High Performance Dynamic Typing in C++ using a Replacement for boost::any

Christopher Diggins, 10 May 2011 CPOL

A high-performance alternative to boost::any.

Introduction

The Boost library provides a very useful little class called `boost::any` which can contain a value of virtually any type as long as that value supports copy construction and assignment. This `boost::any` type allows dynamic querying of the contained type, and safe type conversions. Despite its usefulness, `boost::any` is not as efficient as it could be, so I have rewritten my own, which I have immodestly called `cdiggins::any`.

Updated in 2011: After five years, I have rewritten the class from scratch to fix some really nasty bugs and to simplify the code.

Using cdiggins::any

The `cdiggins::any` type can be used to hold normal value types, and provides a mechanism to safely and explicitly cast back to the appropriate type.

```
any a = 42;
cout << a.cast<int>() << endl;
a = 13;
cout << a.cast<int>() << endl;
a = "hello";
cout << a.cast<const char*>() << endl;
a = std::string("1234567890");
cout << a.cast<std::string>() << endl;
int n = 42;
a = &n;
cout << *a.cast<int*>() << endl;
any b = true;
cout << b.cast<bool>() << endl;
swap(a, b);
cout << a.cast<bool>() << endl;
a.cast<bool>() = false;
cout << a.cast<bool>() << endl;
```

Design of cdiggins::any

The `cdiggins::any` class contains two pointers: one to a policy class (`any::policy`) and one is a pointer to the data (`any::data`) which may in certain cases be used to contain the data itself (for example, if storing a primitive data type smaller than or equal to the size of a pointer).

The policy class is used for performing allocations, deallocation, copies, etc., and determines whether to use`any::data` to hold the data or point to the data.

Finally, without further ado, here is the implementation of the `cdiggins::any` class for your enjoyment:

```
#pragma once
/*
 * (C) Copyright Christopher Diggins 2005-2011
 * (C) Copyright Pablo Aguilar 2005
 * (C) Copyright Kevlin Henney 2001
 *
 * Distributed under the Boost Software License, Version 1.0. (See
 * accompanying file LICENSE_1_0.txt or copy at
 * http://www.boost.org/LICENSE_1_0.txt
 */

#include <stdexcept>

namespace cdiggins
{
namespace anyimpl
{
    struct bad_any_cast
    {
    };

    struct empty_any
    {
    };

    struct base_any_policy
    {
        virtual void static_delete(void** x) = 0;
        virtual void copy_from_value(void const* src, void** dest) = 0;
        virtual void clone(void* const* src, void** dest) = 0;
        virtual void move(void* const* src, void** dest) = 0;
        virtual void* get_value(void** src) = 0;
        virtual size_t get_size() = 0;
    };

    template<typename T>
    struct typed_base_any_policy : base_any_policy
```

```
{
    virtual size_t get_size() { return sizeof(T); }
};

template<typename T>
struct small_any_policy : typed_base_any_policy<T>
{
    virtual void static_delete(void** x) { }
    virtual void copy_from_value(void const* src, void** dest)
        { new(dest) T(*reinterpret_cast<T const*>(src)); }
    virtual void clone(void* const* src, void** dest) { *dest = *src; }
    virtual void move(void* const* src, void** dest) { *dest = *src; }
    virtual void* get_value(void** src) { return reinterpret_cast<void*>(src); }
};

template<typename T>
struct big_any_policy : typed_base_any_policy<T>
{
    virtual void static_delete(void** x) { if (*x)
        delete(*reinterpret_cast<T**>(x)); *x = NULL; }
    virtual void copy_from_value(void const* src, void** dest) {
       *dest = new T(*reinterpret_cast<T const*>(src)); }
    virtual void clone(void* const* src, void** dest) {
       *dest = new T(**reinterpret_cast<T* const*>(src)); }
    virtual void move(void* const* src, void** dest) {
      (*reinterpret_cast<T**>(dest))->~T();
      **reinterpret_cast<T**>(dest) = **reinterpret_cast<T* const*>(src); }
    virtual void* get_value(void** src) { return *src; }
};

template<typename T>
struct choose_policy
{
    typedef big_any_policy<T> type;
};

template<typename T>
struct choose_policy<T*>
{
    typedef small_any_policy<T*> type;
};

struct any;

/// Choosing the policy for an any type is illegal, but should never happen.
/// This is designed to throw a compiler error.
template<>
struct choose_policy<any>
{
```

```
        typedef void type;
    };

    /// Specializations for small types.
    #define SMALL_POLICY(TYPE) template<> struct
       choose_policy<TYPE> { typedef small_any_policy<TYPE> type; };

    SMALL_POLICY(signed char);
    SMALL_POLICY(unsigned char);
    SMALL_POLICY(signed short);
    SMALL_POLICY(unsigned short);
    SMALL_POLICY(signed int);
    SMALL_POLICY(unsigned int);
    SMALL_POLICY(signed long);
    SMALL_POLICY(unsigned long);
    SMALL_POLICY(float);
    SMALL_POLICY(bool);

    #undef SMALL_POLICY

    /// This function will return a different policy for each type.
    template<typename T>
    base_any_policy* get_policy()
    {
        static typename choose_policy<T>::type policy;
        return &policy;
    };
}

struct any
{
private:
    // fields
    anyimpl::base_any_policy* policy;
    void* object;

public:
    /// Initializing constructor.
    template <typename T>
    any(const T& x)
        : policy(anyimpl::get_policy<anyimpl::empty_any>()), object(NULL)
    {
        assign(x);
    }

    /// Empty constructor.
    any()
        : policy(anyimpl::get_policy<anyimpl::empty_any>()), object(NULL)
    { }
```

```
/// Special initializing constructor for string literals.
any(const char* x)
    : policy(anyimpl::get_policy<anyimpl::empty_any>()), object(NULL)
{
    assign(x);
}

/// Copy constructor.
any(const any& x)
    : policy(anyimpl::get_policy<anyimpl::empty_any>()), object(NULL)
{
    assign(x);
}

/// Destructor.
~any() {
    policy->static_delete(&object);
}

/// Assignment function from another any.
any& assign(const any& x) {
    reset();
    policy = x.policy;
    policy->clone(&x.object, &object);
    return *this;
}

/// Assignment function.
template <typename T>
any& assign(const T& x) {
    reset();
    policy = anyimpl::get_policy<T>();
    policy->copy_from_value(&x, &object);
    return *this;
}

/// Assignment operator.
template<typename T>
any& operator=(const T& x) {
    return assign(x);
}

/// Assignment operator, specialed for literal strings.
/// They have types like const char [6] which don't work as expected.
any& operator=(const char* x) {
    return assign(x);
}
```

```
    /// Utility functions
    any& swap(any& x) {
        std::swap(policy, x.policy);
        std::swap(object, x.object);
        return *this;
    }

    /// Cast operator. You can only cast to the original type.
    template<typename T>
    T& cast() {
        if (policy != anyimpl::get_policy<T>())
            throw anyimpl::bad_any_cast();
        T* r = reinterpret_cast<T*>(policy->get_value(&object));
        return *r;
    }

    /// Returns true if the any contains no value.
    bool empty() const {
        return policy == anyimpl::get_policy<anyimpl::empty_any>();
    }

    /// Frees any allocated memory, and sets the value to NULL.
    void reset() {
        policy->static_delete(&object);
        policy = anyimpl::get_policy<anyimpl::empty_any>();
    }

    /// Returns true if the two types are the same.
    bool compatible(const any& x) const {
        return policy == x.policy;
    }
};
}
```

Final Words

Thanks to Pablo Aguilar who helped with an early version of the `cdiggins::any` class. Thanks to Raute who helped me identify the critical problems in the original version of `cdiggins::any`.

License

About the Author

Christopher Diggins

Software Developer Autodesk

Canada

This article was written by Christopher Diggins, a computer science nerd who currently works at Autodesk as an SDK specialist.

Follow on Twitter Google LinkedIn