Kevlin Henney is an independent consultant and trainer based in the UK. He may be contacted at kevlin@curbralan.com.

FROM MECHANISM TO METHOD

Valued Conversions

"HOW WOULD YOU like to pay for that?" Good question. Digging deep into pockets, wallets, and bags uncovered a wealth of possibilities, a handful of different currencies and mechanisms to choose from: credit cards, debit cards, coins, bills, and a couple of IOUs, each form in some way substitutable for another when realizing monetary value.

Cash is the simplest, least troublesome form for small amounts and quick transactions. However, sifting through the metal and paper, it seemed that my currencies were no good. Well, that's not strictly true: The currencies were fine, just for somewhere other than here. Like any form of use, appropriate use of currency is context sensitive, requiring explicit conversion (typically incurring an overhead) for use elsewhere. The same was true of the debit cards I had, so I settled on one of the credit cards. Relatively transparent use, with all the mechanism of billing and conversion safely hidden behind a signature and a smile.

As the assistant struggled with the point-of-sale system, my thoughts inevitably turned to software. Substitutability in programming is often associated with good-practice use of inheritance in object-oriented development,[1,2] providing a thorough *is-a-kind-of* litmus test for structural relationships between classes. In this sense, it gives purpose and some sense of quality to what is otherwise simply a language mechanism; in and of itself inheritance is neither good nor bad.

Generally we can see substitutability as a measure of the fit between expectation, mechanism, and actual use. It is a more general principle than simply an inheritance recommendation, applying equally well to other mechanisms—of which C++ has many. Where practice makes sense of mechanism, thinking about substitutability offers an alternative way of thinking about language features. Table 1 identifies types of substitutability in C++ that we may consider useful in reasoning about our types and functions.[3]

Where a type defines a set of operations—not necessarily member functions[4,5]—applicable to a type, a type hierarchy defines the fit between types, and may or may not be associated with a class hierarchy. In the case of payment we can see that there is no useful structural relationship between the types that leads us to any concrete form of inheritance. Substitutability here is based on values and conversions between values. Sometimes the use is implicit, at other times it must be made explicit. Conversions can be fully value preserving, widening, or narrowing. Widening conversions are always safe and typically acceptable (e.g., tipping), whereas narrowing conversions may not be (e.g., shortchanging tends to lead to exceptional or even undefined behavior).

Rescuing me from further metaphor stretching, the point-of-sale system and the assistant's smile kicked into life.

VALUE CLASSES 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: They are typically fine-grained objects, representing simple concepts in the system such as quantities.[6]

In C++, values are associated with an idiomatic set of capabilities and conventions. The emphasis of a value lies in its state, not its identity. Thus 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. As a consequence of this immediacy, indirection transparency, and granularity, operator overloading and conversions often make sense for value classes whereas they do not for more granular, heap-based objects manipulated through pointers.

All Things to All People There are times when a generic (in the sense of *general* rather than *template-based programming*) type is needed, 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

Table 1. Different Kinds of Substitutability in C++.

Substitutability	Mechanisms
Conversions	Implicit and explicit conversions
Overloading	Overloaded functions and operators, often in combination with conversions
Derivation	Inheritance
Mutability	Qualification (typically `const`) and the use of conversions, overloading, and derivation
Genericity	Templates and the use of conversions, overloading, derivation, and mutability

them, for instance interpreting `5` as `"5"` or vice-versa. Such types are common in scripting and other interpreted languages. In implementation these are often interpreted strings, or encapsulated `unions` of a fixed set of types that freely support the required conversions, or closed class hierarchies.[7,8]

- 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. In implementation these are often held either as encapsulated, discriminated `unions` of a fixed set of types or through a combination of `void *` and a known type code.
- 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.

State of the Union In demonstrating substitutability concepts—conversions in particular—the remainder of this article is going to explore a generalized, discriminated `union` type named `any`.

Working from the inside out, how can a generic value contain any arbitrary value safely? A conventional `union` is out of the question, as these alias only a predefined, fixed set of types. A next guess might land on a `void *` and `const type_info *` pairing. This seems to allow easy creation and querying, but falls down on type-safe copying and destruction: How can you correctly copy or delete an instance of a type of which you are unaware? From this question comes the seed of a solution: Inheritance and runtime polymorphism offer a form of substitutability between types that allows us to work safely through a common interface while ignoring the differences we can neither know nor manage. A `virtual` destructor provides the mechanism for safe deletion, and a `virtual clone` function offers a route for safe copying—effectively a Virtual Copy Constructor.[2,9] However, this requires that value types inherit from a common base—not possible for preexisting types—and would defeat the original objective of the `any` class.

Template classes provide a mechanism for defining arbitrary containers. Combining templates with derivation reveals a solution (see Listing 1). A generalized base class, `placeholder`, offers the required copying, querying, and deletable interface. From this, the templated `holder` class fills in the details for any arbitrary type. This example mixes derivation substitutability and generic substitutability. The generic interface requirement is that contained values must be `CopyConstructible`.[10] Clients of `any` remain blissfully unaware of all this encapsulated detail. This design is most generally an example of the Adapter pattern,[9] and more specifically the External Polymorphism pattern.[11]

Listing 1. Representation and basic construction of a generalized union type.

```
class any
{
public:
  any()
    : content(0)
  {
  }
  ~any()
  {
      delete content;
  }
  const std::type_info &type_info() const
  {
      return content
        ? content->type_info()
        : typeid(void);
  }
  ...
private:
  class placeholder
  {
  public:
    virtual ~placeholder()
    {
    }
    virtual const std::type_info &
      type_info() const = 0;

    virtual placeholder *clone() const = 0;
  };
  template<typename value_type>
  class holder : public placeholder
  {
  public:
    holder(const value_type &value)
      : held(value)
    {
    }
    virtual const std::type_info &type_info() const
    {
        return typeid(value_type);
    }
    virtual placeholder *clone() const
    {
        return new holder(held);
    }
    const value_type held;
  };
  placeholder *content;
};
```

INWARD CONVERSIONS An implicit conversion from one or more other types into one we are developing can be supported by the introduction of one or more single-argument converting constructors on a class. Such conversions should be used in support of making con-

Listing 2. Inward conversions and helpers for a generalized union type.

```
class any
{
public:
  ...
  any(const any &other)
    : content(other.content ? other.content->clone() : 0)
  {
  }
  template<typename value_type>
  any(const value_type &value)
    : content(new holder<value_type>(value))
  {
  }
  any &swap(any &rhs)
  {
      std::swap(content, rhs.content);
      return *this;
  }
  any &operator=(const any &rhs)
  {
      return swap(any(rhs));
  }
  template<typename value_type>
  any &operator=(const value_type &rhs)
  {
      return swap(any(rhs));
  }
  ...
};
```

ceptually similar types substitutable, emphasizing their commonality, and allowing an existing type to be used where a new one is expected. For instance, `string` and `char *` are each different realizations of the concept of a character string. They are not perfect substitutes for one another, but there is an implied level of equivalence in meaning that should be respected and supported by the developer. Where single constructor arguments are needed, but equivalence does not make sense, the `explicit` keyword should be used. For instance, a file object may be initialized from a string representing its pathname, but it cannot be considered a realization of strings.

A degenerate form of conversion is the identity conversion, i.e., where an instance of a type can be converted into another instance of the same type. The copy constructor and assignment operator express this concept. An overloaded assignment operator can be used to optimize any use of a converting constructor followed by a copy assignment. For a string class, this means:

```
class string
{
public:
  string(const char *);
  string(const string &);
  string &operator=(const string &);
  string &operator=(const char *);
  ...
};
```

Providing a converting constructor also provides the developer with a cast form for a type. It is not possible to define a literal form for a new type, but the constructor expression syntax comes close, e.g., `string("theory")`. This is stylistically preferable to using `static_cast`, as the conversions are well-defined—as opposed to a potentially dangerous conversion that must be highlighted in the source code—and corresponds well to the idea of constructing a new value. The preferred "constructor-literal" style also means that code appears consistent when used with other multiple argument constructed forms, e.g., `string(5, '*')`.

Unionization The `any` class can be fleshed out further by considering what inward conversions it is reasonable to support (see Listing 2). Certainly, copying one `any` to another by construction or by assignment is essential for any value class. The copy constructor takes advantage of the representation's `clone` function to perform polymorphic copying, and a nonthrowing `swap` function allows for an exception and a self-assignment-safe copy assignment operator.[4,12]

Employing the member template mechanism supports implicit conversion from values of an arbitrary type into an `any`. This is used in the converting constructor and the templated assignment operator, allowing values of any type to be used where an `any` is expected.

OUTWARD CONVERSIONS

An implicit conversion from another type into a type we are defining can be provided through a user-defined conversion operator (UDC). However, UDCs should be treated with some caution; they are typically far less appropriate than a corresponding inward conversion. For instance, although a `const char *` can be reasonably passed where a `string` object is expected, the converse is not true:

```
class string
{
public:
  ...
  operator const char *() const;
  ...
};
```

Because of the lifetime of temporary objects, the following would result in undefined behavior:

```
string prefix, suffix;
...
const char *whole = prefix + suffix;
cout << whole << endl;
```

This is the reason that `std::string` does not support such a conversion.

Truth and Beauty Whereas a conversion from any type into an `any` type is widening, and therefore always safe, a conversion outward is narrowing, and therefore potentially unsafe—all types can be used where an `any` is expected, but not vice-versa. Alas, the absence of `explicit` UDCs in the language means we cannot retain uniform usage syntax for casts while also preserving the constraint of explicitness. We must resort to a more conservative approach, such as the named `to_ptr` member template function (see Listing 3).

There is one query, however, that may be conveniently expressed through a UDC: Does an `any` hold a value? For many classes this immediately translates to `operator bool`. However, in many cases it turns out that `bool` is not the safest realization of a Boolean type. It introduces a number of subtle conversion problems for many classes, such as smart pointers[13] or

Listing 3. Functions to extract the value from the generalized union type.

```
class any
{
public:
  ...
  operator const void *() const
  {
      return content;
  }
  template<typename value_type>
  bool copy_to(value_type &value) const
  {
      const value_type *copyable =
        to_ptr<value_type>();
      if(copyable)
          value = *copyable;
      return copyable;
  }
  template<typename value_type>
  const value_type *to_ptr() const
  {
      return type_info() == typeid(value_type)
          ? &static_cast<
            holder<value_type> *>(content)->held
          : 0;
  }
  ...
};
template<typename value_type>
value_type any_cast(const any &operand)
{
  const value_type *result =
    operand.to_ptr<value_type>();
  return result ? *result : throw std::bad_cast();
}
```

IOStreams, which at one stage in their standardization sported such an operator. These problems stem typically from `bool`'s underlying integer nature: Its eagerness to participate in all kinds of (surprising) arithmetic and comparison. In contrast to `bool`, a `const void *` is positively hermitlike in its interactions with other types and operators.

CUSTOM KEYWORD CASTS How can an explicit outward conversion be provided for a type, or for a conversion between two existing types using a particular conversion method not already implemented by either type? The omission of explicit UDCs from the language closes one avenue, but the inclusion of templates and, in particular, explicit template function qualification opens another.

The keyword casts—e.g., `dynamic_cast`—are templatelike in appearance. It is possible to emulate them with template functions, idiomatically defining new custom keyword casts that provide new kinds of named, explicit conversion.[14,15] For instance, the following offers a simple approach for converting between any two types that support streaming:

```
template<typename result_type, typename arg_type>
result_type interpret_cast(const arg_type &arg)
{
  std::stringstream interpreter;
  interpreter <<arg;
  result_type result = result_type();
  interpreter >> result;
  return result;
}
```

This makes scriptlike interpretation of values a convenience in C++, e.g.:

```
string forty = interpret_cast<string>(40);
int two = interpret_cast<int>("2");
```

Cast out of the Union Based on the `to_ptr` member template, it is possible to provide a checking cast, `any_cast`, that may be used to extract values of a particular type from an `any` (see Listing 3).

CONCLUSION Money is a mechanism. As parents, partners, and both public and private enterprise will recognize, understanding the mechanism does not necessarily impart wisdom as to its best use. The same can be said of C++'s many features: Knowledge of denomination does not necessarily settle design issues. Principles and practices associated with conceptually organizing features into a more coherent whole can assist the programmer.

References

1. Liskov, B. "Data Abstraction and Hierarchy," *OOPSLA '87 Addendum to the Proceedings*, Oct. 1987.
2. Coplien, J. O. *Advanced C++: Programming Styles and Idioms*, Addison–Wesley, Reading, MA, 1992.
3. Henney, K. "From Mechanism to Method: Substitutability," *C++ Report*, 12(5): 28–30, May 2000.
4. Sutter, H. *Exceptional C++*, Addison–Wesley, 2000.
5. Meyers, S. "How Non-Member Functions Improve Encapsulation," *C/C++ Users Journal*, Feb. 2000.
6. Bäumer, D. et al. "Values in Object Systems," *Ubilab Technical Report 98.10.1*, 1998.
7. Coplien, J. O. "C++ Idioms," *Pattern Languages of Program Design 4*, N. Harrison, B. Foote, and H. Rohnert, Eds., Addison-Wesley, Reading, MA, 2000.
8. Sommerlad, P. and M. Rüedi. "Do-It-Yourself Reflection," *Proceedings of the 3rd European Conference of Pattern Languages of Programming and Computing 1998*, J. Coldeway and P. Dyson, Eds., 1999.
9. Gamma, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison–Wesley, 1995.
10. International Standard. Programming Language—C++, ISO/IEC 14882:1998(E), 1998.
11. Cleeland, C., D. C. Schmidt, and T. Harrison. "External Polymorphism," *Pattern Languages of Program Design 3*, R. Martin, D. Riehle, and F. Buschmann, Eds., Addison–Wesley, 1998.
12. Henney, K. "Creating Stable Assignments," *C++ Report*, 10(6): 25–30, June 1998.
13. Meyers, S. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Addison–Wesley, 1996.
14. Stroustrup, B. *C++ Programming Language*, 3rd ed., Addison–Wesley, 1997.
15. Boost Library Website, http://www.boost.org.