boost::any && boost::string_ref

Two simple and useful libraries

Created by Ben / @predatorhat

boost::string_ref

#include <boost/utility/string_ref.hpp>

Documentation

boost::string_ref

- Introduced 1.53.0
- Now really mature
- Base on Jeffrey Yaskin's N3442: string_ref: a non-owning reference to a string
- Which is based on LLVM's StringRef class
- Want to see that in C++17

Basically just

```
template <class CharT> class string_ref
{
    // ...
    const CharT* ptr_;
    std::size_t len_;
};
```

boost::string_ref

- A read-only *view* into a std::string
- Similar interface to std::string
- Container requirements? It has a begin() and end()

boost::string_ref

Really cheap to copy

```
boost::string_ref get_body() const noexcept
{
    auto node = document->find_node("body");
    if (node)
    {
        // Construct from a pointer, length pair
        return boost::string_ref(node->value(), node->value_size());
    }

    return boost::string_ref(); // This will have length() == 0
}
```

boost::string_ref

Drop in replacement for const char*

```
// Before:
void func(const char* str)
{
    assert(str);
    // str cannot contain \0-terminus
}

// After:
void func(boost::string_ref str)
{
  // No need to check nullptr dref
  // str can contain as much \0 as it likes
  // Note: we do not pass as const&
}
```

boost::string_ref

String-like operations

```
// Creates a new string_ref
string_ref substr(size_type pos, size_type n=npos) const;
bool starts_with(CharT c) const;
bool starts_with(string_ref x) const;
bool ends_with(CharT c) const;
bool ends_with(string_ref x) const;
```

boost::any

Type erasure!

boost::any

```
#include <boost/any.hpp>
```

Documentation

Header <boost/any.hpp>

```
namespace boost
{
    class bad_any_cast;
    class any;
    void swap(any&, any&);
    template <typename T> T any_cast(any&);
    template <typename T> T any_cast(any&&);
    template <typename T> T any_cast(const any&);
    template <typename T> const T* any_cast(const any*);
    template <typename T> T* any_cast(any*);
}
```

boost::any in action

```
int i = 42;
boost::any a;
a = i;
a = 2.0;

struct foo {};
a = foo{};

a = std::string("anything!");

void func(boost::any val)
{
    // Throws if val doesn't hold expected type
    std::string str = boost::any_cast<std::string>(val);

    std::cout << str << std::endl;
}

func(a); // Prints 'anything!'
```

boost::any

Useful if start and end point know the type, but everything in between shouldn't

boost::any

```
std::vector<any> something;
something.push_back(any(1729));
something.push_back(any(std::string("Coca Cola")));
something.push_back(any(web_socket("http://example.com/")));
whatever.set_something(std::move(something));

// Later in a galaxy far away...

auto container = whatever.get_something();
int carmichael_number = std::any_cast<int>(container[0]);
std::string drink = std::any_cast<std::string>(container[1]);
web_socket wsock = std::any_cast<web_socket>(container[2]);
```

boost::any

Magic: templates, run-time polymorphism, and RTTI

boost::any implementation

```
class any
{
    // ...

private:
    struct placeholder
    {
        virtual ~placeholder() = default;
        virtual const std::type_info& type() const = 0;
        virtual placeholder* clone() = 0;
    };
    template <class T>
    struct holder : public placeholder
    {
        // ...
        T held;
    };
    placeholder* content;
};
```

```
class any
{
public:
    any();
    any(const any&);
    any(any&&);
    template <typename T> any(const T&);
    template <typename T> any(T&&);
    any& operator=(const any&);
    any& operator=(any&&);
    template <typename T> any& operator=(const T&);
    template <typename T> any& operator=(T&&);
    ~any();

    // ...
};
```

boost::any alternatives

- void*
- Take a look at Boost.TypeErasure library
- If you happen to know the types up-front: boost::variant
- Always: custom solution for your problem at hand (common base-class, compile-time polymorphism)
- boost::spirit::hold_any (Not part of public API though)

boost::any performance

Complicated

Good: function call overhead is exactly the same as with pointer-based inheritance (virtual function dispatch)

Bad: a lot of heap allocations. Underlying type is copied a lot!

boost::any

Pros	Cons
Value semantics!	Copying is quite expensive
No need for new or delete	
Works for nearly any type (only requirement: copyable)	