag` variable, the program can safely determine which element of the `union` is valid without worrying about reading garbage data. Here's a code snippet that demonstrates how a tagged union might work:

```
LegalUnion lu;
UnionTag tag;

lu.myString = new string("Help!  I can't pay my union dues!");
tag = StringType; // Mark that we stored a string.
```

If we then wanted to write a function that accessed the fields of the `union`, we could then write it like this:

```
void DoSomething(const LegalUnion& lu, UnionTag tag)
{
    /* Based on the type of the element, perform different behaviors. */
    switch(tag)
    {
        case StringType:
            cout << "Length of string: " << lu.myString->length() << endl;
            break;
        case VectorType:
            cout << "Length of vector: " << lu.myVector->size() << endl;
            break;
        default:
            NOT_REACHED("Invalid tag."); // See Handout #8 for more info.
    }
}
```

Why Use `unions`?

`unions` are difficult to use correctly. You cannot store most types in a `union` and must be careful only to read the last value written. If this is the case, why use a `union` at all? Why not just have a `struct` containing multiple fields but only reference one of them at a time?

---

* This will invoke the `string` class's destructor on the memory reserved for the `vector`. This is not a pretty sight, trust me!

The answer, as is commonly the case in C++, is efficiency. Because at most one element of a `union` can be active at any one time, the C++ compiler can implement `union`s by having all of the elements in the `union` use overlapping memory locations. As a result, the size of a `union` is determined only by the size of the largest element it contains. Thus a `union` containing an `int`, a `char*`, a `double`, a `vector<int>*`, and a `float` only requires eight bytes of storage on most machines, the space required to store the `double`. By contrast, a `struct` containing that information would require storage space for each of these elements, which on 32-bit machines would be at least twenty-four bytes. If we treated the `struct` like a `union` (i.e. only used one element at a time), the `struct` is at least three times as memory intensive as the `union`. If we plan on creating multiple instances of the `union`, as would be the case in a Javascript interpreter (one `union` per variable) or a CSS parser (one `union` per CSS declaration), the savings can accumulate very quickly. Moreover, if we need to add additional elements to the `union`, provided that the new fields do not take up more space than the largest element of the `union`, we do not incur any additional space overhead.

The Assignment

In this assignment, you'll create a variant type called `Variant` that manages a tagged union of an `int` and a `string`. To simplify the task of implementing `Variant`, I've broken the project down into four smaller steps. While you're free to implement the class features in any order you see fit, I strongly encourage you to write `Variant` in the suggested order since each task builds off of its predecessors. A complete interface for the `Variant` class is at the back of this handout.

Step 0: Implement the Basic `Variant` Class.

Before diving into the `Variant`'s more advanced features, it's probably best to warm up by implementing the `Variant` basics. Below is a partial specification of the `Variant` class that provides core functionality:

Figure 0: Basic (incomplete) interface for the `Variant` class

```
class Variant
{
public:
    /**
     * An enumerated type that represents the type of the variable currently
     * being stored in the Variant.
     */
    enum VariantType {IntType, StringType};

    /**
     * Conversion constructors allow us to implicitly convert between regular
     * C++ types and Variants.
     */
    Variant(int val = 0);
    Variant(const char* val);
    Variant(const string& val);

    /**
     * Cleans up the Variant, freeing any dynamically-allocated memory.
     */
    ~Variant()

    /**
     * Returns the type of the object currently stored inside the Variant.
     */
    VariantType getType() const;

    /**
     * Returns the value of the object being stored.  Both of these functions
     * first check to make sure that the object has the correct type and
     * raise assertions otherwise; see the next section for more details.
     */
          int& asInt();
    const int& asInt() const;

          string& asString();
    const string& asString() const;
};
```

These functions are defined in more detail here:

`Variant(int val = 0)`	Constructs a new `Variant` holding the specified integer. Note that because this constructor is not marked `explicit`, it is a conversion constructor that lets us convert between regular C++ `int`s and our custom `Variant` type. Notice that we have set the default value of the integer parameter to 0. This means that if we declare a `Variant` object and don't specify otherwise, it defaults to holding the integer value 0. This behavior is entirely arbitrary and in fact we could just as easily have put the `Variant` into some form of "empty" state. However, making the `Variant` default to holding the integer value 0 simplifies the rest of the interpretation, so we will keep it this way.

Variant(const string& val)
Variant(const char* val)
	Constructs a new `Variant` holding the specified string value. There are two different constructors provided here, one for C-style strings and one for C++-style strings. This may seem redundant, but makes the client-side use of `Variant` much simpler. Since C++ makes at most one implicit conversion at a time, if we only provided a constructor taking in a `string`, then code like this would not compile:
	
	`void DoSomething(Variant v);`
	`DoSomething("Hello!"); // Error`
	
	This code does not compile correctly because C++ can't see how to convert between the `char*` string literal `"Hello!"` and the expected parameter of type `Variant`. By explicitly providing a constructor which accepts a `char*`, however, we eliminate this problem.

~Variant()	Cleans up any memory allocated by the `Variant`. In particular, if the `Variant` stores a string value, it should be deallocated here.
VariantType getType() const	Returns the type of the current value of the `Variant`.
int& asInt() **const int&** **asInt() const** **string& asString()** **const string&** **asString() const**	Returns a reference to the value stored in the `Variant` as either an `int` or a `string`. Because the `Variant` might not actually be of the proper type, both of these functions should raise an `assert` error if the type of the `Variant` is incorrect. This function is `const`-overloaded so that we can both read values from `const Variant`s and write values to non-`const Variant`s. If the `Variant` holds an integer and the client calls `asString`, the function should raise an `assert` error rather than returning a string representation of the integer. Similarly, calling `asInt` when the `Variant` holds a string that can be parsed into an integer should also call `assert`. The purpose of the `Variant` is to hold an object of one of two distinct types, so these sorts of conversions are not necessary.

Although there are many ways that you could implement the `Variant` behind the scenes, to get more practice with `union`s your solution **must** implement the `Variant` class using a tagged union as described above. My suggestion is to define the `private` section of the class to include the following:

```
union VariantUnion
{
    int     intValue;
    string* strValue; // Or char* strValue, if you're up for a challenge.
};

VariantUnion value; // Actual stored value
VariantType  type;  // Tag tracking active field
```

Here, the `VariantUnion` data member is the actual `union` holding either an `int` or a pointer to a `string`, and the `type` field keeps track of which of these two types the `union` actually holds. It is critically important that you keep track of the type of the object stored in the `union`, since if you're storing a string in the `strValue` field you will need to explicitly manage the memory for it.

Step 1: Add Deep-Copying Support

Now that you have the basic `Variant` class written, it's time to write a copy constructor and assignment operator for the `Variant` class. Remember that to copy a `Variant` that stores a string, you will need to dynamically-allocate memory to hold the copy.

You are free to implement the copy constructor and assignment operator any way you see fit, either using the approach outlined in Handout #23 or using copy-and-swap as detailed in Problem Set 3. If you use copy-and-swap, it is perfectly acceptable to make `swap` a public member function.

An interesting consequence of having both an assignment operator and a set of conversion constructors for the `Variant` type means that defining an assignment operator lets you write code that looks like the following:

```
Variant v;
v = 137;
```

This code is legal because there is a legal implicit conversion from `int` to `Variant`. Thus, once you have written a copy constructor and assignment operator, it is possible to treat `Variant`s much in the same way as you would primitive types. For example, all of the following code should work correctly:

```
Variant v = "This is a string!";
v = "And now it's another string!";
v = 42;
```

Step 2: Add Support for IOStream

Now that you've gotten the core `Variant` type down, it's time to spice up the class by integrating it into the existing C++ libraries. Implement a stream insertion operator for the `Variant` type that prints out the value of the object stored in the `Variant`. That is, given this code:

```
Variant v = /* ... */
cout << v << endl;
```

The program should print out an integer value if `v` stores an integer and a string value otherwise. You may want to look into Handout #26 on operator overloading for more details.

Step 3: Create a Comparator for Variants

The `Variant` type we've defined so far is suitable for use in a wide range of tasks, but unfortunately will not work as the key in an STL `map` or as a value in an STL `set` because we haven't defined `operator <` for `Variant`. Unfortunately, writing `operator <` for the `Variant` would prove to be a mistake. Suppose that we did provide an implementation of `operator <` such that we could store `Variant`s in STL associative containers. We would then need to define `operator <` such that any two `Variant`s could be compared, including `Variant`s storing different types of elements. This is somewhat suspicious, because it allows us to write code like this:

```
Variant v = "This is a string!";
if(v < 137)
    cout << "My variable is less than 137." << endl;
```

Because we've defined an implicit conversion from `int`s to `Variant`s through our conversion constructor, this code is perfectly legal. However, it leads to extremely counterintuitive code because it allows us to silently compare two values of completely different types.

We are at an impasse – we want to be able to store `Variant`s in an STL `map` or `set`, but defining the < operator as we normally would will let us write nonsensical code. How can we resolve this?

If you'll recall from Handout #27 on functors, we can store custom objects in an STL `map` or `set` if we define a custom comparison functor whose `operator()` function compares two elements. In this last portion of the assignment, we'll define a custom functor for the `Variant` type which can perform comparisons. That way, if we want to store `Variant`s in an STL `map` or `set` we can do so by explicitly indicating the comparison function. Moreover, since the comparison requires the explicit use of the comparison functor, we don't run the risk of allowing misleading statements like `v < 5` to compile.

Modify the `Variant` class definition by adding the following `struct` to its public section:

*Figure 1: Comparator interface for the **Variant** class*

```
/**
 * Allow the comparator to access Variant private data members if necessary.
 */
friend struct Comparator;
struct Comparator
{
    bool operator()(const Variant& one, const Variant& two) const;
};
```

The `Comparator` `struct`'s `operator()` should compare two `Variant`s in any way that you see fit, provided that the comparison function has similar properties as the mathematical < operator. For example, given two `Variant`s `one` and `two`, if `one` and `two` are not equal, then exactly one of `operator()(one, two)` and `operator()(two, one)` should return true, and if `one` compares less than `two` and `two` compares less than `three`, `one` should compare less than `three`.

Once you have defined the `Comparator` `struct`, you can begin storing `Variant`s in an STL `map` or `set` using syntax such as the following:

```
typedef map<Variant, Variant, Variant::Comparator> VariantMap;

VariantMap myMap;
myMap["Isn't this cool?"] = "Yup!";
myMap[137] = "This should strike you as pretty awesome.";
```

Step 4: Test your `Variant`!

Now that your `Variant` is fully up-and-running, it's time to test your code to see that you've ironed out all of the bugs. To help you get started, I've provided a test harness on the CS106L website. The testing framework is far from complete, but should help set you on the path toward bug-free code.

Hints and Advice

This assignment is not as difficult as it may appear. The resulting class implementation is not particularly long (my solution runs at just under two hundred lines of code, including whitespace and simple commenting), and if you understand the basics of how to manipulate the underlying `union` the code can be very elegant. That said, there are some points to be aware of when implementing the `Variant` class. Here are some tips and tricks on how to avoid common pitfalls:

- Make sure that you don't store the `union` tag variable tracking which element of the `union` was last accessed inside the `union` itself. If you do, marking which variable was last accessed would overwrite the contents of the `union`.
- Because `union`s cannot store `string` variables, you will need to store strings in the `union` either as dynamically-allocated C-style strings or as a pointer to a C++ `string` object. Both of these approaches require some form of explicit memory management, so be sure that your destructor and assignment operator deallocate this memory if it exists. On a similar note, if the `Variant` holds an integer value, be sure that you do *not* deallocate the memory for the string, since there is no string to deallocate.
- Remember that to define a stream insertion operator, you should define `operator <<` as a free function and may want to make it a `friend` of the `Variant` class. Feel free to use the lecture code from the CS106L website as a reference.
- Make sure that your comparator class can compare any two distinct `Variant`s. One possible way to implement this function is to compare the `Variant`s first by their type, arbitrarily making integers precede strings or strings precede integers, and then comparing their values only if the types match. The exact way that your comparator works is not important, so long as you define a total ordering over `Variant`s.
- *Test your code thoroughly!* The testing code available on the website is a good starting point, but there may be a few edge cases it does not exercise. Think about what sorts of coding mistakes are common with C++ classes (i.e. self-assignment) and make sure that you have all of your edge cases covered.
- *Don't hesitate to ask questions!* This assignment stresses a lot of the language features we've covered over the past few weeks and introduces `union`s, which can be a bit tricky. If you need any guidance, advice, or suggestions, send me an email or come talk to me after class and I'd be glad to help out.

Extensions

There are many a features you might want to consider adding to the `Variant`. If you're up for a bigger challenge, consider implementing the following extensions, or come up with your own!

- **Numeric conversions:** The `Variant` type we've defined here can hold an integer value or a string value, but does not allow implicit conversions between these two types. For example, we might want to allow the user to get a string representation of the integer value held in a `Variant`, and in some cases can convert a string stored in a `Variant` into an integer (if, for example, the string is "137" or "271828"). Consider modifying `asInt` and `asString` to perform these sorts of conversions when possible.
- **Stream extraction:** The current incarnation of the `Variant` class can be written into a stream using << but cannot be read from a stream using >>. Consider writing a stream extraction operator for the `Variant` type. How might you differentiate between integer and string inputs? Is it always unambiguous?

- **Additional types:** Can you devise an elegant framework for adding more types to the `Variant`? Several approaches exist, whether using the X Macro trick or through more advanced means.

More to Explore

If you're interested in using variant types in your programs, you might want to consider the Boost library's `any` type. Unlike the `Variant` defined here, `any` can hold on to a variable of any type, including other `any` variables. The curious reader might want to check out `any` – it's a remarkable class!

Submission Instructions

Once you've completed the assignment, email all of your source files to htiek@cs.stanford.edu, putting your name and SUNetID on top of all of your files. If you've added any extensions or made any changes to the code base, make a note of it in your submission email and I'd love to see what you've cooked up. Then pat yourself on the back – you've just written your first major C++ class!

Figure 2.1: The complete interface for the `Variant` class, part 1.

```
class Variant
{
public:
    /**
     * Enumerated type used to track the type of the value currently stored
     * inside the Variant.
     */
    enum VariantType {IntType, StringType};

    /**
     * Default constructor sets the Variant to hold the integer value zero.
     * In addition, this constructor is used to perform implicit conversions
     * between ints and Variants.
     */
    Variant(int val = 0);

    /**
     * Conversion constructors which transform C and C++ style strings into
     * Variants.
     */
    Variant(const char* val);
    Variant(const string& val);

    /**
     * Copy constructor and assignment operator are responsible for handling
     * deep-copying of Variants.
     */
    Variant(const Variant& other);
    Variant& operator= (const Variant& other);

    /**
     * Destructor cleans up any memory allocated by the Variant, if any.
     */
    ~Variant()
```

Figure 2.2: The complete interface for `Variant`, part 2.

```
    /**
     * If you implement the assignment operator using copy-and-swap, you can
     * optionally implement this function.
     */
    void swap(Variant& other);

    /**
     * Returns the type of the element stored inside the Variant.
     */
    VariantType getType() const;

    /**
     * Returns a reference to the element stored inside the Variant as either
     * an int or a string.  If the stored value has the wrong type (i.e.
     * the Variant stores 137 but the client calls asString), your
     * implementation should raise an assert error.  You do not need to
     * worry about implicit conversions from ints to strings or vice-versa-
     * if the client stores an integer value it can only be accessed through
     * asInt() and if the client stores a string it can only be accessed
     * through asString().
     */
          int&    asInt();
    const int&    asInt() const;
          string& asString();
    const string& asString() const;

    /**
     * Writes the value stored in the Variant to the specified output stream.
     */
    friend ostream& operator<< (ostream& out, const Variant& toPrint);

    /**
     * A struct whose operator() function compares two Variants.  The actual
     * means by which this comparison is made is unspecified, but is
     * guaranteed to be a valid comparison function.
     */
    friend struct Comparator;
    struct Comparator
    {
        bool operator()(const Variant& one, const Variant& two) const;
    };
private:
    /* ... implementation specific ... */

    /* Remember that the Variant MUST be backed by a tagged union.  I
     * suggest the following:
     */
     union VariantUnion
     {
         int     intValue;
         string* strValue;
     };
     VariantUnion value;
     VariantType  type;
};
```