Migration from boost::any

The lesson shows how transition from boost to std can make things more flexible for you.

WE'LL COVER THE FOLLOWING ^

- Examples of std::any
 - Parsing files
 - Message Passing
 - Properties

Boost Any was introduced around the year 2001 (Version 1.23.0). Interestingly, the author of the boost library - Kevlin Henney - is also the author of the proposal for std::any. So the two types are strongly connected, and the STL version is heavily based on the predecessor.

Here are the main changes:

Feature	Boost.Any (1.67.0)	std::any
Extra memory allocation	Yes	Yes
Small buffer optimisation	Yes	Yes
emplace	No	Yes
in_place_type_t in constructor	No	Yes

There are not many differences between the two types. Most of the time you can easily convert from boost.any into the STL version.

Examples of std::any

The core of std::any is flexibility. In the below examples, you can see some ideas (or concrete implementations) where holding variable type can make an application a bit simpler.

Parsing files

In the examples for std::variant you can see how it's possible to parse
configuration files and store the result as an alternative of several types. If
you write an entirely generic solution - for example as a part of some library,
then you might not know all the possible types.

Storing std::any as a value for a property might be good enough from the performance point of view and will give you flexibility.

Message Passing

In Windows API, which is C mostly, there's a message passing system that uses message ids with two optional parameters which store the value of the message. Based on that mechanism you can implement <code>WndProc</code> to handle the messages passed to your window/control:

```
LRESULT CALLBACK WindowProc(
_In_ HWND hwnd,
_In_ UINT uMsg,
_In_ WPARAM wParam,
_In_ LPARAM 1Param
);
```

The trick here is that the values are stored in wParam or lParam in various forms. Sometimes you have to use only a few bytes of wParam ...

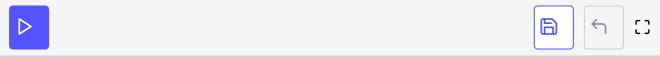
What if we changed this system into std::any, so that a message could pass
anything to the handling method?

For example:

```
#include <string>
#include <iostream>
#include <any>
#include whility
```

```
#include <utility>
void* operator new(std::size_t count)
{
    std::cout << " allocating: " << count << " bytes" << std::endl;</pre>
    return malloc(count);
}
void operator delete(void* ptr) noexcept
    std::puts("global op delete called");
    std::free(ptr);
}
class Message
public:
    enum class Type
        Init,
        Closing,
        ShowWindow,
        DrawWindow
    };
public:
    explicit Message(Type type, std::any param) :
        mType(type),
        mParam(param)
    explicit Message(Type type) :
        mType(type)
        }
    {
    Type mType;
    std::any mParam;
};
class Window
{
public:
    virtual void HandleMessage(const Message& msg) = 0;
};
class DialogWindow : public Window
public:
    void HandleMessage(const Message& msg) override
        try
        {
            switch (msg.mType)
            {
                case Message::Type::Init:
                     std::cout << "Init\n";</pre>
                     break;
                case Message::Type::Closing:
                     std::cout << "Closing\n";</pre>
                     break;
                case Message::Type::ShowWindow:
                     auto pos = std::anv cast<std::pair<int, int>>(msg.mParam);
```

```
std::cout << "ShowWidow: "<< pos.first << ", " << pos.second << '\n';</pre>
                    break;
                }
                case Message::Type::DrawWindow:
                    auto col = std::any_cast<uint32_t>(msg.mParam);
                    std::cout << "DrawWindow, color: "<< std::hex << col << '\n';</pre>
                    break;
                }
            }
        }
        catch(const std::bad_any_cast& e)
            std::cout << e.what() << '\n';
        }
    }
};
int main()
    auto a = std::make_any<int>(10);
    DialogWindow dlg;
    Message m1(Message::Type::Init);
    dlg.HandleMessage(m1);
    Message m2(Message::Type::ShowWindow, std::make_pair(10, 11));
    dlg.HandleMessage(m2);
    Message m3(Message::Type::DrawWindow, static_cast<uint32_t>(0xFF00FFFF));
    dlg.HandleMessage(m3);
    dlg.HandleMessage(Message{Message::Type::Closing});
}
```



Now you can send a message to a window like:

```
Message m(Message::Type::ShowWindow, std::make_pair(10, 11));
yourWindow.HandleMessage(m);
```

And then the window can respond to the message with the following message handler (as seen in the code above):

Of course, you have to define how the values are specified (what the types of the value of a message are), but now you can use real types rather than doing various tricks with integers.

Properties

The original paper that introduces any to C++, N19394 shows an example of a property class.

```
struct property
{
  property();
  property(const std::string &, const std::any &);
  std::string name;
  std::any value;
};

typedef std::vector<property> properties;
```

The properties object looks quite powerful as it can hold many different types. One of the examples where such structure might be leveraged is a game editor.

So far we have discussed quite a lot of concepts, let's have a quick wrap up in the following section!