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

# Substitutability

IF I WERE TO SAY that C++ is a language with a large number of features, it is unlikely that I would be in much danger of igniting a flame war. Where fact gives way to opinion—and sometimes to the holiest and most heated of wars—is in the recommendation of how to work with these features. What is their purpose? What concepts do they express? In short, what methods give sense to the mechanisms, elevating the programmer's view of the language from a shopping list of features to recipes that marshal features into designs? Sure, programmers need knowledge of syntax and semantics to make any headway with the language, but this learning curve quickly levels out to a plateau with the sheer face of design rising above it.

Idioms offer a steady and complementary companion route—patterns that capture an understanding of problem and solution married together at the level of the language.[1,2] Idioms establish a vocabulary for communicating and formulating design, where "design is the activity of aligning the structure of the application analysis with the available structures of the solution domain."[3]

On the other hand, while practices based on accepted wisdom generate comfortable habits that generally simplify day-to-day development, this is no reason to be complacent. Occasionally revisiting old maxims can reveal other useful insights that were hiding in the shadows.[4]

**IS-A IS NOT ENOUGH** Attractive as the idea first seems, the English language does not offer consistently useful guidance for the analysis of problems and the statement of software structure. Excitement about nouns as objects and verbs as methods often wears off when the nouning of verbs and verbing of nouns is recognized as common usage.

So it is with the common good practice recommendation guiding the use of public inheritance: that it should model an *is-a* relationship rather than ad hoc code reuse. For example, a save dialog is a dialog box and a list is a sequence, whereas a bounded list is not an unbounded list and vice versa. The view is that reuse-(over)driven approaches lead to the kind of spaghetti inheritance hierarchies that give inheritance a bad name, and the *is-a* approach constrains inheritance hierarchies to more closely model expected classifications.

However, like all analogies, it has a breaking point beyond which rhyme and reason denatures to protracted theological debate around the coffee machine. For instance, Rex is a dog, and although his owner might regard him in a class of his own, it is more likely that developers would view this as an *instance-of* relationship. Preferring *is-a-kind-of* to *is-a* steers us clear of many of the vagaries of natural language, but not so clear that it clarifies how state models are inherited or how a virtual function should be overridden. For example, can a function that in the base class is guaranteed to accommodate null pointer arguments and never return a null pointer be sensibly overridden to accept only non-null arguments and return null pointers? The linguistic view does not provide a useful map through such territory.

**Liskov Substitution Principle** The Liskov Substitution Principle (LSP)[1,5] is often cited as giving more detailed guidance on the use of inheritance. It makes a clear separation between type—described in terms of the behavior of operations—and class—the realization of the behavior in programmatic detail. LSP then establishes the criteria for subtyping relationships in terms of conformant behavior[5]:

> *A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a subtype is one whose objects provide all the behavior of objects of another type (the supertype) plus something extra. What is wanted here is something like the following substitution property: If for each object $o_1$ of type S there is an object $o_2$ of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when $o_1$ is substituted for $o_2$, then S is a subtype of T.*

So, in answer to the previous question: No, a function cannot sensibly be overridden to strengthen the requirements on its arguments and weaken the requirements on its return type. This same recommendation can be found echoed in Design by Contract.[6]

**Polymorphic Polymorphism** LSP is normally presented as an inheritance guideline, but taking a step back we can see that it need not be so narrow: It is a relationship about types, not implemen-

**Table 1. Different kinds of polymorphism.**

| Polymorphism | Description |
|---|---|
| **Inclusion** | Conventional OO model of polymorphism, i.e., virtual functions |
| **Parametric** | Based on provision or deduction of extra type information that determines the types used in a function, i.e., templates |
| **Overloading** | Based on the use of the same name denoting different functions, with selection based on the context of use, i.e., argument number and type |
| **Coercion** | Based on the conversion of an object of one type to an object of another type based on its context of use |

tations—i.e., subtyping, not subclassing. LSP relies on polymorphism; the concept of structural relationships, such as public inheritance, need not enter into it.

C++ developers have "inherited" from other OO languages and literature the notion that polymorphism concerns only inheritance and virtual functions. Deserving of its name, polymorphism manifests itself in many forms (see Table 1). This classification[7] gives us a broader and more liberated view of type relationships. Constraining C++ solely to OO usage ignores its other strengths and denies its multiparadigm nature.

**TYPES OF SUBSTITUTABILITY** Substitutability is the property that different elements have if they are grouped—and then treated—in terms of their commonality; variations that are not relevant to their use are not a part of this abstraction, or rather they do not dominate.[3] Building on the previous discussion allows us to think outside the box, giving a broader view of substitutability grounded in C++'s mechanisms (see Table 2).

The substitutability types are neither perfect peers nor perfectly hierarchical; they overlap and build on each other. Nonetheless they offer a useful way to understand and reason about features. There is a close correspondence between these substitutability types and the polymorphism categories described in Table 1.

**Conversions** Conversions may be implicit or explicit, which places them under the control of the compiler or developer, respectively. Whether a conversion should be implicit or explicit is a matter of taste, safety, and requirement. Widening conversions—from a specific to a general type—are always safe and can be implicit without offending sensibilities or the compiler, e.g., int to double or derived to base. Narrowing conversions—from a general to a specific type—are not guaranteed to be safe and should be explicit.

One would hope that narrowing conversions would be required to be explicit, but this is not always the case, e.g., double to int. Even though the compiler does not require it, one might argue that taste does. Where possible, narrowing conversions should be checked, e.g., the use of dynamic_cast to go from base to derived.

Developers need to consider the interoperability of new and existing types. Particularly for value types—i.e., for fine-grained objects such as strings, which express quantities, rather than persistent or strongly behavioral objects—the use of conversions to and from other types makes more sense than the use of inheritance relationships.

A single argument constructor is also, by default, a converting constructor. Where a type may be safely, sensibly, and easily used in place of another type, an implicit conversion into that type either by a converting constructor or a user-defined conversion (UDC) operator may be justified, e.g., use of const char * in many places that a string may be used:

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

class file
{
public:
   explicit file(const string &);
   ...
};
...
const char *const log_name = getenv("LOGFILENAME");
file log(log_name ? log_name : "default.log");
```

Otherwise—and this applies especially to UDCs—single argument constructors should be declared explicit and UDCs left unwritten, e.g., a string may not be sensibly used in most of the places a file object is expected, nor may a string always be used safely where a const char * is expected:

```
class string
{
public:
   ...
   operator const char *( ) const; // questionable
   ...
};
string operator+(const string &, const string &);
...
const string log_suffix = ".log";
const char *const log_name = getenv("LOGBASENAME") + log_suffix;
file log(log_name);
```

**Overloading** Overloading is based on the idea that a common name implies a common purpose, which frees programmers from indulging in their own name-mangling to differentiate similar functions (this is the job of the compiler). Overloading works

**Table 2. 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 |

closely with—and sometimes against—conversions. Developers are cautioned to keep any eye open for any such interference.

Operators provide a predefined set of names and definitions, and therefore expectations: Overloading operator-> suggests the smart pointer idiom, and overloading operator( ) suggests the function object idiom. Although compilers do not run through code checking it for stylistic content (e.g., use of meaningful variable names, sensible use of overloading, and conformant use of inheritance) these conventions derive from the language itself to establish a common frame for working with the language "when in doubt, do as the ints do."[8]

Extension through overloading forms an important part of operator overloading–based substitutability. For instance, a class that is otherwise closed to change can apparently be extended to work within a framework, e.g., "extending" iostreams to understand new types.

Some extension is less transparent, but it is important that it follow as much of the base form as possible. An obvious example is the use of placement new operators, which, in spite of taking additional arguments, have the same purpose and much of the same form as the vanilla new. Tagged overloads, such as new(std::nothrow), provide a means of compile-time selection that is a compromise between operators and named functions.

**Derivation** Substitutability with respect to derivation is perhaps the most familiar category for programmers coming to C++ with a conventional OO angle. A public class interface establishes a behavioral contract that the user relies on; derivation through public inheritance tells the compiler that the types are interchangeable, which only makes sense if the derived class developer promises to still fulfill the contract, even if it is extended in some way. Dropping LSP can lead to surprising behavior.

In terms of coupling, derivation is also the strongest relationship one can have in a C++ system: It defines a dependency that is strongly physical, e.g., with respect to #include, as well as one that is strongly logical. The combined effect can lead to a tsunami-like recompilation of a whole system, rather than a gentle ripple effect, whenever the slightest change is made. Although C++ does not clearly separate the concepts of subtyping and subclassing in its inheritance mechanism, it is possible to effect this with interface classes,[9] i.e., abstract classes containing only pure virtual functions. This offers a vehicle for substitutability independent of representation issues. Interestingly, this means that as a mechanism, inheritance may be used to either significantly increase or significantly decrease the coupling in a system.

**Mutability** Mutability is concerned with the effects of change. For objects in C++ this means qualification: const and volatile. ...Well, const if we are being really honest with ourselves. From this perspective every class has two public interfaces: the interface for const qualified objects and the interface for non-const qualified objects. The const interface is effectively a subset of the non-const interface, and therefore a non-const object may be used wherever a const one is expected; i.e., the qualified interface may be considered a supertype of the unqualified one. Note that the subtyping relationship need not be strict: Overloaded functions differing only in const or non-const will be selected according to the calling const-ness. In OO terms this can be considered a form of compile-time overriding of the const function by the non-const variant, typically to return something more specific:

```
class string
{
public:
    char operator[](size_t) const;
    char &operator[](size_t); // 'overrides' const version
    ...
};
...
const string read_only = ...;
cout << read_only[0];
string read_write = ...;
cin >> read_write[0];
cout << read_write[0];
```

**Genericity** Templates offer a route to substitutability that grows out of the basic concept of overloading. A templated function can be considered to have a potentially infinite number of overloads all sharing a common name, purpose, and structure, but differing in their parameter types. A similar view can be extended to template classes, and from there, to member templates.

Generic programming, as typified by the part of the standard library derived from the STL, is based on the compile-time polymorphism of C++ templates, as well as the concepts and mechanisms of the other substitutability categories.

**CONCLUSION** Substitutability provides a useful way to structure a system's meaning. It can be recognized as reaching further than a commonplace recommendation for inheritance, drawing together many C++ features in a set of practices that make sense of mechanism.

Future columns will explore the different types of substitutability in greater detail, looking at language features and their use in idioms, and highlighting the practices that commonly (or uncommonly) appear in application and library code. ☾

**References**

1. Coplien, J. *Advanced C++: Programming Styles and Idioms*, Addison–Wesley, Reading, MA, 1992.
2. Coplien, J. *Software Patterns*, SIGS, New York, 1996.
3. Coplien, J. *Multi-Paradigm Design for C++*, Addison–Wesley, Reading, MA, 1999.
4. Henney, K. "Creating Stable Assignments," *C++ Report*, 10(6): 25–30, June 1998.
5. Liskov, B. "Data Abstraction and Hierarchy," *OOPSLA '87 Addendum to the Proceedings*, Oct. 1987.
6. Meyer, B. *Object-Oriented Software Construction*, 2nd ed., Prentice–Hall, Englewood Cliffs, NJ, 1997.
7. Cardelli, L. and P. Wegner. "On Understanding Types, Data Abstraction, and Polymorphism," *Computing Surveys*, 17(4): 471–522, Dec. 1985.
8. Meyers, S. *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Addison–Wesley, Reading, MA, 1996.
9. Carroll, M. D. and M. A. Ellis. *Designing and Coding Reusable C++*, Addison–Wesley, Reading, MA, 1995.