# Boost.Any

Kevlin Henney



## Table of Contents



# Introduction

There are times when a generic (in the sense of *general* as opposed to *template-based programming*) type is needed: variables that are truly variable, accommodating values of many other more specific types rather than C++'s normal strict and static types. We can distinguish three basic kinds of generic type:

- Converting types that can hold one of a number of possible value types, e.g. `int` and `string`, and freely convert between them, for instance interpreting `5` as `"5"` or vice-versa. Such types are common in scripting and other interpreted languages. `boost::lexical_cast` supports such conversion functionality.
- Discriminated types that contain values of different types but do not attempt conversion between them, i.e. `5` is held strictly as an `int` and is not implicitly convertible either to `"5"` or to `5.0`. Their indifference to interpretation but awareness of type effectively makes them safe, generic containers of single values, with no scope for surprises from ambiguous conversions.
- Indiscriminate types that can refer to anything but are oblivious to the actual underlying type, entrusting all forms of access and interpretation to the programmer. This niche is dominated by `void *`, which offers plenty of scope for surprising, undefined behavior.

The `boost::any` class (based on the class of the same name described in "Valued Conversions" by Kevlin Henney, *C++ Report* 12(7), July/August 2000) is a variant value type based on the second category. It supports copying of any value type and safe checked extraction of that value strictly against its type. A similar design, offering more appropriate operators, can be used for a generalized function adaptor, `any_function`, a generalized iterator adaptor, `any_iterator`, and other object types that need uniform runtime treatment but support only compile-time template parameter conformance.

# Examples

The following code demonstrates the syntax for using implicit conversions to and copying of any objects:

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

using boost::any_cast;
```

```
typedef std::list<boost::any> many;

void append_int(many & values, int value)
{
    boost::any to_append = value;
    values.push_back(to_append);
}

void append_string(many & values, const std::string & value)
{
    values.push_back(value);
}

void append_char_ptr(many & values, const char * value)
{
    values.push_back(value);
}

void append_any(many & values, const boost::any & value)
{
    values.push_back(value);
}

void append_nothing(many & values)
{
    values.push_back(boost::any());
}
```

The following predicates follow on from the previous definitions and demonstrate the use of queries on any objects:

```
bool is_empty(const boost::any & operand)
{
    return operand.empty();
}

bool is_int(const boost::any & operand)
{
    return operand.type() == typeid(int);
}

bool is_char_ptr(const boost::any & operand)
{
    try
    {
        any_cast<const char *>(operand);
        return true;
    }
    catch(const boost::bad_any_cast &)
    {
        return false;
    }
}

bool is_string(const boost::any & operand)
{
    return any_cast<std::string>(&operand);
}

void count_all(many & values, std::ostream & out)
{
    out << "#empty == "
        << std::count_if(values.begin(), values.end(), is_empty) << std::endl;
    out << "#int == "
        << std::count_if(values.begin(), values.end(), is_int) << std::endl;
    out << "#const char * == "
        << std::count_if(values.begin(), values.end(), is_char_ptr) << std::endl;
    out << "#string == "
        << std::count_if(values.begin(), values.end(), is_string) << std::endl;
}
```

The following type, patterned after the OMG's Property Service, defines name-value pairs for arbitrary value types:

```
struct property
{
    property();
    property(const std::string &, const boost::any &);

    std::string name;
    boost::any value;
};

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

The following base class demonstrates one approach to runtime polymorphism based callbacks that also require arbitrary argument types. The absence of virtual member templates requires that different solutions have different trade-offs in terms of efficiency, safety, and generality. Using a checked variant type offers one approach:

```
class consumer
{
public:
    virtual void notify(const any &) = 0;
    ...
};
```

# Reference

## *ValueType* requirements

Values are strongly informational objects for which identity is not significant, i.e. the focus is principally on their state content and any behavior organized around that. Another distinguishing feature of values is their granularity: normally fine-grained objects representing simple concepts in the system such as quantities.

As the emphasis of a value lies in its state not its identity, values can be copied and typically assigned one to another, requiring the explicit or implicit definition of a public copy constructor and public assignment operator. Values typically live within other scopes, i.e. within objects or blocks, rather than on the heap. Values are therefore normally passed around and manipulated directly as variables or through references, but not as pointers that emphasize identity and indirection.

The specific requirements on value types to be used in an `any` are:

- A *ValueType* is *CopyConstructible* [20.1.3].
- A *ValueType* is optionally *Assignable* [23.1]. The strong exception-safety guarantee is required for all forms of assignment.
- The destructor for a *ValueType* upholds the no-throw exception-safety guarantee.

## Header <boost/any.hpp>

```
namespace boost {
  class bad_any_cast;
  class any;
  template<typename ValueType> ValueType any_cast(const any &);
  template<typename ValueType> const ValueType * any_cast(const any *);
  template<typename ValueType> ValueType * any_cast(any *);
}
```

# Class bad_any_cast

The exception thrown in the event of a failed any_cast of an any value.

```
class bad_any_cast : public std::bad_cast {
public:
  const char * what() const;
};
```

# Class any

A class whose instances can hold instances of any type that satisfies ValueType requirements.

```
class any {
public:
  // construct/copy/destruct
  any();
  any(const any &);
  template<typename ValueType> any(const ValueType &);
  any & operator=(const any &);
  template<typename ValueType> any & operator=(const ValueType &);
  ~any();

  // modifiers
  any & swap(any &);

  // queries
  bool empty() const;
  const std::type_info & type() const;
};
```

# Description

## any construct/copy/destruct

1. `any();`

    1. **Postconditions:** `this->empty()`

2. `any(const any & other);`

    1. **Effects:** Copy constructor that copies content of `other` into new instance, so that any content is equivalent in both type and value to the content of `other`, or empty if `other` is empty.
    2. **Throws:** May fail with a `std::bad_alloc` exception or any exceptions arising from the copy constructor of the contained type.

3. `template<typename ValueType> any(const ValueType & value);`

    1. **Effects:** Makes a copy of `value`, so that the initial content of the new instance is equivalent in both type and value to `value`.
    2. **Throws:** `std::bad_alloc` or any exceptions arising from the copy constructor of the contained type.

4. `any & operator=(const any & rhs);`

    1. **Effects:** Copies content of `rhs` into current instance, discarding previous content, so that the new content is equivalent in both type and value to the content of `rhs`, or empty if `rhs.empty()`.
    2. **Throws:** `std::bad_alloc` or any exceptions arising from the copy constructor of the contained type. Assignment satisfies the strong guarantee of exception safety.

5.
```
template<typename ValueType> any & operator=(const ValueType & rhs);
```

1. **Effects:** Makes a copy of `rhs`, discarding previous content, so that the new content of is equivalent in both type and value to `rhs`.
2. **Throws:** `std::bad_alloc` or any exceptions arising from the copy constructor of the contained type. Assignment satisfies the strong guarantee of exception safety.

6.
```
~any();
```

1. **Effects:** Releases any and all resources used in management of instance.
2. **Throws:** Nothing.

# any modifiers

1.
```
any & swap(any & rhs);
```

1. **Effects:** Exchange of the contents of `*this` and `rhs`.
2. **Returns:** `*this`
3. **Throws:** Nothing.

# any queries

1.
```
bool empty() const;
```

1. **Returns:** `true` if instance is empty, otherwise `false`.
2. **Throws:** Will not throw.

2.
```
const std::type_info & type() const;
```

1. **Returns:** the `typeid` of the contained value if instance is non-empty, otherwise `typeid(void)`.
2. **Notes:** Useful for querying against types known either at compile time or only at runtime.

# Function any_cast

Custom keyword cast for extracting a value of a given type from an any.

```
template<typename ValueType> ValueType any_cast(const any & operand);
template<typename ValueType> const ValueType * any_cast(const any * operand);
template<typename ValueType> ValueType * any_cast(any * operand);
```

# Description

1. **Returns:** If passed a pointer, it returns a similarly qualified pointer to the value content if successful, otherwise null is returned. If passed a value or reference, it returns a copy of the value content if successful.
2. **Throws:** Overloads taking an `any` pointer do not throw; the overload taking an `any` value or reference throws `bad_any_cast` if unsuccessful.
3. **Rationale:** The value/reference version returns a copy because the C++ keyword casts return copies.

# Acknowledgements

Doug Gregor ported the documentation to the BoostBook format.