Duck Typing with C++ templates

So, folks, what is this Duck Typing thingy? Trusty wikipedia says: In duck typing, an object's suitability is determined by the presence of certain methods and properties (with appropriate meaning), rather than the actual type of the object (https://en.wikipedia.org/wiki/Duck_typing). Let's take a look at a concrete example from a pet library of mine.

Often in code - especially often when logging stuff - you want to represent things as a string to be printed. A useful start is to use function overloading, mainly because there are some types of things that you cannot really extend. For example, you could have something like this:

```
inline std::string as_string(const char* p)
{
    return p ? p : "<nullptr>";
}

inline std::string as_string(bool value)
{
    return value ? "true" : "false";
}

inline std::string as_string(int32_t number)
{
    return string::format("%d", number);
}
```

The nice thing of this is: you can type `as_string(x)` and for many things, you will get out a useful string representation without having to care what the actual object is.

Enter objects. For many of these, the above approach will work as well:

```
inline std::string as_string(const std::vector<std::string> v)
{
    return string::format("vector<string> with %d objects", (int) v.size());
}

inline std::string as_string(const my_elaborate_class& c)
{
    return string::format("my_elaborate_class at %p", &c);
}
```

Now, maybe you want to have more information in `as_string`, and maybe you need to be able to access private data members of the objects. This can be solved by introducing a member function `as_string`, starting with this design:

```
class foo
{
public:
    std::string as_string() const;
};

std::string as_string(const foo& x)
{
    return x.as_string();
}

class bar
{
public:
    std::string as_string() const;
};

std::string as_string(const bar& x)
{
    return x.as_string();
}
```

This works only if you write separate `as_string` overloads for each class that implements `as_string` members, a rather pointless task. Nothing for us lazy programmer folk!

You could define an interface and force all objects to inherit it:

```
interface can_be_represented_as_string
{
    virtual std::string as_string() const = 0;
};

class foo : public can_be_represented_as_string
{
    virtual std::string as_string() const override;
};

class bar : public can_be_represented_as_string
{
    virtual std::string as_string() const override;
};

std::string as_string(const can_be_represented_as_string& something)
{
    return something.as_string();
}
```

OK, nice! Now you can throw almost anything at `as_string`, and get a string back. This works, but it requires you to define inheritance on *all* your objects, which is an eyesore and anyway may sometimes not be feasible, because you may have objects that you don't have control over (or you already have a large hierarchy of objects and people would complain if you'd start messing up their inheritances). Enter duck typing

As written at the beginning, in duck typing, an object's suitability is determined by the presence of certain methods and properties (with appropriate meaning), rather than the actual type of the object (https://en.wikipedia.org/wiki/Duck_typing)

Looking back where we started:

```
class foo
{
public:
    std::string as_string() const;
};

class bar
{
public:
    std::string as_string() const;
};
```

Both `foo` and `bar` are objects that have methods named `as_string` with appropriate meaning, but the types are different. What can we do about that? This:

```
template <typename T> std::string as_string(const T& o)
{
    return o.as_string();
}
```

So this is a generic method that works because there are methods with the proper meaning - not because the type has inherited something special. Nice!

OK, but now what about objects that *do no* implement a method `as_string` with appropriate meaning? It would be nice if we could say something like

1. If there is a `as_string` overload, use that
2. If the object has `as_string()`, use that
3. Otherwise, default to just dumping the address of the object

At this point, the first two work, but the third one is causing a headache. It turns out that there is a SFINAE based idiom to detect if a type has a member (http://stackoverflow.com/questions/257288/is-it-possible-to-write-a-c-template-to-check-for-a-functions-existence) but that is only half the solution: most (https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Member_Detector) examples (http://stackoverflow.com/questions/1005476/how-to-detect-whether-there-is-a-specific-member-variable-in-class) just end

up in a template function that can be used to *detect the fact*, but not really do something about it. To see the problem, let's assume you've followed the examples I've linked to and have a template has_as_string<type>::value. And you have the following setup:

```
class foo
{
public:
    std::string as_string() const;
};

class baz
{
public:
    // look ma, no 'as_string'
};
...
static_assert(has_as_string<foo>::type, "");
static_assert(!has_as_string<baz>::type, "");
...
template <typename T> std::string as_string(const T& o)
{
    if(has_as_string<T>::value)
        return o.as_string();
    return string::format("%p", &o);
}
```

The static-asserts will work (provided your implementation is correct), but the as_string template won't, because it will attempt to generate code that uses as_string even if the object doesn't have the method. So you need something more involved:

```
template <typename T> class repr_type
{
public:
    repr_type(const T& o)
        :
        m_o(o)
    {
    }

    std::string as_string() const
    {
        return call_as_string<T>(nullptr);
    }

private:
    template <class C> std::string call_as_string(decltype(&C::as_string)) const
    {
        return m_o.as_string();
    }

    template <class C> std::string call_as_string(...) const
    {
        return string::format("%p", &m_o);
    }

    const T& m_o;
};

template <typename T> std::string as_string(const T& o)
{
    return repr_type<T>(o).as_string();
}
```

It is worth looking at the fine print on this one:

- Based on whether the type passed has a method as_string, this template will print either the result of that function, or simply the objects address.
- Note the template function call_as_string: the template resolver will prefer the more specific type, and it will be able to use the first template if the address of as_string can be taken.
- Note that because this is a template function, it is not compiled until it is actually used: so for things that don't have as_string a call to as_string will never be issued.

- Also note for this to work, I cannot also pass in the object instance: so I need a proxy object - `repr_type` template - to hold a reference to the object while letting SFINAE do its work

---

So what use is any of this?

Well, assume you have a set of `as_string` thingies as discussed here: functions using overloading, some template magic and so on. All is fine and well and one day you start writing a method that takes strings. For example, assume you have a spreadsheet and you want to fill it with text:

```
class spreadsheet
{
public:
    void set(int x, int y, const char* text);
};
```

Now for most of the things you have, you can write

```
spreadsheet s;
...
s.set(0,0,as_string(i)); // where i could be "anything"
```

But wait, it gets better: let's enhance the spreadsheet class:

```
class spreadsheet
{
public:
    void set(int x, int y, const char* text);

    template <typename T> void set(int x, int y, const T& instance)
    {
        set(x,y, as_string(instance));
    }
};
```

This will result in even more clean client code:

```
spreadsheet s;
...
s.set(0,0,i); // where i could be "anything"
```

Personally, I like my client code as readable as possible. I know that can be hard, because C++ tries really hard to be ugly (https://news.ycombinator.com/item?id=5056156) and uglier still, but sometimes after years disparate features like overloading and templates and improved rules in C++11 work together to make the code look simple. Nice!