Addendum to OOPSLA95 Proceedings, ACM OOPS Messenger, October 1995

Why C++ is not just an Object-Oriented Programming Language

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

C++ directly supports a variety of programming styles. In this, C++ deliberately differs from languages designed to support a single way of writing programs. This paper briefly presents key programming styles directly supported by C++ and argues that the support for multiple styles is one of its major strengths. The styles presented include: traditional C-style, concrete classes, abstract classes, traditional class hierarchies, abstract classes and class hierarchies, and generic programming. To provide a context for this overview, I discuss criteria for a reasonable and useful definition of ‘‘object-oriented programming.’’

1 Introduction

There are many tools and techniques that can help in our effort to build useful, economical, and maintainable systems. To complete ambitious and complex projects, we rely on a wide variety of techniques and tools that must work together.

The title of this paper singles out a programming language†. However, the real topic is programming, or if you prefer a longer formulation, the design and implementation of systems. A programming language is just one of the means by which we try to achieve our goals.

The definition of ‘‘object-oriented programming’’ is no longer a popular topic of discussion at major conferences. A practical definition of ‘‘object-oriented programming,’’ ‘‘object-oriented analysis,’’ ‘‘object-oriented design,’’ ‘‘object-oriented technology,’’ etc., is, however, a burning issue for people who want to turn the oft-repeated promises made for techniques and languages called ‘‘object-oriented’’ into reality in everyday projects. It has become a practical rather than academic topic of discussion. What is ‘‘object-oriented technology?,’’ what benefits can be expected from it? at what risks?, how do those techniques, benefits, and risks compare with those associated with alternatives?

A systems builder trying to explain to an accountant why money should be spent for tools supporting object-oriented techniques needs more than a statement to the effect that ‘‘object-oriented is great’’ or that ‘‘really great techniques are really object-oriented.’’ You simply cannot ask someone to bet their company’s future on vague promises phased in ill-defined terms. Nor is a well-polished and logically coherent semi-mathematical treatment of the subject of direct practical use.

We need to define ‘‘object-oriented’’ to be something specific so that we can point out specific benefits and risks associated with its use. We must also be specific about what is *not* object-oriented, and what benefits and lack of benefits we can expect from various non-object-oriented techniques.

Consequently, this paper starts out discussing what makes a good definition of ‘‘object-oriented.’’ Next, I present a range of useful techniques which may or may not be object oriented and discuss their advantages and disadvantages.

2 Defining ‘‘Object-oriented’’

To be useful and intellectually honest, a definition of ‘‘object-oriented’’ must

[1] not be a mere synonym for ‘‘good,’’
[2] not exclude most accepted meanings,
[3] have a firm historical basis,
[4] exclude something.

Not everything good is object- oriented, and not everything object-oriented is good. I think I can support both claims from experience. I have seen examples of the latter often enough: it is not

† This paper is primarily based on an invited talk with the same title given at OOPSLA’95 in Austin Texas. The style of this paper is clearly affected by its origins as a relatively short talk. I would have preferred this paper to be either much longer or much shorter, but I did not have the time to do either.

uncommon to find programs that apply techniques usually deemed object-oriented extensively or even exclusively, yet are hard to comprehend, hard to maintain, and perform abysmally. Such examples occur in every programming language. But then, of course, some people respond ‘‘that just proves that the program wasn’t *truly* object-oriented.’’ To which the answer must be that either the term has become meaningless or there must be something good beyond what is called ‘‘object-oriented.’’

On the other hand, when we define ‘‘object-oriented,’’ we must not be too exclusive. Object-oriented programming is a broad intellectual discipline, not the mere use of specific language features. Attempts to define ‘‘object-oriented’’ to mean ‘‘what I’m selling’’ are not uncommon, but are fundamentally sleazy.

Any definition of ‘‘object-oriented’’ should be historically reasonable. Words are only useful for communication, really only mean something, if we agree on a meaning for them. There are several plausible, logically coherent, and mutually contradictory definitions of ‘‘object oriented’’ in use. However, the mainstream usage stems directly from the ideas pioneered by programming language Simula and the design techniques it was developed to support. The communities of programmers and designers centered around languages such as C++, CLOS, Eiffel, Object Pascal, and Smalltalk have contributed much to this tradition.

A meaningful definition of any concept must exclude something.

3 A Broad Definition of ‘‘Object-oriented’’

Given these general criteria for a definition of ’’object-oriented’’ you can find several plausible candidates, and several communities have their own definitions. However, I suggest we stick to the traditional definition of object-oriented used within broad communities of programmers. A language or technique is object-oriented if and only if it directly supports:

[1] Abstraction − providing some form of classes and objects.
[2] Inheritance − providing the ability to build new abstractions out of existing ones.
[3] Run-time polymorphism − providing some form of run-time binding.

This definition includes all major languages commonly referred to as object-oriented: Ada95, Beta, C++, CLOS, Eiffel, Simula, Smalltalk, and many other languages fit this definition. Classical programming languages without classes, such as C, Fortran4, and Pascal, are excluded. Languages that lack direct support for inheritance or run-time binding, such as Ada88 and ML are also excluded.

ML is a good example of something that is good but not object-oriented. I like ML; it is an interesting, innovative, and powerful language, but it is functional rather than object-oriented and its polymorphism is resolved at compile time rather than at run-time. Thus, saying that ML isn’t object-oriented is not a criticism, it’s an observation about definitions and the nature of ML.

Techniques and tools are object-oriented if and only if they support the use of object-oriented programming. For example, a design method is object-oriented if its regular and proper use leads to programs that exploit abstraction, inheritance, and polymorphism where appropriate. I strongly prefer design methods that directly and naturally support the use of at least one of the major object-oriented languages supporting in ways that exploit its features in an idiomatic way.

For example, it is often possible to simplify application code by hiding objects with different representations and different implementation details behind a common ‘‘abstract’’ interface (see §6.4 and §6.6). Conversely, the implementation of related concepts can often be greatly simplified by exploiting commonality through inheritance (see §6.5 and §6.6). A major purpose of design methods and the CASE tools that commonly support them is to make design simpler, more regular, and more predictable. Thus, to earn the label ‘‘object-oriented,’’ a design method must regularly and predictably help the discovery of commonality that can be exploited in these ways. Ideally, an object-oriented design method must strongly encourage the expression of this commonality using the most appropriate facilities in one or more of the languages supporting object-oriented programming. Minimally, the method and its supporting tools must not be a hindrance to the use of object-oriented facilities in the programming language used to implement the design. Much confusion arise because not every design method that claims to be object-oriented does that.

Please remember that I’m looking for a practical understanding of the notion of ‘‘object-oriented’’ rather than a formal definition. A formal definition is useful, indeed it may be essential. However, to be relevant, a formal definition must match a coherent view of what the formal

definition is meant to specify precisely.

4 Purity

There has been much debate about ‘‘purity,’’ in the context of languages supporting object-oriented programming. In my opinion, much of that discussion is confused by the − often unstated − assumption that not only does ‘‘object-oriented’’ imply ’’good,’’ but also by the further assumption that only ‘‘object-oriented’’ features are good. Consequently, it is − wrongly − assumed that a language that provides features deemed non-object-oriented must be worse than a language that does not. People who like a language to support classes as part of a hierarchy only and functions/methods attached to one specific class only often call such a language ‘‘a pure object-oriented language.’’ If we don’t like the idea of restricting the definition of classes and functions that way we can call such a language ‘‘just an object-oriented programming language.’’

I prefer to have more facilities available than can be provided by methods defined on classes within a single hierarchy. A lot of good design goes beyond that relatively narrow domain. Incidentally, I have come to dislike the adjective ‘‘hybrid’’ as used to distinguish ‘‘pure’’ object-oriented systems from others. Too often, ‘‘hybrid’’ is used in a prejudicial manner. If I must apply a descriptive label, I use the phrase ‘‘multi-paradigm language’’ to describe C++.

4.1 Use of Language Features

Even when all the features required to support object-oriented programming are available, you don’t need to use them all the time. Some classes just don’t belong in a hierarchy and some functions don’t belong to any particular object.

The key to maintainable, efficient, and evolvable programs isn’t particular language features. It is the ability to develop concepts needed for a solution and to express them clearly in a program. Language features exist to make such expression simple and direct.

Object-oriented programming can be done in a language lacking one or more of the features required to directly support object-oriented programming. However, doing so is unnecessarily difficult, very difficult to support with tools, and often prohibitively expensive.

Furthermore, there are things that can’t be expressed directly using only the ‘‘pure’’ object-oriented constructs mentioned above. For example, some entities belong together, but their relationships are not hierarchal. Some entities simply do not obey the rules of a particular object-oriented language. Some things that you build in an object-oriented world are manipulated from the outside so that it is difficult to make guarantees about the way they are used.

5 C++ Design Ideals

I felt the need for facilities outside what is conventionally called ‘‘object-oriented,’’ so I supplied some in C++. However, C++ isn’t meant to be everything to everybody. No one programming language and no one view of how to write programs is sufficient for everything. Constraints-based programming, logic programming, functional programming, and various forms of concurrent programming are examples of good and useful styles of programming not supported by C++.

No single language can support every style. However, a variety of styles can be supported within the framework of a single language. Where this can be done, signifcant benefits arise from sharing a common type system, a common toolset, etc. These technical advantages translate into important practical benefits such as enabling groups with moderately differing needs to share a language rather than having to apply a number of specialized languages.

C++ was designed to support a range of styles that I considered fundamentally good and useful. Whether they were object-oriented, and in which sense of the word, was either irrelevant or a minor concern:

[1] Abstraction − the ability to represent concepts directly in a program and hide incidental details behind well-defined interfaces − is the key to every flexible and comprehensible system of any significant size.

[2] Encapsulation − the ability to provide guarantees that an abstraction is used only according to its specification − is crucial to defend abstractions against corruption.

[3] Polymorphism − the ability to provide the same interface to objects with differing implementations − is crucial to simplify code using abstractions.

[4] Inheritance − the ability to compose new abstractions from existing one − is one of the most powerful ways of constructing useful abstractions.

[5] Genericity − the ability to parameterize types and functions by types and values −

is essential for expressing type-safe containers and a powerful tool for expressing general algorithms.

[6] Coexistence with other languages and systems − essential for functioning in real-world execution environments.

[7] Run-time compactness and speed − essential for classical systems programming.

[8] Static type safety − an integral property of languages of the family to which C++ belongs and valuable both for guaranteeing properties of a design and for providing run-time and space efficiency.

These facilities and general properties can be supported in several alternative ways. For example, one programming language may support a facility in its core language where another supports it in a library. Similarly, a facility provided by a run-time mechanism in one language may be provided by a compile-time mechanism in another.

The requirement for coexistence is essential for any language claiming to be general-purpose. Looking at the world from the perspective of a given programming language, we find that almost every real-life system contain parts that are written in others languages and designed according to principles foreign to that language. To be general-purpose, a language must somehow take the unpredictable, ugly, and constantly changing demands of program fragments written in ‘‘other languages’’ into account.

To be genuinely general purpose, a language must possess facilities that allow it to share data with program fragments written in other languages, to invoke code fragments written in other languages, and have code invoked by code written in other languages. For example, systems relying on callbacks can be rather ugly to program, but not being able to use such systems in a direct and idiomatic way would be crippling for a language as a tool for real-world programming. In many languages, a common use of ‘‘foreign’’ code is exactly to violate the languages rules: to do things that can’t be done − or can’t be done efficiently − in the language itself.

Alternatively, access to facilities in ’’the outside world’’ could be carefully fitted into the framework of the object-oriented programming language through special facilities in the run-time environment or in libraries. However, accessing facilities in the manner they were meant to be used is often easier and less awkward than to fit them into our language framework. A general mechanism for accessing ‘‘foreign’’ code also leads to more extensible systems than a requirement to fit each individual ‘‘foreign’’ facility into the language framework.

Over the years, we have seen spectacular improvements both in hardware performance and in compilation techniques. However, run-time efficiency and compact representation is still absolutely essential to many people.

Static type safety is an essential part in my view of both design and implementation (see, for example [Stroustrup,1991]). The guarantees provided and the discipline of design imposed have been found extremely valuable by many people working in a wide range of application areas. Static type checking is of course not a panacea, but it is something I would not attempt major projects without.

The fundamental ideal of C++ is actually the fundamental ideal for a lot of languages:

center box; c. Represent concepts and relationships between concepts directly and affordably.

Naturally, there are many ways of approaching this ideal. It is worth remembering that all of the languages usually mentioned in a discussion of practical use of object-oriented techniques are suitable vehicles for good design. A rational discussion of languages is one of relative merits, applicability to specific problem areas, and personal preferences, rather than one of absolutes.

Representing concepts directly is a restatement and possibly a generalization of ideas relating to data abstraction and information hiding. Representing hierachical relationships is the traditional key to object-oriented programming. There are, however, clean and useful relationships that are not hierarchical yet can still be represented directly in a program (for example, see §6.3 and §6.7).

Being more concerned with producing good software than with finding the most elegant expression of ideas in the abstract, I insist on affordability. Affordability is a multi-facetted issue that involves not only run-time efficiency, but also availability of suitable hardware, availability of designers and programmers comfortable with new techniques, etc.

6 Programming Style and Language Features

I will now give examples of programming styles and language features supporting them. Some are commonly referred to as ‘‘object-oriented,’’ some are not, but that doesn’t prevent me from recommending them in some contexts.

6.1 Conventional Notation

There are aspects of conventional code and conventional notation that I would like to see maintained even in a strictly object-oriented overall design. Being able to say plain square root of two, `sqrt(2)` is nice, and so is the ability to write `x+y*z` and know that it means add `x` to the product of `y` and `z`. We have about 400 years of experience with such notation and it is deeply ingrained in our technical culture.

6.2 Concrete Types

Very simple concepts, such as integers, floating point numbers, complex numbers, points, lines, pairs, dates, disk locations, bcd characters, error messages, currency, are usually not considered suitable topics for discussion in academic articles or at conferences. These are usually considered too simple to merit discussion. However, the mundane is often statistically more significant than the sophisticated.

Provided they can be implemented in a way that is simple, elegant, efficient and flexible enough, I consider such simple concepts excellent candidates for independent proper types − as opposed to presenting them to users as plain data structures or as parts of a larger class hierarchy. Consequently, part of a design effort should focus on these little abstractions. These very concrete types should be designed carefully and supported well.

To illustrate why, I'll contrast this approach to the use of a plain data structure and to the use of class hierarchies. To make this discussion concrete, I'll use the example of a date.

6.2.1 Structures and Functions

The simplest way of presenting a date in a program is simply to specify its data layout. For example:

```
struct date {
    // representation
};
```

Given that, programmers can do anything at all with `date`s. That ''anything at all'' is the strength and weakness of this idea. Naturally, a ''standard'' set of functions is usually provided to manipulate a structure such as `date`. However, such a set of functions is rarely complete, and even when it is, programmers find reasons to manipulate `date`s directly. Consequently, it is usually not possible to change the definition of `date` after the initial release of the software; it is simply too difficult to track down every use of a `date` and modify it use to the new definition.

The reason that the set of functions is rarely complete is that there is no incentive to make it so. A programmer can always write new functions accessing the `date` structure, and the dominant culture encourages the programmer to do so. Writing a new function that is ''just right'' for the job, carries no overhead, and relies on no potentially untrustworthy code is often considered better than improving a standard and general-purpose set of access functions and using it. Often, it is also far easier. This trend is typically reinforced by poor documentation.

6.2.2 A Concrete Class

A simple `Date` type can remedy most of the problems related to using a data structure directly. Consider:

```
class Date {
public:
    // public interface, consisting
    // of non-virtual functions
private:
    // representation and other
    // implementation details
};
```

Such a `Date` type will provide

[1] constructors specifying how objects of the type are to be initialized;

[2] functions for examining a `Date`; these functions will be explicitly declared not to modify the value of the object;

[3] functions for manipulating `Date`s without actually having to know the details of the representation or fiddle with the intricacies of the implementation.

In addition, `Date`s can be freely copied.

This set of member functions supplied as members of `Date` should be those that provide a basic semantics for a `Date` and also requires direct access to the representation of `Date` to be implemented.

The set of member functions should be almost minimal; many operations that users would find convenient can be supplied separately (see §6.3). I dislike classes with dozens or even hundreds of member functions. Such a class does not represent a well-thought-out concept; it's a glorified data structure produced by somebody who couldn't decide on what was really wanted.

The member functions are declared non-virtual to ensure that there is no time or space overheads involved in using this `Date`, and to ensure that the semantics of `Date` cannot be

modified later. Similarly, the representation of `Date` is declared `private` to prevent access by any function not explicitly mentioned in the class itself.

The representation of a concrete type should be compact. Sometimes millions of objects of such classes exist, and even with modern memory sizes space overheads can be a burden. If nothing else, reading and writing objects with bloated representations can be a nuisance.

The use of concrete types must be fast. In my world at least, programmers are prone to represent something as plain data structure out of fear of overheads supposedly associated with abstractions.

There are no time or space overheads associated with the `Date` class as defined above. The size is identical to that of the plain `date` structure, and inlining is done for simple member functions to make these as fast as the code a programmer would write accessing a plain structure directly.

Often, it is important that simple concrete types, such as `Date`, be layout-compatible with simple data structures, such as `date`, as used in traditional languages. This allows simple exchange or sharing of information with code written in traditional languages. This can be a major convenience if your operating system, your database, or your high-performance numeric library is written in a traditional language and requires manipulation of data of a specific layout.

The `Date` class is very simple and very basic. It requires no elaborate framework, no class hierarchies, no clever dispatcher to mediate access, etc. It doesn't affect the overall structure of a program much; it just provides a lot of help at the detailed programming level − below the level of detail of interest to most managers and to many designers.

If such types are that simple, why bother spending time on them?

[1] The concepts best represented by such simple types are common; most applications can use a few dozen or a few hundred such types. Thus, any benefits we get from a single concrete type, we get many times over.

[2] The problems relating to lack of encapsulation of plain structures (§6.2.1) are eliminated.

[3] The replication of effort writing simple access functions is eliminated.

[4] Making a concrete type the subject of a conscious design effort typically results in a better thought out, more comprehensive, and better documented concept. In principle, this could equally well be done for the plain structure approach, but in practice that typically doesn't happen.

[5] Writing the basic functions of a concrete type is not difficult, but it is not trivial either. For example, adding a year to a date requires us to handle leap years. By relying on common access functions, we eventually achieve an implementation that has been better thought out and has fewer errors.

[6] Since more of the implementation is documented and shared, user code becomes more uniform. Thus, code written by others become easier to comprehend.

These are classical reuse benefits and I don't think we should decline them just because they are easy to obtain.

6.2.3 Re-using Concrete Types

For many concrete types, derivation doesn't make sense. Consider, deriving a new class from `Date`:

```
class MyDate : public Date {
    // …
};
```

Is it ever valid for `MyDate` to be used as a plain `Date`? Well, that depends on what `MyDate` is, but in my experience it is rare to find a concrete type that makes a good base class without modification.

Derivation from a concrete type is almost always a mistake. A concrete type is a self-contained entity that can't easily added to in a way that makes sense. A concrete type is ''reused'' unmodified in the same way as built-in types such as `int` are. For example:

```
class Date_and_time {
public:
    // …
private:
    Date d;
    Time t;
};
```

This form of use (reuse?) is usually simple, effective, and efficient.

Maybe it was a mistake not to design `Date` to be easy to modify through derivation? It is sometimes asserted that *every* class should be open to modification by overriding and by access from derived class member functions. This view leads

to a variant of `Date` along these lines:

```
class Date2 {
public:
    // public interface, consisting
    // primarily of virtual functions
protected:
    // representation and other
    // implementation details
};
```

Here, the functions are declared `virtual`, meaning that a class derived from `Date2` (in the style of `MyDate` above) can provide its own versions. To make it possible to write such overriding functions easily and efficiently, the representation is declared `protected`. A `protected` member of a class is accessible not just to the classes' own members, but also to the member functions of derived classes.

This achieves the objective of making `Date2` arbitrarily mallable by derivation, yet keeping its user interface unchanged. However, there are costs:

[1] Efficiency of basic operations − a C++ virtual function call is a fraction slower than an ordinary function call, virtual functions cannot be inlined as often as non-virtual functions, and a class with virtual functions typically incurs a one word space overhead.

[2] Need to use free store − the aim of `Date2` is to allow objects of different classes derived from `Date2` to be used interchangeably. Because the sizes of these derived classes differ, the obvious thing to do is to allocate them on the free store and access them through pointers or references. Thus, the use of genuine local variables dramatically decreases.

[3] Inconvenience to users − to benefit from the polymorphism provided by the virtual functions, accesses to `Date2`s must be through pointers or references.

Naturally, these costs are not always significant, and as we will see in §6.4 the behavior of a class defined in this way is often exactly what we want. However, for a simple concrete type, such as `Date2`, the costs are unnecessary and can be significant.

Please note that the costs are fundamental; different languages present the facilities differently, but every language that provides run-time polymorphism incur these costs in some way or other.

Finally, a well-designed concrete type is often the ideal representation for a more mallable type. For example:

```
class Date3 {
public:
    // public interface, consisting
    // primarily of virtual functions
protected:
    Date d;
};
```

6.3 Namespaces

Class hierarchies express (hierarchical) relationships, but not every relationship in a program can or should be expressed as a hierarchical relationship between classes. For example, if a class is intented for use only in the context of another class it can be declared a member of that class exactly the way a function can be:

```
class Date {
public:

    enum Month {
        jan, feb, mar,
        apr, may, jun,
        jul, aug, sep,
        oct, nov, dec
    };

    // …
};

Date::Month m = Date::nov;
```

More generally, C++ provides namespaces for grouping declarations [Stroustrup,1994]. For example, many operations on `Date`s shouldn't be members of class `Date` because they don't need direct access to the representation of a `Date`. Providing such functions as non-member functions leads to a cleaner `Date` class, but we would still like to make the association between the functions and the class explicit:

```
namespace Chrono {

    // facilities for dealing with time:

    enum Month {
        // …
    };

    class Date {
        // …
    };
```

```
        int diff(Date a, Date b);
        bool leapyear(int y);
        Date next_weekday(Date d);
        Date next_saturday(Date d);

        // ...
    }
```

A namespace is not a module; it is not an object. A namespace is a general scope mechanism to support a variety of techniques related to modularity. Not incidentally, namespaces provides a way of avoiding name clashes in software composed out of libraries from different suppliers. For example:

```
namespace LibA {
    class String {
        // A-style string
    };
    // ...
}

namespace LibB {
    class String {
        // B-style string
    };
    // ...
}

LibA::String s1 = "Nicholas";
LibB::String s2 = "Annemarie";
```

6.4 Abstract Classes

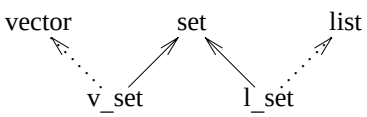
It is possible to completely disassociate implementation and interface. For example, we might implement a set using either an array or a list in such a way that the two kinds of sets can be used interchangeably:

```
class set {
    // ...
};

class v_set : public set,
    private vector {
    // ...
};

class l_set : public set,
    private list {
    // ...
};
```

or graphically:



I use the dotted lines to show that `private` inheritance is an implementation issue that does not affect the interface of the derived class.

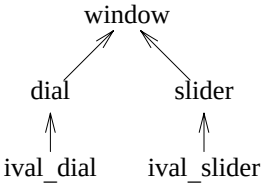
Importantly, a common interface (here, `set`) can be provided long after the design and implementation of implementation classes (here, `vector` and `list`). I find that when people design things, they typically first invent something fairly concrete. They design an array, they invent a list, and only later do they discover an abstraction that covers both in a given context. Using abstract classes as shown above, we use (re-use?) `vector` and `list` without the foresight (and cost) necessary to design them as part of a common hierarchy.

As a matter of fact, you can do this ‘‘late abstraction’’ several times. Say, I want to represent the notion of ‘‘something you could read from.’’ This is a very different abstraction from `set`, yet I can provide such an interface to arbitrary sets as well as for lists, vectors, files, and input streams much in the way I provided `set` as an interface to `vector` and `list`.

Late abstraction using abstract classes allows us to provide different implementations of a concept even when there is no significant similarity between the implementations.

6.5 Classical Hierarchies

Sometimes we do have sufficient foresight to design a classical hierarchy. More importantly, sometimes the various implementations of a concept have a high degree of commonality so that there is significant benefit in organizing these implementations into a hierarchy. For example, consider a class hierarchy that one might find in an application relying on a windows system:



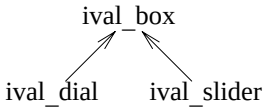
Presumably, the implementations of the application classes `ival_dial` and `ival_slider` are greatly facilitated by code and data provided by the ‘‘system classes’’ `window`, `dial`, and `slider`. That is, you build your code incrementally and your interfaces incrementally.

6.6 Hierarchies and Abstract Classes

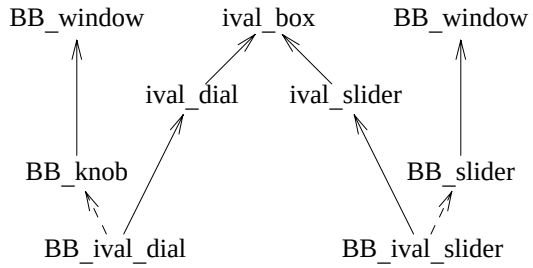
A classical hierarchy is a nice way of providing a variety of related concepts and a nice way of minimizing the effort of building their implementations. However, you do get the classes in the hierarchy tightly coupled. If anything significant in a base class changes, all of the derived classes must change (or at least be recompiled) to match. In particular, any significant change to ''system classes'' at the base of a hierarchy, such as `window`, will affect application classes, such as `ival_dial`. Worse, the choice of a foundation library representing system resources determines the structure of the application class hierarchy and permeates the application code.

There are quite a few ways of dealing with this. For example, some languages have a solution (at its associated costs) mandated, and some implementations of C++ allow major changes to base classes without requiring re-compilation of derived classes. However, here I'll show a solution using abstract classes to make dependencies explicit. Logically, it closely parallels the way the abstract class `set` was used to insulate users from the details of `vector`s and `list`s.

Here, an application hierarchy



is written independently of implementation details and then later tied into an implementation hierarchy without affecting the users of the application hierarchy:



This expresses the design in such a way that the application code becomes independent of any change in the implementation hierarchy.

I have used the `BB` prefix for realism; suppliers of major libraries invariably prepend some identifying initials. In the future, I expect namespaces (§6.3) to be used instead.

The declaration of a class that ties an application class to the implementation hierarchy will look something like this:

```
class BB_ival_slider
    : public ival_slider,
    protected BB_slider {
public:
    // functions overriding ival_slider
    // functions as needed to implement
    // the application concepts
protected:
    // functions overriding BB_slider
    // and BB_window functions as
    // required to conform to user
    // interface standards
private:
    // representation and other
    // implementation details
};
```

This structure assumes that details of what is to be displayed by a windows system is expressed by overriding virtual functions in the `BB_window`s hierarchy. This may not be the ideal organization of a user interface system, but it is not uncommon.

I use protected members and protected inheritance to allow classes derived from `BB_ival_slider` to use information about its implementation.

6.7 Generic Programming

A major theme in the C++ community over the last few years has been the development of techniques exploiting the template mechanism.

6.7.1 Parameterization

Independent concepts should be independently represented and should be combined only when needed.

Where this principle is violated, you either bundle unrelated concepts together or create unnecessary dependencies in the implementation of classes and functions. In particular, fitting weakly related class into a single hierarchy can be a source of unnecessary and problem-causing dependencies.

Consider the concepts of sorting, character, string, and collating sequence. A sort algorithm is independent from the concept of a character. The concept of a string is independent of any particular kind of a character. Finally, the collating sequence which you use when you sort strings of characters is independent of these other three concepts.

This independence can be expressed directly. Here is a string parameterized by the kind of characters contained so that we can make strings of both built-in and user-defined character types:

```
template<class C>
class string {
    // …
};

class Jchar {
    // Japanese characters
};

string<char> s1, s2;

string<unsigned char> us1, us2;

string<Jchar> js1, js2;
```

Independently, we can define the notion of a collating sequence and a string comparison function:

```
template<class C>
class std_coll {
public:
    bool eq(C a, C b)
        { return a==b; }
    bool lt(C a, C b)
        { return a<b; }
};

template<
    class C,
    class Coll = std_coll<C>
        >
int cmp(
    string<C>& s1,
    string<C>& s2
       )
{
    // compare s1 with s2
    // using  Coll::eq and Coll::lt
    // to do character comparisons
}
```

The `cmp` template function takes two template arguments:

- the type of characters in the strings, and
- the collator supplying the character comparison operations.

The ability to pass operations as template parameters is a very powerful expressive mechanism. It is also important for efficiency. For example, it is trivial for a compiler to inline all uses of `eq()` and `lt()`. This can be a significant advantage compared to C where operations can only be passed as pointers to functions so that function call overheads are incurred.

The second template paramenter has a default so we need only specify it if we want a non-standard collating sequence. The first template parameter can usually be deduced from the arguments to `cmp()`. For example:

```
cmp(s1,s2);
cmp(js1,js2);
cmp(us1,us2)
cmp<char,no_case>(s1,s2);
```

Here, `no_case` is a collator defining `eq()` and `lt()` not to be case sensitive.

Typically, the `string` class, the `cmp()` function, the collator classes, and the character classes will be written by different people. Only the final user puts all of the independently developed pieces together.

This style of design relying on templates and additional template arguments (in this example, `Coll`) to express policies, is the basis of much of the C++ standard library. The result is exceptional flexibility and unsurpassed run-time efficiency.

6.7.2 Containers and Algorithms

I want to have algorithms written once and used for objects of many different types. I want these algorithms to run as fast as functions written for a single argument type. I want this to be compile-time checked so that I can be more confident of my code. I don't want to be forced to fit my types into a hierarchy simply to be able to use them for the generic algorithms.

The containers and algorithms in the C++ standard library use a variant of the philosophy of keeping independent concepts separate. Much of the library is based on the notion of a of sequence. Examples of sequences are arrays, sets, lists, maps, files.

A sequence has a beginning and an end. The end is one beyond the last element of the sequence. Positions in a sequence are represented by iterators.

```
begin                  end
  |                     |
  v                     v
 XXX -> .... -> XXX -> 0
```

Given an iterator for an element of a sequence, we can get to the next element using the `++` (increment) operator. Given an iterator for an element, we can access the element itself using the `*` (dereference) operator.

Given this simple notion, a surprising number of useful algorithms can be expressed. For example, this template function writes all elements of a container to output:

```
template<class C>
void print(C& s)
{
    C::iterator p=s.begin();

    while ( p!=s.end() ) {
        cout << *p;  // output
        p++;         // next
    }
}
```

The C++ standard library containers and algorithms are primarily the work of Alex Stepanov. A surprising number of containers and algorithms can be expressed using just a few kinds of iterators. Importantly, the resulting generic algorithms are efficient even compared to hand-crafted assembly code. For example, the C++ standard library algorithm, `sort()` is for many simple and realistic examples several times faster than the C standard library `qsort()` function. For more information about the C++ standard library and the principles underlying its design see [Koenig,1995] [Stepanov,1994].

7 Closing Remarks

Are the various facilities presented above object-oriented or not? Which ones? Using what definition of object-oriented?

In most contexts, I think these are the wrong questions. What matters is what ideas you can express clearly, how easily you can combine software from different sources, and how efficient and maintainable the resulting programs are. In other words, how you support good programming techniques and good design techniques matters more than labels and buzz words.

The fundamental idea is simply to improve design and programming through abstraction. You want to hide details, you want to exploit any commonality in a system, and you want to make this affordable.

I would like to encourage you not to make object-oriented a meaningless term. The notion of ‘‘object-oriented’’ is too frequently debased

- by equating it with good,
- by equating it with a single language, or
- by accepting everything as object-oriented.

I have argued that there are − and must be − useful techniques beyond object-oriented programming and design. However, to avoid being totally misunderstood, I would like to emphasize that I wouldn’t attempt a serious project using a programming language that didn’t at least support the classical notion of object-oriented programming. In addition to facilities that support object-oriented programming, I want − and C++ provides − features that go beyond those in their support for direct expression of concepts and relationships.

Several of the themes related to C++ programming style in this paper have been developed further in [Koenig,1995b]. The design and evolution of C++, including its most recent features, is discussed in [Stroustrup,1994].

8 Acknowledgements

Thanks to the OOPSLA’95 program committee for inviting me to give the talk upon which this paper is based, and especially to May Loomis for encouraging me to get this paper written. Carolyn Heaps transcribed the audio tape of my talk to produce the first draft of this paper. Tim Griffin and Christos Polyzois made many constructive comments.

9 References

[Koenig,1995] Andrew Koenig (editor): *Draft Proposed International Standard for Information Systems − Programming Language C++*. ANSI Standards Secretariat. CBEMA, 1250 Eye Street NW, Suite 200, Washington DC20005, USA. 1995.

[Koenig,1995b] Andrew Koenig and Bjarne Stroustrup: *A Foundation for Native C++ Styles*. Software − Practice & Experience. To appear 1995.

[Stepanov,1994] Alexander Stepanov and Meng Lee: *The Standard Template Library*. ISO Programming language C++ project. Doc No: X3J16/94-0095, WG21/N0482. May 1994.

[Stroustrup,1991] Bjarne Stroustrup: *The C++ Programming Language*. Addison-Wesley. 1991.

[Stroustrup,1994] Bjarne Stroustrup: *The Design and Evolution of C++*. Addison-Wesley. 1994.